Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

И. Л. Стефановский, Т. С. Семенченя

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА КОРПОРАТИВНЫХ ИНФОРМАЦИОННЫХ СИСТЕМ НА ОСНОВЕ ТЕХНОЛОГИИ *JEE*

ПРАКТИКУМ
по выполнению лабораторных работ
по одноименной дисциплине для студентов
специальности 1-40 05 01 «Информационные
системы и технологии (по направлениям)»
дневной и заочной форм обучения

Гомель 2021

УДК 004.75(075.8)
ББК 32.973.4я73
С79

Рекомендовано научно-методическим советом
факультета автоматизированных и информационных систем
ГГТУ им. П. О. Сухого
(*протокол № 10 от 04.05.2020 г.*)

Рецензент: зав. каф. «Промышленная электроника» ГГТУ им. П. О. Сухого
канд. техн. наук, доц. *Ю. В. Крышнев*

Стефановский, И. Л.

С79 Проектирование и разработка корпоративных информационных систем на основе технологии *JEE* : практикум по выполнению лаборатор. работ по одноим. дисциплине для студентов специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» днев. и заоч. форм обучения / И. Л. Стефановский, Т. С. Семенченя. – Гомель : ГГТУ им. П. О. Сухого, 2021. – 80 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

Практикум знакомит студентов с архитектурой, разработкой и использованием информационных систем уровня предприятия, а также содержит материал для выполнения лабораторных работ.

Для студентов специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» дневной и заочной форм обучения.

УДК 004.75(075.8)
ББК 32.973.4я73

Содержание

ВВЕДЕНИЕ

Целью изучения дисциплины является подготовка специалиста, владеющего знаниями и практическими навыками по архитектуре, разработке и использованию информационных систем (ИС) уровня предприятия.

Основными задачами изучаемой дисциплины являются:

– изучение студентами теоретических основ и технологий разработки ИС уровня предприятия;

– приобретение студентами практических навыков по проектированию и особенностям использования ИС в сетях на основе технологии *Java*;

– освоение студентами технологий: применения средств проектирования ИС на основе технологии *Java*, доступа к БД с использованием *Java*, разработки *web*-систем на основе технологии *Java*, сетевого и многопоточного программирования на основе технологии *Java*, технологий удаленного доступа к ИС, обработки данных и доступа к данным реляционных БД из внешних приложений.

Лабораторный практикум предназначен для выполнения лабораторных работ по курсу «Проектирование и разработка корпоративных информационных систем на основе технологии *JEE*» для студентов специальностей 1-40 05 01 «Информационные системы и технологии (по направлениям)».

Цель данного практикума:

– приобретение студентами практических навыков по проектированию и особенностям использования ИС в сетях на основе технологии *Java*;

– освоение студентами технологий: применения средств проектирования ИС на основе технологии *Java*, доступа к БД с использованием *Java*, разработки *web*-систем на основе технологии *Java*, сетевого и многопоточного программирования на основе технологии *Java*, технологий удаленного доступа к ИС, обработки данных и доступа к данным реляционных БД из внешних приложений.

– знакомство с функциональными возможностями современных языков программирования в создании серверных приложений;

– овладение навыками работы с основными инструментальными средствами конструирования и создания

прикладных программных продуктов различной сложности, используя различные технологии;

– приобретение студентами практических навыков решения задач с использованием современных методов программирования;

– обучение студентов самостоятельной работе и хорошей ориентации в области технологий и программных комплексов.

Лабораторная работа №1 «Использование *JDBC* для работы с БД»

Цель работы: научиться использовать *JDBC* для работы с БД в *Java*

Теоретические сведения

***JDBC* Драйверы**

Как правило, все поставщики СУБД распространяют библиотеки для работы со своими СУБД для разных платформ. Такие библиотеки в *Java* называются *JDBC* драйверами.

Раньше, когда *Java* ещё не занимала достойного места на рынке, считалось, что будут популярны разные виды драйверов, в т.ч. реализованные как вызовы более старого интерфейса *ODBC* (*JDBC-ODBC Bridge*). Сегодня практически любой драйвер представляет собой *JAR* файл с классами, реализующими интерфейсы из пакетов *java.sql* и *javax.sql*.

Для того, чтобы начать работать с СУБД в *classpath* необходимо добавить *JDBC* драйвер для соответствующей СУБД.

Основные классы *JDBC*:

– *DriverManager* – Основная задача данного класса – получение соединения с СУБД по *JDBC URL*

– *Connection* – Соединение с СУБД.

– *Statement* – Объекты этого класса можно порождать в контексте соединения и использовать для выполнения запросов.

– *PreparedStatement* – То же, что и *Statement*, но запрос разбирается только один раз, а потом в него подставляются параметры и он исполняется столько раз, сколько нужно.

– *CallableStatement* – То же самое, что и *PreparedStatement*, но для исполнения хранимых процедур и функций СУБД.

– *ResultSet* – Класс, представляющий собой курсор базы данных. Другими словами – это итератор по результатам выполненного запроса. Как правило, конкретная реализация *ResultSet* подкачивает строки выборки небольшими буферами для оптимизации.

– *Blob* – Один из типов значений колонки в выборке, который соответствует *Binary Large Object* типу конкретной СУБД. Из него можно читать как из файла с помощью *InputStream*.

– *Clob* – То же самое что и *Blob*, но для данных типа *Character Large Object*.

– *DataSource* – Данный интерфейс очень важен в *JavaEE*. Обычно он представляет собой пул соединений с СУБД, которые можно повторно использовать по мере необходимости и после чего возвращать обратно в пул без закрытия сетевого соединения

Основные приёмы работы

Получение соединения

Способы получения соединения с СУБД в *Java SE* и в *Java EE* сильно отличаются. В этом документе будет описан только обычный способ, характерный для *Java SE*:

```
public class LoadDriver {
  public static void main(String[] args) {
      try {
            // The newInstance() call is a work around for some
            // broken Java implementations
            Class.forName("com.mysql.jdbc.Driver").newInstance();
      } catch (Exception ex) {
            // handle the error
      }
  }
}
```

Сперва необходимо загрузить главный класс драйвера. В нашем случае – *com.mysql.jdbc.Driver*.

Потом используем *DriverManager*:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
try {
  Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost/test");
  // Do something with the Connection
} catch (SQLException ex) {
  // handle any errors
  System.out.println("SQLException: " + ex.getMessage());
  System.out.println("SQLState: "     + ex.getSQLState());
  System.out.println("VendorError: "  + ex.getErrorCode());
}
```

Судя по всему, при загрузке класса драйвера срабатывает блок статической инициализации, который регистрирует в *DriverManager*'е себя и что он обрабатывает соединения, *URL* которых имеет вид *jdbc:mysql://**

С соединениями примерно такая же беда, как с *Input*/*Output* потоками. Их необходимо закрывать, иначе очень быстро наступит предельно допустимое количество открытых соединений и к ней нельзя будет подключиться.

Итак, соединение получено. Теперь нужно научиться делать самые простые вещи. Выборку данных (как правило это *SQL* оператор *SELECT*) и их модификацию (*INSERT*, *UPDATE* и *DELETE*).

В *JDBC* операции чтения и модификации выполняются немного по-разному, т. к. при чтении нужно возвращать сами данные (*ResultSet*), а при модификации можно обойтись количеством строк, которые были изменены в хоте выполнения запроса.

Читаем из базы (*Select*):

```
Connection conn = null;
try {
   conn = ds.getConnection();
   PreparedStatement ps = conn.prepareStatement("SELECT *
FROM SOME_TABLE WHERE id=? AND name=? AND time=?");
   ps.setInteger(1, 10);
   ps.setString(2, "Mike");
   ps.setLong(3, System.currentTimeMillis());
```

```
    ResultSet rs = rs;
      while (rs.next()) {
       // Извлекаем значения колонок текущей строки выборки
       String ctxName = rs.getString("col1");
       int ctxId      = rs.getInteger("col2");
       int pending    = rs.getLong("col3");
      }
} finally {
    JdbcDaoHelper.safeClose(conn, log);
}
```

Обычно стоит написать утилитный метод, который закрывает соединение и игнорирует исключения (*JdbcDaoHelper*). Очень важно закрывать соединение при любых обстоятельствах.

Пишем в базу (*Update*, *Delete* и *Insert*):

```
Connection conn = null;
try {
conn = connectionPool.getConnection();
PreparedStatement ps = conn.prepareStatement(
"delete from SOME_TABLE where id=?"
);
ps.setInt(1, someId);
ps.execute();
} finally {
JdbcDaoHelper.safeClose(conn, log);
}
```

Задание на лабораторную работу

1. Создать приложение на *Java* для работы с БД с использованием *JDBC*.

2. Приложение должно обеспечивать ввод, редактирование, удаление данных, осуществлять выборки в соответствии с вариантом задания

3. Разработанное приложение должно использовать графический интерфейс пользователя для осуществления всех действий.

Варианты заданий:

1. ***Student***: *id*, Фамилия, Имя, Отчество, Дата рождения, Адрес, Телефон,

Создать массив объектов. Вывести:

a) список студентов заданного факультета;

b) списки студентов для каждого факультета и курса;

c) список студентов, родившихся после заданного года;

d) список учебной группы.

2. ***Customer***: *id*, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Номер банковского счета.

Создать массив объектов. Вывести:

a) список покупателей в алфавитном порядке;

b) список покупателей, у которых номер кредитной карточки находится в заданном интервале.

3. ***Patient***: *id*, Фамилия, Имя, Отчество, Адрес, Телефон, Номер медицинской карты, Диагноз.

Создать массив объектов. Вывести:

a) список пациентов, имеющих данный диагноз;

b) список пациентов, номер медицинской карты у которых находится в заданном интервале.

4. ***Abiturient***: *id*, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки.

Создать массив объектов. Вывести:

a) список абитуриентов, имеющих неудовлетворительные оценки;

b) список абитуриентов, средний балл у которых выше заданного;

c) выбрать заданное число *n* абитуриентов, имеющих самый высокий средний балл (вывести также полный список абитуриентов, имеющих полупроходной балл).

5. ***Book***: *id*, Название, Автор(ы), Издательство, Год издания, Количество страниц, Цена, Переплет.

Создать массив объектов. Вывести:

a) список книг заданного автора;

b) список книг, выпущенных заданным издательством;

c) список книг, выпущенных после заданного года.

6. ***House***: *id*, Номер квартиры, Площадь, Этаж, Количество комнат, Улица, Тип здания, Срок эксплуатации.

Создать массив объектов. Вывести:

a) список квартир, имеющих заданное число комнат;

b) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;

c) список квартир, имеющих площадь, превосходящую заданную.

7. ***Phone***: *id*, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Дебет, Кредит, Время городских и междугородных разговоров.

Создать массив объектов. Вывести:

a) сведения об абонентах, у которых время внутригородских разговоров превышает заданное;

b) сведения об абонентах, которые пользовались междугородной связью;

c) сведения об абонентах в алфавитном порядке.

8. ***Car***: *id*, Марка, Модель, Год выпуска, Цвет, Цена, Регистрационный номер.

Создать массив объектов. Вывести:

a) список автомобилей заданной марки;

b) список автомобилей заданной модели, которые эксплуатируются больше *n* лет;

c) список автомобилей заданного года выпуска, цена которых больше указанной.

9. ***Product***: *id*, Наименование, *UPC*, Производитель, Цена, Срок хранения, Количество.

Создать массив объектов. Вывести:

a) список товаров для заданного наименования;

b) список товаров для заданного наименования, цена которых не превосходит заданную;

c) список товаров, срок хранения которых больше заданного.

10. ***Train***: Пункт назначения, Номер поезда, Время отправления, Число мест (общих, купе, плацкарт, люкс).

Создать массив объектов. Вывести:

a) список поездов, следующих до заданного пункта назначения;

b) список поездов, следующих до заданного пункта назначения и отправляющихся после заданного часа;

c) список поездов, отправляющихся до заданного пункта назначения и имеющих общие места.

11. ***Bus***: Фамилия и инициалы водителя, Номер автобуса, Номер маршрута, Марка, Год начала эксплуатации, Пробег.

Создать массив объектов. Вывести:

a) список автобусов для заданного номера маршрута;

b) список автобусов, которые эксплуатируются больше 10 лет;

c) список автобусов, пробег у которых больше 100000 км.

12. ***Airlines***: Пункт назначения, Номер рейса, Тип самолета, Время вылета, Дни недели.

Создать массив объектов. Вывести:

a) список рейсов для заданного пункта назначения;

b) список рейсов для заданного дня недели;

c) список рейсов для заданного дня недели, время вылета для которых больше заданного.

13. ***Abiturient***: *id*, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки.

Создать массив объектов. Вывести:

a) список абитуриентов, имеющих неудовлетворительные оценки;

b) список абитуриентов, средний балл у которых выше заданного;

c) выбрать заданное число *n* абитуриентов, имеющих самый высокий средний балл (вывести также полный список абитуриентов, имеющих полупроходной балл).

14. ***Book***: *id*, Название, Автор(ы), Издательство, Год издания, Количество страниц, Цена, Переплет.

Создать массив объектов. Вывести:

a) список книг заданного автора;

b) список книг, выпущенных заданным издательством;

c) список книг, выпущенных после заданного года.

15. ***House***: *id*, Номер квартиры, Площадь, Этаж, Количество комнат, Улица, Тип здания, Срок эксплуатации.

Создать массив объектов. Вывести:

a) список квартир, имеющих заданное число комнат;

b) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;

c) список квартир, имеющих площадь, превосходящую заданную.

Контрольные вопросы

1. Какой интерфейс из пакета *java.sql* должен реализовывать каждый драйвер *JDBC*?
2. С помощью какого метода интерфейса *Connection* можно получить сведения о базе данных, с которой установлено соединение?
3. Какой интерфейс пакета *java.sql* используется, когда запрос к источнику данных является обращением к хранимой процедуре ?
4. Какой метод интерфейса *Statement* необходимо использовать при выполне-нии *SQL*-оператора *SELECT*, который возвращает объект *ResulSet* ?
5. Назначение *JDBC* в *Java* ?
6. Что такое *JDBC* ? Нарисуйте схему взаимосвязи программы на *Java* с базой данных с использованием *JDBC*.
7. Какие основные интерфейсы пакета *java.sql* ? Для чего каждый из них нужен?
8. Нарисуйте алгоритм работы *JDBC*-приложения.
9. Напишите команду *SQL* в *JDBC* для добавления строки в БД.
10. Напишите команду *SQL* в *JDBC* для выбора строк из БД.

Лабораторная работа № 2 «Создание серверного приложения»

Цель работы: научиться создавать серверные приложения в *JAVA*

Теоретические сведения

Java поддерживает протокол *TCP*/*IP*, во-первых, расширяя свой интерфейс потоков ввода-вывода, описанного в предыдущей лекции, и во вторых, добавляя возможности, необходимые для построения объектов ввода-вывода при работе в сети.

InetAddress

Java поддерживает адреса абонентов, принятые в *Internet*, с помощью класса *InetAddress*. Для адресации в *Internet* используются служебные функции, работающие с обычными, легко запоминающимися символическими именами, эти функции преобразуют символические имена в 32-битные адреса.

Фабричные методы

В классе *InetAddress* нет доступных пользователю конструкторов. Для создания объектов этого класса нужно воспользоваться одним из его фабричных методов. Фабричные методы – это обычные статические методы, которые возвращают ссылку на объект класса, которому они принадлежат. В данном случае, у класса *InetAddress* есть три метода, которые можно использовать для создания представителей. Это методы *getLocalHost*, *getByName* и *getAllByName*.

В приведенном ниже примере выводятся адреса и имена локальной машины, локального почтового узла и *WWW*-узла компании, в которой работает автор.

```
InetAddress Address = InetAddress.getLocalHost();
System.out.println(Address);
Address = InetAddress.getByName("mailhost");
System.out.println(Address);
InetAddress SW[] = InetAddress.getAllByNarne(“www.starwave.com");
System.out.println(SW);
```

У класса *InetAddress* также есть несколько нестатических методов, которые можно использовать с объектами, возвращаемыми только что названными фабричными методами:

- *getHostName*() возвращает строку, содержащую символическое имя узла, соответствующее хранящемуся в данном объекте адресу *Internet*.
- *getAddress*() возвращает байтовый массив из четырех элементов, в котором в порядке, используемом в сети, записан адрес *Internet*, хранящийся в данном объекте.
- *toString*() возвращает строку, в которой записано имя узла и его адрес.

Дейтаграммы

Дейтаграммы, или пакеты протокола *UDP* (*User Datagram Protocol*) – это пакеты информации, пересылаемые в сети по принципу "*fire-and-forget*" (выстрелил и забыл). Если вам надо добиться оптимальной производительности, и вы в состоянии минимизировать затраты на проверку целостности информации, пакеты *UDP* могут оказаться весьма полезными.

UDP не предусматривает проверок и подтверждений при передаче информации. При передаче пакета *UDP* по какому-либо адресу нет никакой гарантии того, что он будет принят, и даже того, что по этому адресу вообще есть кому принимать такие пакеты. Аналогично, когда вы получаете дейтаграмму, у вас нет никаких гарантий, что она не была повреждена в пути или что ее отправитель все еще ждет от вас подтверждения ее получения.

Java реализует дейтаграммы на базе протокола *UDP*, используя для этого два класса. Объекты класса *DatagramPacket* представляют собой контейнеры с данными, *a DatagramSocket* – это механизм, используемый при передаче и получении объектов *DatagramPacket*.

Сокеты "для клиентов"

TCP/*IP*-сокеты используются для реализации надежных двунаправленных, ориентированных на работу с потоками соединений точка-точка между узлами *Internet*. Сокеты можно использовать для соединения системы ввода-вывода *Java* с программами, которые могут выполняться либо на локальной машине, либо на любом другом узле *Internet*. В отличие от класса *DatagramSocket*, объекты класса *Socket* реализуют высоконадежные устойчивые соединения между клиентом и сервером.

В пакете *java.net* классы *Socket* и *ServerSocket* сильно отличаются друг от друга. Первое отличие в том, что *ServerSocket* ждет, пока клиент не установит с ним соединение, в то время, как обычный *Socket* трактует недоступность чего-либо, с чем он хочет соединиться, как ошибку. Одновременно с созданием объекта *Socket* устанавливается соединение между узлами *Internet*. Для создания сокетов вы можете использовать два конструктора:

- *Socket*(*String host*, *int port*) устанавливает соединение между локальной машиной и указанным портом узла *Internet*, имя которого было передано конструктору. Этот конструктор может возбуждать исключения *UnknownHostException* и *IOException*.
- *Socket*(*InetAddress address*, *int port*) выполняет ту же работу, что и первый конструктор, но узел, с которым требуется установить соединение, задается не строкой, а объектом *InetAddress*. Этот конструктор может возбуждать только *IOException*.

Из объекта *Socket* в любое время можно извлечь информацию об адресе *Internet* и номере порта, с которым он соединен. Для этого служат следующие методы:

- *getInetAddressQ* возвращает объект *InetAddress*, связанный с данным объектом *Socket*.
- *getPort*() возвращает номер порта на удаленном узле, с которым установлено соединение.
- *getLocalPort*() возвращает номер локального порта, к которому присоединен данный объект.

После того, как объект *Socket* создан, им можно воспользоваться для того, чтобы получить доступ к связанным с ним входному и выходному потокам. Эти потоки используются для приема и передачи данных точно так же, как и обычные потоки ввода-вывода, которые мы видели в предыдущей лабораторной работе:

- *getInputStream*() возвращает *InputStream*, связанный с данным объектом.
- *getOutputStream*() возвращает *OutputStream*, связанный с данным объектом.
- *close*() закрывает входной и выходной потоки объекта *Socket*.

Приведенный ниже очень простой пример открывает соединение с портом 880 сервера “*timehost*” и выводит полученные от него данные.

```
import java.net.*;
import java.io.*;
```

```
class TimeHost {
public static void main(String args[]) throws Exception {
int c;
Socket s = new Socket("timehost.starwave.com",880);
InputStream in = s.getInputStream();
while ((c = in.read()) != -1) {
System.out.print( (char) c);
}
s.close();
} }
```

Сокеты “для серверов”

Как уже упоминалось ранее, *Java* поддерживает сокеты серверов. Для создания серверов *Internet* надо использовать объекты класса *ServerSocket*. Когда вы создаете объект *ServerSocket*, он регистрирует себя в системе, говоря о том, что он готов обслуживать соединения клиентов. У этого класса есть один дополнительный метод *accept*(), вызов которого блокирует подпроцесс до тех пор, пока какой-нибудь клиент не установит соединение по соответствующему порту. После того, как соединение установлено, метод *accept*() возвращает вызвавшему его подпроцессу обычный объект *Socket*.

Два конструктора класса *ServerSocket* позволяют задать, по какому порту вы хотите соединяться с клиентами, и (необязательный параметр) как долго вы готовы ждать, пока этот порт не освободится.

- *ServerSocket*(*int port*) создает сокет сервера для заданного порта.
- *ServerSocket*(*int port*, *int count*) создает сокет сервера для заданного порта. Если этот порт занят, метод будет ждать его освобождения максимум *count* миллисекунд.

URL

URL (*Uniform Resource Locators* – однородные указатели ресурсов) – являются наиболее фундаментальным компонентом “Всемирной паутины”. Класс *URL* предоставляет простой и лаконичный программный интерфейс для доступа к информации в *Internet* с помощью *URL*.

У класса *URL* из библиотеки *Java* - четыре конструктора. В наиболее часто используемой форме конструктора *URL* адрес ресурса задается в строке, идентичной той, которую вы используете при работе с браузером:

URL(String spec)

Две следующих разновидности конструкторов позволяют задать *URL*, указав его отдельные компоненты:

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

URL(String protocol, String host, String file)

Четвертая, и последняя форма конструктора позволяет использовать существующий *URL* в качестве ссылочного контекста, и создать на основе этого контекста новый *URL*.

URL(URL context, String spec)

В приведенном ниже примере создается *URL*, адресующий *www*-страницу (поставьте туда свой адрес), после чего программа печатает свойства этого объекта.

```
import java.net.URL;
class myURL {
public static void main(String args[]) throws Exception {
URL hp = new URL("http://coop.chuvashia.edu");
System.out. println("Protocol: " + hp.getProtocol());
System.out.printin("Port: " +hp.getPort());
System.out.println("Host: " +        hp.getHost());
System.out.println("File: " + hp.getFile());
System.out.println("Ext: " + hp.toExternaLForm());
} }
```

Для того, чтобы извлечь реальную информацию, адресуемую данным *URL*, необходимо на основе *URL* создать объект *URLConnection*, воспользовавшись для этого методом *openConnection*().

URLConnection

URLConnection – объект, который мы используем либо для проверки свойств удаленного ресурса, адресуемого *URL*, либо для получения его содержимого. В приведенном ниже примере мы создаем *URLConnection* с помощью метода *openConnection*, вызванного с объектом *URL*. После этого мы используем созданный объект для получения содержимого и свойств документа.

```
import java.net.*;
import java.io.*;
class localURL {
public static void main(String args[]) throws Exception {
int c;
URL hp = new URL("http", "127.0.0.1", 80, "/");
URLConnection hpCon = hp.openConnection();
```

```
System.out.println("Date: " + hpCon.getDate());
System.out.println("Type: " + hpCon.getContentType());
System.out.println("Exp: " + hpCon.getExpiration());
System.out.println( "Last M: " + hpCon.getLastModified());
System.out.println("Length: " + hpCon.getContentLength());
if (hpCon.getContentLength() > 0) {
System.out.println("=== Content ===");
InputStream input = hpCon.getInputStream();
int i=hpCon.getContentLength();
while (((c = input. read()) != -1) && (--i > 0)) {
System.out.print((char) c);
}
input.close();
}
else {
System.out.println("No Content Available");
}
} }
```

Эта программа устанавливает *HTTP*-соединение с локальным узлом по порту 80 (у вас на машине должен быть установлен *Web*-сервер) и запрашивает документ по умолчанию, обычно это - *index.html*. После этого программа выводит значения заголовка, запрашивает и выводит содержимое документа.

Задание на лабораторную работу

Разработать серверное приложение с использованием асинхронных сокетов на языке программирования *java*. Сервер должен обслуживать несколько клиентов одновременно. Сервер выполняет действия в соответствии с таблицей. Клиент подключается к серверу, передает необходимые данные и получает результат**. Для реализации сервера использовать *NIO* и неблокирующие сокеты. Четные варианты используют протокол *TCP*, нечетные – *UDP*.** Создать на основе сокетов клиент/серверное визуальное приложение:

1. Клиент посылает через сервер сообщение другому клиенту.
2. Клиент посылает через сервер сообщение другому клиенту, выбранному из списка.
3. Чат. Клиент посылает через сервер сообщение, которое получают все клиенты. Список клиентов хранится на сервере в файле.

4. Клиент при обращении к серверу получает случайно выбранный сонет Шекспира из файла.

5. Сервер рассылает сообщения выбранным из списка клиентам. Список хранится в файле.

6. Сервер рассылает сообщения в определенное время определенным клиентам.

7. Сервер рассылает сообщения только тем клиентам, которые в настоящий момент находятся в *on-line*.

8. Чат. Сервер рассылает всем клиентам информацию о клиентах, вошедших в чат и покинувших его.

9. Клиент выбирает изображение из списка и пересылает его другому клиенту через сервер.

10. Игра по сети в “Морской бой”.

11. Игра по сети в “21”.

12. Игра по сети “Го”. Крестики-нолики на безразмерном (большом) поле. Для победы необходимо выстроить пять в один ряд.

13. Написать программу, сканирующую сеть в указанном диапазоне IP адресов на наличие активных компьютеров.

14. Прокси. Программа должна принимать сообщения от любого клиента, работающего на протоколе TCP, и отсылать их на соответствующий сервер. При передаче изменять сообщение.

15. Телнет. Создать программу, которая соединяется с указанным сервером по указанному порту и производит обмен текстовой информацией.

Контрольные вопросы

1. Как получить содержимое страницы, используя его *URL* ?
2. Как подключиться к серверу используя протокол *TCP* в *Java*?
3. Как подключиться к серверу используя протокол *UDP* в *Java*?
4. Порядок организации клиент-серверного взаимодействия с использованием протокола *TCP* в *Java*?
5. Порядок организации клиент-серверного взаимодействия с использованием протокола *UDP* в *Java*?
6. Порядок организации клиент-серверного взаимодействия с использованием протокола *ICMP* в *Java*?
7. Порядок организации клиент-серверного взаимодействия с использованием протокола *HTTP* в *Java*?

8. Порядок организации клиент-серверного взаимодействия с использованием протокола *HTTPS* в *Java* ?

9. Как используется интерфейс *ResultSet* ? Назовите его методы.

10. Реализация *DNS*- запросов в *Java*.

Лабораторная работа № 3 «Разработка многопоточных приложений в *Java*»

Цель работы: научиться создавать многопоточные приложения в *Java*

Теоретические сведения

Класс *Thread* инкапсулирует все средства, которые могут вам потребоваться при работе с подпроцессами. При запуске *Java*-программы в ней уже есть один выполняющийся подпроцесс. Вы всегда можете выяснить, какой именно подпроцесс выполняется в данный момент, с помощью вызова статического метода *Thread.currentThread*. После того, как вы получите дескриптор подпроцесса, вы можете выполнять над этим подпроцессом различные операции даже в том случае, когда параллельные подпроцессы отсутствуют. В очередном нашем примере показано, как можно управлять выполняющимся в данный момент подпроцессом:

```
class CurrentThreadDemo {
public static void main(String args[]) {
Thread t = Thread.currentThread();
t.setName("My Thread");
System.out. println("current thread: " + t);
try {
for (int n = 5; n > 0; n--) {
System.out.println(" " + n);
Thread.sleep(1000);
} }
catch (InterruptedException e) {
System.out.println("interrupted");
}
} }
```

В этом примере текущий подпроцесс хранится в локальной переменной *t*. Затем мы используем эту переменную для вызова метода *setName*, который изменяет внутреннее имя подпроцесса на "*My Thread*", с тем, чтобы вывод программы был удобочитаемым. На следующем шаге мы входим в цикл, в котором ведется обратный отсчет от 5, причем на каждой итерации с помощью вызова метода *Thread.sleep*() делается пауза длительностью в 1 секунду. Аргументом для этого метода является значение временного интервала в миллисекундах, хотя системные часы на многих платформах не позволяют точно выдерживать интервалы короче 10 миллисекунд. Обратите внимание – цикл заключен в *try*/*catch* блок. Дело в том, что метод *Thread.sleep*() может возбуждать исключение *InterruptedException*. Это исключение возбуждается в том случае, если какому-либо другому подпроцессу понадобится прервать данный подпроцесс. В данном примере мы в такой ситуации просто выводим сообщение о перехвате исключения. Ниже приведен вывод этой программы:

```
C:\> java CurrentThreadDemo
current thread: Thread[My Thread,5,main]
5
4
3
2
1
```

Обратите внимание на то, что в текстовом представлении объекта *Thread* содержится заданное нами имя легковесного процесса – *My Thread*. Число 5 – это приоритет подпроцесса, оно соответствует приоритету по умолчанию, "*main*" – имя группы подпроцессов, к которой принадлежит данный подпроцесс.

Runnable

Не очень интересно работать только с одним подпроцессом, а как можно создать еще один? Для этого нам понадобится другой экземпляр класса *Thread*. При создании нового объекта *Thread* ему нужно указать, какой программный код он должен выполнять. Вы можете запустить подпроцесс с помощью любого объекта, реализующего интерфейс *Runnable*. Для того, чтобы реализовать этот

интерфейс, класс должен предоставить определение метода *run*. Ниже приведен пример, в котором создается новый подпроцесс:

```
class ThreadDemo implements Runnable {
ThreadDemo() {
Thread ct = Thread.currentThread();
System.out.println("currentThread: " + ct);
Thread t = new Thread(this, "Demo Thread");
System.out.println("Thread created: " + t);
t.start();
try {
Thread.sleep(3000);
}
catch (InterruptedException e) {
System.out.println("interrupted");
}
System.out.println("exiting main thread");
}
public void run() {
try {
for (int i = 5; i > 0; i--) {
System.out.println("" + i);
Thread.sleep(1000);
} }
catch (InterruptedException e) {
System.out.println("child interrupted");
}
System.out.println("exiting child thread");
}
public static void main(String args[]) {
new ThreadDemo();
} }
```

Обратите внимание на то, что цикл внутри метода *run* выглядит точно так же, как и в предыдущем примере, только на этот раз он выполняется в другом подпроцессе. Подпроцесс *main* с помощью оператора *new Thread*(*this*, "*Demo Thread*") создает новый объект класса *Thread*, причем первый параметр конструктора – *this* – указывает, что нам хочется вызвать метод *run* текущего объекта.

Затем мы вызываем метод *start*, который запускает подпроцесс, выполняющий метод *run*. После этого основной подпроцесс (*main*) переводится в состояние ожидания на три секунды, затем выводит сообщение и завершает работу. Второй подпроцесс – "*Demo Thread*" – при этом по-прежнему выполняет итерации в цикле метода *run* до тех пор, пока значение счетчика цикла не уменьшится до нуля. Ниже показано, как выглядит результат работы этой программы этой программы после того, как она отработает 5 секунд:

```
C:\> java ThreadDemo
Thread created: Thread[Demo Thread,5,main]
5
4
3
exiting main thread
2
1
exiting child thread
```

Приоритеты подпроцессов.

Если вы хотите добиться от *Java* предсказуемого независимого от платформы поведения, вам следует проектировать свои подпроцессы таким образом, чтобы они по своей воле освобождали процессор. Ниже приведен пример с двумя подпроцессами с различными приоритетами, которые не ведут себя одинаково на различных платформах. Приоритет одного из подпроцессов с помощью вызова *setPriority* устанавливается на два уровня выше *Thread*. *NORM_PRIORITY*, то есть, умалчиваемого приоритета. У другого подпроцесса приоритет, наоборот, на два уровня ниже. Оба этих подпроцесса запускаются и работают в течение 10 секунд. Каждый из них выполняет цикл, в котором увеличивается значение переменной-счетчика. Через десять секунд после их запуска основной подпроцесс останавливает их работу, присваивая условию завершения цикла *while* значение *true* и выводит значения счетчиков, показывающих, сколько итераций цикла успел выполнить каждый из подпроцессов:

```
class Clicker implements Runnable {
int click = 0;
```

```
private Thread t;
private boolean running = true;
public clicker(int p) {
t = new Thread(this);
t.setPriority(p);
}
public void run() {
while (running) {
click++;
} }
public void stop() {
running = false; }
public void start() {
t.start();
} }
class HiLoPri {
public static void main(String args[]) {
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
lo.start();
hi.start();
try Thread.sleep(-10000) {
}
catch (Exception e) {
}
lo.stop();
hi.stop();
System.out.println(lo.click + " vs. " + hi.click);
} }
```

По значениям, фигурирующим в распечатке, можно заключить, что подпроцессу с низким приоритетом достается меньше на 25 процентов времени процессора:

```
C:\>java HiLoPri
304300 vs. 4066666
```

Синхронизация

Когда двум или более подпроцессам требуется параллельный доступ к одним и тем же данным (иначе говоря, к совместно используемому ресурсу), нужно позаботиться о том, чтобы в каждый конкретный момент времени доступ к этим данным предоставлялся только одному из подпроцессов. *Java* для такой синхронизации предоставляет уникальную, встроенную в язык программирования поддержку. В других системах с параллельными подпроцессами существует понятие *монитора.* Монитор – это объект, используемый как защелка. Только один из подпроцессов может в данный момент времени владеть монитором. Когда под-процесс получает эту защелку, говорят, что он *вошел* в монитор. Все остальные подпроцессы, пытающиеся войти в тот же монитор, будут заморожены до тех пор, пока подпроцесс-владелец не выйдет из монитора.

У каждого *Java*-объекта есть связанный с ним неявный монитор, а для того, чтобы войти в него, надо вызвать метод этого объекта, отмеченный ключевым словом *synchronized*. Для того, чтобы выйти из монитора и тем самым передать управление объектом другому подпроцессу, владелец монитора должен всего лишь вернуться из синхронизованного метода:

```
class Callme {
void call(String msg) {
System.out.println("[" + msg);
try Thread.sleep(-1000) {}
catch(Exception e) {}
System.out.println("]");
} }
class Caller implements Runnable {
String msg;
Callme target;
public Caller(Callme t, String s) {
target = t;
msg = s;
new Thread(this).start();
}
public void run() {
target.call(msg);
} }
```

```
class Synch {
public static void main(String args[]) {
Callme target = new Callme();
new Caller(target, "Hello.");
new Caller(target, "Synchronized");
new Caller(target, "World");
}
}
```

Вы можете видеть из приведенного ниже результата работы программы, что *sleep* в методе *call* приводит к переключению контекста между подпроцессами, так что вывод наших 3 строк-сообщений перемешивается:

```
[Hello.
[Synchronized
]
[World
]
]
```

Это происходит потому, что в нашем примере нет ничего, способного помешать разным подпроцессам вызывать одновременно один и тот же метод одного и того же объекта. Для такой ситуации есть даже специальный термин – *race condition* (состояние гонки), означающий, что различные подпроцессы пытаются опередить друг друга, чтобы завершить выполнение одного и того же метода. В этом примере для того, чтобы это состояние было очевидным и повторяемым, использован вызов *sleep*. В реальных же ситуациях это состояние, как правило, трудноуловимо, поскольку непонятно, где именно происходит переключение контекста, и этот эффект менее заметен и не всегда воспроизводятся от запуска к запуску программы. Так что если у вас есть метод (или целая группа методов), который манипулирует внутренним состоянием объекта, используемого в программе с параллельными подпроцессами, во избежание состояния гонки вам следует использовать в его заголовке ключевое слово *synchronized*.

Взаимодействие подпроцессов

В *Java* имеется элегантный механизм общения между подпроцессами, основанный на методах *wait*, *notify* и *notifyAll*. Эти методы реализованы, как *final*-методы класса *Object*, так что они имеются в любом *Java*-классе. Все эти методы должны вызываться только из синхронизованных методов. Правила использования этих методов очень просты:

• *wait* – приводит к тому, что текущий подпроцесс отдает управление и переходит в режим ожидания – до тех пор пока другой под-процесс не вызовет метод *notify* с тем же объектом.

• *notify* – выводит из состояния ожидания первый из подпроцессов, вызвавших *wait* с данным объектом.

• *notifyAll* – выводит из состояния ожидания все подпроцессы, вызвавшие *wait* с данным объектом.

Ниже приведен пример программы с наивной реализацией проблемы поставщик-потребитель. Эта программа состоит из четырех простых классов: класса *Q*, представляющего собой нашу реализацию очереди, доступ к которой мы пытаемся синхронизовать; поставщика (класс *Producer*), выполняющегося в отдельном подпроцессе и помещающего данные в очередь; потребителя (класс *Consumer*), тоже представляющего собой подпроцесс и извлекающего данные из очереди; и, наконец, крохотного класса *PC*, который создает по одному объекту каждого из перечисленных классов:

```
class Q {
int n;
synchronized int get() {
System.out.println("Got: " + n);
return n;
}
synchronized void put(int n) {
this.n = n;
System.out. println("Put: " + n);
} }
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
```

```
public void run() {
int i = 0;
while (true) {
q.put(i++);
} } }
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while (true) {
q.get();
}
} }
class PC {
public static void main(String args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
} }
```

Хотя методы *put* и *get* класса *Q* синхронизованы, в нашем примере нет ничего, что бы могло помешать поставщику переписывать данные по того, как их получит потребитель, и наоборот, потребителю ничего не мешает многократно считывать одни и те же данные. Так что вывод программы содержит вовсе не ту последовательность сообщений, которую нам бы хотелось иметь:

```
C:\> java PC
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
```

```
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

Как видите, после того, как поставщик помещает в переменную *n* значение 1, потребитель начинает работать и извлекает это значение 5 раз подряд. Положение можно исправить, если поставщик будет при занесении нового значения устанавливать флаг, например, заносить в логическую переменную значение *true*, после чего будет в цикле проверять ее значение до тех пор, пока поставщик не обработает данные и не сбросит флаг в *false*.

Правильным путем для получения того же результата в *Java* является использование вызовов *wait* и *notify* для передачи сигналов в обоих направлениях. Внутри метода *get* мы ждем (вызов *wait*), пока *Producer* не известит нас (*notify*), что для нас готова очередная порция данных. После того, как мы обработаем эти данные в методе *get*, мы извещаем объект класса *Producer* (снова вызов *notify*) о том, что он может передавать следующую порцию данных. Соответственно, внутри метода *put*, мы ждем (*wait*), пока *Consumer* не обработает данные, затем мы передаем новые данные и извещаем (*notify*) об этом объект-потребитель. Ниже приведен переписанный указанным образом класс *Q*:

```
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
if (!valueSet)
try wait();
catch(InterruptedException e):
System.out.println("Got: " + n);
valueSet = false;
notify();
return n;
}
synchronized void put(int n) {
if (valueSet)
```

```
try wait(); catch(InterruptedException e);
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
} }
```

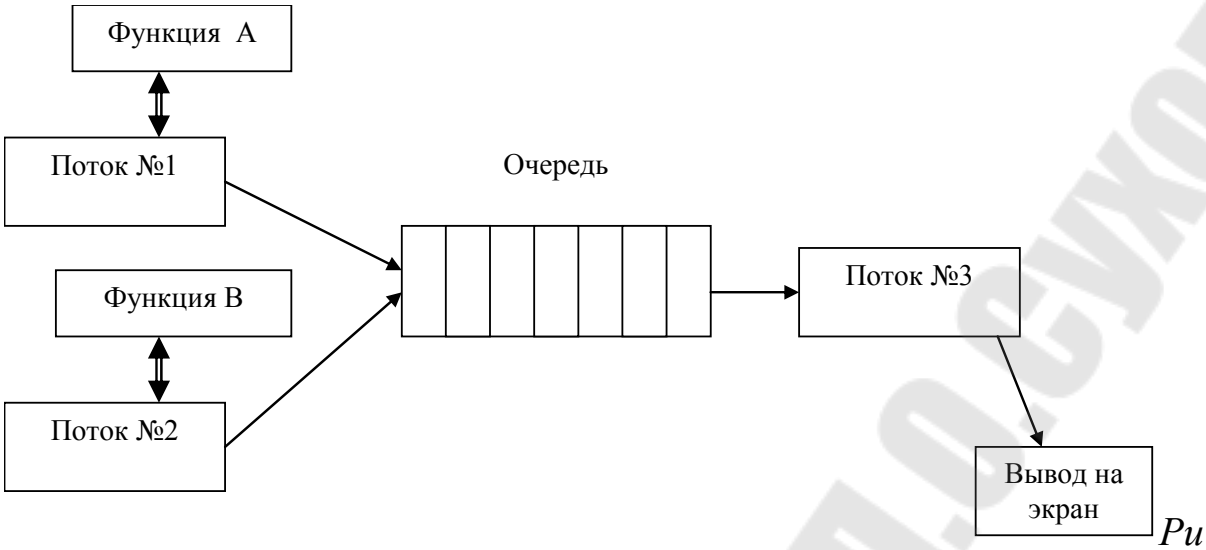
А вот и результат работы этой программы, ясно показывающий, что синхронизация достигнута:

```
C:\> java Pcsynch
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Клинч (*deadlock*)

Задание на лабораторную работу

Разработать приложение, в котором выполняется следующий алгоритм: два параллельных потока циклически выполняют вызов функций (согласно варианта). Каждый поток использует свою функцию. Результаты работы каждый поток помещает в общую очередь конечной длины. Третий поток забирает из очереди результаты работы функций и отображает их на экране в произвольной форме. Схема взаимодействия потоков приведена на рисунке 1.



Ри с. 1 – Схема многопоточного приложения

Очередь должна быть реализована с использованием двух объектов синхронизации Семафор (для потоков 1 и 2, и для потока 3). При заполнении очереди потоки 1 и 2 должны приостанавливаться функцией ожидания. При отсутствии результатов в очереди поток 3 также должен приостанавливаться функцией ожидания. Участок кода помещения результатов в очередь должен быть оформлен с использованием объекта синхронизации Монитор.

Потоки должны корректно завершаться при завершении приложения. Для этого в программе нужно предусмотреть объект синхронизации Событие, меняющий свое состояние при завершении приложения. В приложении должна быть предусмотрена возможность только однократного запуска через использование объекта синхронизации Мьютекс. В случае запуска второй копии программы должно выводиться соответствующее предупреждение.

Для реализации выбрать любые две функции из таблицы согласно варианта. Варианты заданий приведены в таблице 1.

Таблица 1

Варианты заданий

Вариант	Приложение	Функции сервера
1	Целочисленный калькулятор	Операции: сложение, вычитание, умножение, целочисленное деление, НОД (наибольший общий делитель)

Продолжение табл. 1

Вариант	Приложение	Функции сервера
2	Калькулятор чисел с плавающей точкой	Операции: сложение, вычитание, умножение, деление квадратный корень $\sqrt{x}$
3	Математический калькулятор №1	Операции: $sin\ x$, $cos\ x$, e^x, $log_{10}\ x$, $log_2\ x$
4	Математический калькулятор №2	Операции: $sin\ x$, $cos\ x$, $tg\ x$, $\sqrt[y]{x}$, x^y,
5	Редактор строки	Операции: сложение, замена одной подстроки другой, замена строчных букв прописными (в т.ч. для русских букв)
6	Редактор строки	Операции: «переворот» текста, перевод в транслитерацию, замена прописных букв строчными (в т.ч. для русских букв)
7	Работа с комплексными числами	Операции: сложение, вычитание, умножение
8	Восьмеричный калькулятор	Операции: Сложение, вычитание, умножение, остаток от деления
9	Троичный калькулятор	Операции: Сложение, вычитание, умножение, остаток от деления
10	Конвертер целых чисел №1	Преобразование **в** системы счисления: 16-тиричную, 8-ричную, 3-ичную, 2-ичную
11	Конвертер целых чисел №2	Преобразование **из** систем счисления: 16-тиричной, 8-ричной, 3-ичной, 2-ичной
12	Перевод числа от 0 до 99 в текстовый вид	Перевод на языки: русский, белорусский, английский

Окончание табл. 1

Вариант	Приложение	Функции сервера
13	Конвертер единиц длины	Перевод длины из метров в: километры, мили, ярды, футы, дюймы
14	Проверка делимости чисел	Проверка делимости целых чисел на: 2, 3, 5, 10
15	Преобразование текста №1	Замена всех: цифр на символ «*», пробелов на символ «_», русских букв на символ «#»
16	Преобразование текста №2	Преобразование текста: каждое новое слово с большой буквы, а остальной текст – маленькими (и для русских букв), каждое новое предложение с большой буквы, а остальной текст – маленькими (и для русских букв).
17	Преобразование текста №3	Замена в тексте: все гласные буквы становятся большими, все согласные буквы становятся большими, все знаки препинания дублируются (удваиваются).
18	Конвертер единиц объёма	Перевод объёма из литров в: галлоны, кубические футы, кубические дюймы
19	Перевод единиц времени	Перевод отрезка времени, заданного в виде числа суток в: число часов, число минут, число секунд
20	Логические операции	Операции с логическими операндами (Истина, Ложь): «И», «ИЛИ», «Исключающее ИЛИ».

Контрольные вопросы

1. Что такое процесс и поток? В чем разница между ними?
2. Чем определяется порядок передачи управления потокам?
3. Что такое приоритет потока, каков диапазон приоритетов?
4. Может ли быть изменен приоритет выполняющегося потока?
5. Будет ли низкоприоритетный поток исполняться, если существуют высокоприоритетные потоки, постоянно нуждающиеся в процессорном времени?
6. Какие есть способы реализации многозадачности в *Java*?
7. Что необходимо сделать для создания подкласса потоков (подкласса *Thread*)?
8. Когда запускается на выполнение метод *run*() подкласса *Thread*?
9. Что произойдет при запуске задачи, если метод *run* определить, например, так:
10. *public void run*(*int argument*) { … } ?

Лабораторная работа № 4 «Разработка *Web*-приложения с использованием технологии сервлетов»

Цель работы: научиться разрабатывать *Web*-приложения с использованием технологии сервлетов в *Java*

Теоретические сведения

Все сервлеты реализуют и используют интерфейсы и классы, объявленные в пакете *javax.servlet*. Помимо этого, все сервлеты должны "расширять" (ключевое слово *extend* в заголовке сервлета) класс *Servlet*, либо один из его потомков. Рассмотрим все сервлеты расширяющие специализированный именно для этих целей класс *HttpServlet*. Это связано с тем, что здесь рассматривается техника использования сервлетов для обработки *HTTP*-запросов.

В сервлете может быть объявлен метод *init*(). Данный метод вызывается всего лишь один раз за все "время жизни" сервлета. Это происходит либо при запуске сервлет-машины, либо при поступлении первого запроса к этому сервлету. В методе *init*() следует разместить все процедуры, которые необходимое выполнить при инициализации

сервлета. Делать это, не обязательно, так как метод *init*() не является тем методом, который обязательно должен присутствовать в каждом сервлете. Очень похожим методом является метод *destroy*(), который вызывается после завершения работы сервлет-машины. В данном методе следует размещать команды, предназначенные для освобождения ресурсов, занятых сервлетом во время своей инициализации. Аналогично методу *init*(), метод *destroy*() вызывается только один раз за все время существования экземпляра сервлета.

Рассмотрим метод *doPost*(), который вызывается после поступления запроса типа *POST* (т.е. когда пользователь щелкнет на кнопке *Submit* формы, у которой значение атрибута *METHOD* равно *POST*). В теле данного метода следует разместить все команды, отвечающие за обработку пользовательских данных. В качестве команд могут, естественно, использоваться вызовы других методов. Сервлет-машина передает методу *doPost*() в качестве входных параметров два объекта.

Объект *HttpServletRequest*. Он может быть использован для получения такой информации о запросе, как *IP*-адрес пользовательского компьютера, с которого был инициирован этот запрос, а также имена и значения всех переменных соответствующей пользовательской формы.

Объект *HttpServletResponse*. Его можно использовать для генерации ответа на пользовательский запрос.

Обычно в качестве ответа на запрос пользователь получает сгенерированную сервлетом (в общем случае, приложением, обрабатывающим запрос) страницу *HTML*, которая, в свою очередь, нередко содержит еще одну форму. Однако это не является правилом в ответ на запрос пользователь может получить все что угодно, т.е. файл произвольного типа. Одним из популярных форматов файлов, пересылающихся по *Internet*, является формат *.PDF* – стандартный формат программы *Acrobat Reader*. После получения такого файла на компьютере пользователя автоматически запускается данная программа, в которой сразу же открывается полученный файл. Для того чтобы пользователь смог узнать, какой тип ответа он получит от сервера, обрабатывающее запрос приложение должно непосредственно указать тип пересылаемых данных еще до отправки самого ответа. Чтобы сделать это, следует воспользоваться одним из методов класса *HttpServletResponse*.

Если бы вместо метода отправки информации *POST* был использован метод *GET*, то сервлет должен был бы реализовать метод *doGet*() вместо метода *doPost*().Оба метода похожи друг на друга, единственное отличие между ними заключается в их именах.

Сервлет-машина *JServ*

После установки и запуска сервлет-машины она готова к приему сервлет-запросов. Сервлет-запрос очень напоминает *CGI*-запрос. В некотором смысле эта аналогия распространяется и на настройку сервлет-машины. При настройке *JServ* должны быть определены каталоги сервлетов и соответствующие им псевдонимы (которые еще называются зонами). Зоны необходимо объявлять в конфигурационном файле *JServ jserv.conf*. Каждой зоне соответствует файл свойств, в котором указывается соответствующий этой зоне физический каталог сервера (каталог, в котором расположены файлы классов сервлетов). Приведем небольшой пример. Пусть сервлет-машина настроена таким образом, что в ней определена зона *testZone* с псевдонимом *testZone*. В этом случае в файле *jserv.conf* должна быть следующая строка:

ApJServMount /*testZone* /*testZone*

Также должен быть создан соответствующий данной зоне файл свойств *testZone.properties*. Одной из его строк будет примерно следующая:

repositors=/*home*/*vsharma*/*SERVLETS*/*JSERV*/*Apache-JServ*-1.*Ob*1/*testZone*

Таким образом, все сервлеты зоны *testZone* расположены в каталоге

/*home*/*vsharma*/*SERVLETS*/*JSERV*/*Apache-JServ*-1.O*b*1/*testZone*

В пределах сервлет-машины можно определить сразу несколько зон. Это необходимо делать для разделения различных типов приложений по отдельным логическим областям. Все зоны должны быть определены в файле *jserv.properties*.

Еще раз обратимся к примеру. Из приведенных ниже строк можно сделать вывод, что данная сервлет-машина настроена с двумя зонами:

zones=example, *testZone*

testZone.properties=/*local*/*vsharraa*/*SERVLET*/*JSERV*/*Apache-*

JServ-1.O*b*1/*testZone*/ *testZone.properties*

Также отсюда можно узнать полный путь к файлу свойств *testZone.properties* зоны *testZone*.

Другим, не менее важным элементом файла *jserv.properties* является список каталогов и файлов с расширением *.jar*, которые должны быть включены в переменную *CLASSPATH*. (Как-никак, а сервлет-машина *JServ* запускает виртуальную машину *Java*, просматривающую переменную *CLASSPATH* для того, чтобы получить список каталогов, в которых хранятся классы *Java*.) Этот список определяется с помощью параметра *wrapper*, *CLASSPATH*. Для каждого каталога или *.jar*-файла, который необходимо включить в переменную *CLASSPATH*, должна быть отведена отдельная строка. Рассмотрим пример:

wrapper.classpath=/local/vsharma/SERVLETS/JSERV/Apache-JServ-1.O*b*1/*src/java/*

Apache-JServ.jar

*wrapper.classpath=/local/vsharma/SERVLETS/JSDK/JSDK*2.0/*lib/jsdk.jar*

wrapper.classpath=/local/vsharma

wrapper.classpath=/local/vsharma/JAF/jaf/activation.jar

wrapper.classpath=/local/vsharma/JAVAMAIL/javamail-1.1/*mail.jar*

Для того чтобы внесенные изменения данного файла вступили в силу, после добавления новой строки *wrapper.classpath* или внесения каких-либо других изменений необходимо перезапустить сервлет-машину *JServ*. Чтобы сделать это, потребуется заодно перезапустить и сам *Web*-сервер *Apache*.

Теперь, учитывая приведенную выше конфигурацию сервлет-машины, посмотрим, что должно произойти, когда пользователь запросит информацию по адресу: *http*: *X*/*myServer*/*testZone*/*MyServlet*. *Web*-сервер воспримет данный запрос как обращение к сервлету, расположенному в зоне *testZone*. После этого сервер передаст запрос сервлет-машине, которая попытается выяснить, существует ли экземпляр класса *MyServer.class*. Если он существует, сервлет-машина создаст поток, в котором будет выполняться данный экземпляр (если полученный запрос создан из пользовательской формы, поддерживающей метод передачи информации *POST*, то сразу же будет вызван метод *doPost*()). Если же такого экземпляра пока что не

существует, то он будет незамедлительно создан, и первым вызванным методом будет метод *init*(). Затем, после выполнения всех инициализирующих команд, будет вызван метод *doPost*().

Сервлет-машина будет "искать" класс *MyServlet* в каталоге, который был указан в файле свойств *testZone*, *properties* зоны *testZone*. Это каталог

/*home*/*vsharma*/*SERVLETS*/*JSERV*/*Apache-JServ*-1.*Ob*1/*testZone*

Создаем первый сервлет

Ниже приведен исходный код сервлета, который, по предположению, находится в зоне *testZone*. Начнем, однако, с рассмотрения исходного кода документа *HTML* (файл *firstServlet.html*), содержащего пользовательскую форму:

```
<HTML>
<HEAD>
</HEAD> <BODY>
<FORM ACTION='http: //myServer/testZone,/FirstServlet' METHDO='POST ' >
<INPUT TYPE='submit' NAME='submit' VALUE='Execute'> </FORM> </BODY> </HTML>
```

Когда пользователь щелкнет на кнопке *Submit*, на сервере запускается следующий класс (сервлет). Файл *FirstServlet*. *Java* имеет такой вид:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class FirstServlet extends HttpServlet
{
public void doPost (HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
{
 PrintWriter out;
res.setContentType("text/html");
out = res.getWriter();
```

```
out .println ("<HTML><HEAD></HEAD>") ;
out.println("<BODY bgcolor=\"#FFFFFF\">");
out.println("<B>Первый сервлет </B>");
out. println ("</BODY></HTML>" );
out.close ();
}
}
```

Для того чтобы протестировать работоспособность данного сервлета, для начала перенесите файл *firstServlet.html* в корневой каталог файлов *Web*-сервера (в терминологии *Web*-сервера *Apache* – *DocumentRoot*). Откомпилируйте класс *FirstServlet*. *Java* (заметьте, что для успешной компиляции этого класса в переменную *CLASSPATH* обязательно должен быть включен файл *jsdk.jar*) и перенесите файл *FirstServlet.class* в каталог, соответствующий зоне *testZone*. После этого можно смело открыть в обозревателе *Internet* страницу *firstServlet.html* и щелкнуть на кнопке *Execute* (Выполнить).

Когда пользователь щелкнет на кнопке *Execute*, сервер запустит сервлет, обрабатывающий данные формы.

Рассмотрим исходный код сервлета *FirstServlet*. Вначале импортируются нескольких пакетов, необходимых для работы сервлета. Первым импортируется пакет *java.io*, так как ответ сервлета на поступивший запрос (страница *HTML*) будет отправлен на пользовательский компьютер через выходной поток. В пакетах *Javaservlet* и *Java*, *servlet.http* находятся все классы, использующиеся в данном сервлете. Класс *FirstServlet.class* реализует метод *doPost*(), который принимает два входных параметра: объект *HttpServletRequest* и объект *HttpServletResponse*. Этот класс является расширением класса *HttpServlet*. Заметьте, что все сервлеты должны реализовывать интерфейс *Servlet*. Класс *HttpServlet* является расширением класса *GenericServlet*, который, в свою очередь, реализует интерфейс *Servlet*.

Одной из самых важных операций, выполняемых в методе *doPost*(), является получение дескриптора выходного потока. Для этого используется метод *getWriter*() класса *HttpServletResponse*, который возвращает объект *PrintWriter.out = res.getWriter*();

Все данные, записанные в этот объект, будут отправлены обратно обозревателю *Internet* пользовательского компьютера, с которого был передан запрос на сервер. Как упоминалось ранее, обязательным является указание типа информации, содержащейся в

ответе. Поскольку в данном примере в ответ на пользовательский запрос сервер отсылает ответ в виде страницы *HTML*, тип содержимого должен быть установлен в *text*/*html*.

res.*setContentType*("*text*/*html*") ;

Все методы *out*.*println*() используются для вывода строк исходного кода *HTML*-документа, который будет отображен клиентским обозревателем *Internet*. С точки зрения языка *HTML*, сгенерированная сервером *Web*-страница ничем не отличается от обычных *Web*-страниц, статически хранящихся на сервере.

Переменные пользовательской формы

Разобравшись с общим механизмом работы сервлета, рассмотрим способ, с помощью которого сервлет может извлечь данные пользовательской формы, переданные вместе с запросом. Для того чтобы это сделать, необходимо воспользоваться некоторыми достаточно полезными методами, объявленными в классе *HttpServletRe-quest*. Приведем небольшой пример. Предположим, что пользовательская форма содержит текстовое поле ввода и набор переключателей. Рассматриваемый сервлет считывает значения всех переменных формы и отображает их вместе с именами переменных.

Исходный код документа *HTML* имеет следующий вид:

```
<HTML>
<HEAD>
</HEAD> <BODY>
<FORM ACTION='http://myServer/testZone/ShowFormVariables' METHOD='POST'>
<INPUT TYPE='text' NAME='firstName' VALUE="XBR>
<INPUT TYPE='radio' NAME='rd' VALUE=' val1 "XBR>
<INPUT TYPE='radio' NAME='.rd' VALUE=' val2 '><BR>
<INPUT TYPE='submit' NAME='Submit' VALUE='Execute'>
</FORM> </BODY> </HTML>
```

Исходный код сервлета имеет такой вид:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;
public class ShowFormVariables extends HttpServlet
{
public void doPost (HttpServletRequest req, HttpServletResponse
res)
throws ServletException, lOException
{
PrintWriter out;
res.setContentType("text/html");
out = res.getWriter();
out.println("<HTML><HEAD></HEAD>");
out.println("<BODY bgcolor=\"#FFFFFF\">");
String txt = req.getParameterValues("firstName")[0];
out.println("Значение текстового поля ввода, firstName, равно: " +
txt);
out.println("<BR>A вот все переменные формы и их
значения<BR>");
Enumeration names = req.getParameterNames();
while(names.hasMoreElements()) {
String name = (String)names.nextElement();
String value = req.getParameterValues(name)[0];
out.println("Имя: " + name + "Значение: " + value + "<BR>"); }
out. println ("</BODY></HTML>" );
out.close (); } }
```

В классе *HttpServletRequest* есть метод *getParameterValues*() , который возвращает значение любой переменной формы по имени этой переменной. Возвращаемый результат представляет собой массив строк *String*. Почему именно массив строк, а не просто отдельную строку? Все дело в том, что некоторые переменные формы могут иметь несколько значений, например, те переменные, что соответствуют группам флажков. Представим, что форма содержит четыре флажка, объединенных в одну группу (напомним, что в этом случае они все имеют одинаковое имя). Если пользователь выберет, скажем, три флажка и щелкнет на кнопке *Submit*, то на сервер будет отправлено три значения, соответствующих одной переменной. Естественно, что самым удобным способом представления таких данных является массив. Если же переменной формы соответствует всего лишь одно значение, к нему можно достаточно просто

обратиться как к первому элементу массива, возвращаемого методом *getParameterValues*().

Именно таким образом извлекается значение переменной *firstName*:

String txt = req.getParameterValues("*firstName*")*[0]*;

В данном классе приведен также альтернативный метод доступа к переменным формы, который не предполагает предварительного знания их имен. Метод *getParameterNames*() возвращает объект *Enumeration*, в котором содержатся все имена переменных формы (включая кнопку *Submit*).

Enumeration names = req.getParameterNames();

С помощью этого объекта можно получить значения всех имен переменных формы. Каждое имя хранится в виде объекта *String*. Передав имя переменной в качестве входного параметра методу *getParameterValues*(), в ответ можно получить значение данной переменной.

Результат работы такого сервлета выглядит примерно так:

Значение текстового поля ввода, *firstName*, равно: Это текстовое поле А вот все переменные формы и их значения

Имя: *rd* Значение: *val*2 Имя: *Submit* Значение: *Execute* Имя: *firstName* Значение: Это текстовое поле

В данном случае перед тем, как щелкнуть на кнопке *Submit*, пользователь ввел в текстовом поле ввода строку Это текстовое поле и активизировал второй переключатель (первый переключатель при этом автоматически был снят).

Работаем с *cookies*

По умолчанию каждый пользовательский запрос на получение *Web*-страницы или обработку данных формы не зависит от предыдущих запросов, сделанных этим же пользователем. С одной стороны, это, может быть, и хорошо, однако, с другой стороны, это накладывает строгие ограничения на процесс накопления информации, который происходит по мере посещения пользователем данного *Web*-узла. Одним из классических примеров, характеризующих эту проблему, является пример ввода пользователем своего имени при регистрации на *Web*-узле. Если пользователь затем вызовет какой-то другой сервлет, то он не будет

иметь никаких средств, с помощью которых можно было бы определить имя пользователя, введенное ранее при входе на *Web*-узел. Таким образом, второй сервлет не сможет, например, поприветствовать пользователя, что является, в общем-то, стандартной практикой. Для того чтобы обойти это ограничение, была разработана концепция файлов *cookie*. В принципе это не единственный подход к решению данной проблемы, однако вторая технология – технология сеансов (*Sessions*) – будет рассмотрена далее в главе.

Файл *cookie* – это файл небольшого размера, который создается серверным приложением и размещается на компьютере пользователя. Когда пользователь обращается к *Web*-узлу, серверное приложение которого создало файл *cookie*, этот файл автоматически пересылается данному *Web*-серверу. Обратимся вновь к примеру, с двумя сервлетами. Первый сервлет может создать файл *cookie*, сохранив в нем такие данные о пользователе, как его имя, фамилию и т.п. Второй (как, впрочем, и любой следующий по времени вызова) сервлет, в свою очередь, может обратиться к только что созданному файлу *cookie*, считать из него всю информацию и отобразить какое-либо "персонализированное" сообщение пользователю.

Однако, прежде чем перейти к более подробному рассмотрению технологии файлов *cookie*, следует сказать несколько слов об ограничениях, которые, похоже, не обходят уже ни одно явление в компьютерном мире. Большинство обозревателей *Internet* накладывают ограничения на размер файла *cookie* и общее количество *cookie*, которые могут быть установлены на пользовательском компьютере приложениями одного *Web*-сервера. Таким образом, вполне понятно, что при всем желании в файле *cookie* нельзя сохранить большой объем информации. Если все же без этого не обойтись, то в таком случае придется воспользоваться технологией сеансов, которая будет рассмотрена в следующих разделах данной главы.

В приведенном ниже примере рассматривается механизм установки файла *cookie* на пользовательском компьютере и способ его чтения серверным приложением. Чтобы протестировать соответствующие сервлеты, для начала необходимо переместить файл *cookieExample.html* в корневой каталог *Web*-сервера (*DocumentRoot*). Затем следует откомпилировать классы *SetCookie* и *GetCookie* и переместить их в зону *example*.

Открыв файл *cookieExample.html* с помощью обозревателя *Internet*, пользователь увидит форму, в которой ему необходимо заполнить два поля: поле с именем и

поле с фамилией пользователя. После щелчка на кнопке *Submit* вызывается сервлет *SetCookie*. В функции данного сервлета входит чтение информации, введенной пользователем в форму, и ее запись в файл *cookie* с именем *MYSITE*. Файл *cookie* имеет название и хранящееся в этом файле значение. Поскольку в одном файле *cookie* можно хранить лишь одно значение, то здесь возникает маленькая проблема, так как в данном случае нам необходимо записать в *cookie* как имя пользователя, так и его фамилию. Эта проблема легко решается путем использования объекта *String*, в который записываются оба эти значения, разделенные некоторым специальным символом. Символ-разделитель используется затем серверным приложением для того, чтобы отделить имя пользователя от его фамилии.

Отработав, сервлет *SetCookie* передает управление другому сервлету – сервлету *GetCookie*. Данный сервлет считывает значение файла *cookie MYSITE* и выделяет из него имя и фамилию пользователя. Затем он генерирует страницу *HTML*, помещает туда эту информацию (а также всю информацию, которая находится в остальных файлах *cookie*, полученных от данного клиента) и возвращает ее пользователю в качестве ответа.

Файл *cookieExample.html* имеет следующий вид:

<HTML><HEAD></HEAD>

<BODY>

Данный пример наглядно демонстрирует процесс установки и чтения файлов *cookie* с использованием для этой цели сервлетов.

Первый сервлет сохраняет в файле *cookie* информацию об имени и фамилии пользователя, второй же сервлет используется для чтения-этих значений:

<FORM ACTION='/example/SetCookie' METHOD='POST'>

<TABLE>

<TR><TD>Имя</*TD*>

<TD><INPUT TYPE='text' NAME='firstName' SIZE=' 10 ' >

</TD></TR>

<TR><TD>Фамилия</TD>

<TD><INPUT TYPE='text' NAME='lastName' SIZE=' 10 ' >

```
</TD></TR>
    <TR> <TD> </TD>
    <TD> INPUT TYPE='submit' NAME='submit' VALUE= ' Ok' >
</TD> </TR>
    </TABLE></FORM></BODY></HTML>
```

Файл *SetCookie.java* имеет такой вид:

```
Import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SetCookie extends HttpServlet {
String cookieName = "MYSITE";
String cookieDomain = "YOUR_DOMAIN";
String delim = "DLM:";
int maxAge = 3600; // 3600 секунд = 1 час
String redirectUrl = "/example/GetCookie";
public void doGet (HttpServletRequest req, HttpServletResponse res)
{
doPost(req, res);
}
public void doPost (HttpServletRequest req, HttpServletResponse
res)
{
try{
String firstName = "", lastName = "";
Enumeration values = req. getParameterNames ()•;
while(values.hasMoreElements ()) {
String name = (String)values.nextElement();
String value = req.getParameterValues(name)[0];
if(name.equalsIgnoreCase("firstName"))
firstName = value; if(name.equalsIgnoreCase("lastName"))
lastName = value; }
String cookieVal = firstName + delim + lastName; Cookie ck = new
Cookie(cookieName, cookieVal); ck.setDomain(cookieDomain);
ck.setMaxAge(maxAge);
ck.setPath("/");
res.addCookie(ck);
```

res.sendRedirect(*redirectUrl*); }
catch(*Exception ex*){} } }

В состав пакета разработчика сервлетов *Java* (*JSDK*) входит класс *Cookie*, который, как несложно догадаться по его названию, служит для представления файлов *cookie*. С помощью данного класса можно создать новый файл *cookie*, а затем отправить его на пользовательский компьютер вместе с ответом на запрос. По большому счету, файл *cookie* имеет всего лишь два параметра, которые можно изменять – это имя и значение. Помимо этого, с помощью метода *setMaxAge*() можно определить так называемый "срок жизни" файла *cookie*. Что же представляет собой этот срок? Это время в секундах, отсчитываемое от момента создания файла *cookie* до того момента, когда информация, хранящаяся в нем, "морально" устаревает, т.е. становится непригодной.

Также для файла *cookie* можно определить имя домена и путь. Имя домена определяет группу серверов, которым пользовательский компьютер должен переслать файл *cookie* вместе с запросом. По умолчанию файл *cookie* отправляется на тот сервер, приложением которого он был установлен. Имя домена полезно тогда, когда файл *cookie* должен передаваться на несколько серверов, принадлежащих одному и тому же домену. Приведем небольшой пример. Пусть несколько серверов принадлежат домену *xyz*. *com*. Предположим, что имена этих серверов имеют вид *abc.xyz.com*, *def.xyz.com* и т.д. Если файл *cookie* был установлен приложением сервера *def.xyz.com*, и необходимо обеспечить его рассылку любому серверу данного домена (*abc.xyz.com*, *def.xyz.com* и т.д.), к которому только поступит пользовательский запрос, то для достижения этой цели следует воспользоваться методом *setDomain*(), как показано ниже:

cookie.setDomain("*.xyz.com*");

Практически аналогичным образом можно определить, чтобы файл *cookie* пересылался только тогда, когда пользователь запрашивает адрес *URL*, принадлежащий какой-то определенной части *Web*-узла. По умолчанию, файл *cookie* пересылается только тем приложениям или *Web*-страницам, которые расположены в каталоге, где находится приложение или страница, создавшие данный файл. Для того чтобы сделать файл *cookie* доступным для всех приложений данного *Web*-узла, необходимо воспользоваться методом *setPath*(). Приведенная ниже строка кода определяет, что файл *cookie* будет

доступен всем приложениям или страницам *Web*-узла:

cookie.setPath("/");

А вот небольшой фрагмент кода из файла *SetCookie.Java*, в котором непосредственно производится создание и установка файл *cookie*:

```
//Создание нового файла cookie с определенным именем и значением 1
Cookie ck = new Cookie(cookieName, cookieVal);
//Указание имени домена, срока использования и пути
ck.setDomain(cookieDomain); 1 ck.setMaxAge(maxAge);
ck.setPathCV");
//Добавление файла cookie к ответу, отправляемому на компьютер пользователя
res.addCookie(ck);
```

Одной из самых последних команд сервлета *SetCookie* является вызов метода *sendRedirect*() класса *HttpServletResponse*. Данный метод используется для того, чтобы вместо стандартного ответа в виде страницы *HTML* (или чего-то там еще) пользовательский обозреватель *Internet* получил команду сделать переход на другой адрес *URL*. В рассматриваемом случае обозреватель клиентского компьютера получает команду перейти по адресу /*example*/*GetCookie*, т.е. обратиться к еще одному сервлету – сервлету *GetCookie*:

```
res.sendRedirect(redirectUrl);
```

Файл *GetCookie.Java* имеет следующий вид.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class GetCookie extends HttpServlet
{
String cookieName = "MYSITE";
String delim = "DLM:";
public void doGet (HttpSevletRequest req, HttpServletResponse res)
{
doPost(req, res); }
public void doPost (HttpSevletRequest req, HttpServletResponse res)
{
```

```
try{
PrintWriter out = null;
res.setContentType("text/html") ;
out = res.getWriter();
Cookie[] cookieList = res.getCookies () ;
int size = cookieList.length;
for(int i=0; i<size; i++)
{
Cookie c = cookieList [i] ;
String name = c.getName ();
String value = c.getValue();
out.printIn("Имя cookie " + name + "<BR>");
if(name.equals(cookieName))
int ind = value.indexOf(delim);
String firstName = value.substring(0, ind);
String lastName = value.substring(ind + delim.length());
out.println("Значения, сохраненные в данном файле cookie:<BR>");
out.println("Имя: " + firstName + "Фамилия: " + lastName);
} }
out.close(); }catch(Exception ex){} } }
```

Задание на лабораторную работу

В соответствии с вариантом написать сервлет для вычисления значения функции. Ввод/вывод исходных данных осуществляется через *web*-страницы, генерируемые сервлетом. Варианты заданий приведены в таблице 2.

Таблица 2

Варианты заданий

Вариант	Вид функции	Вариант	Вид функции
1	$b=\frac{1+\cos^2(x+z)}{\lvert x^3-2y^2\rvert}$	16	$b=x+\frac{\sqrt[3]{zy}}{y+\cos x}$
2	$b=\frac{\ln^2\lvert z\rvert}{\sqrt[3]{\lvert x\rvert+\lvert y\rvert}}$	17	$b=\lg\left(\sqrt{e^{x-y}+x^{\lvert y\rvert}+z}\right)$

Окончание табл. 2

Вариант	Вид функции	Вариант	Вид функции
3	$b = \dfrac{y^3}{x + y^3 \cos^2 z}$	18	$b = 1 + \dfrac{x^2 + 1}{3 + y^2} + \sin 2z$
4	$b = \sqrt{x + \sqrt[4]{\lvert y \rvert}} + \cos^2 z$	19	$b = \lvert \cos x + \cos y \rvert + 2\sin^2 z$
5	$b = \dfrac{\sqrt[3]{e^{\sin x}} \cdot \cos y}{z^2 + 1}$	20	$b = \dfrac{\ln(y^3)(z - x/2)}{2\cos^2 x}$
6	$b = z(tg\, y - e^{-(x+3)})$	21	$b = \sqrt{10\left(\sqrt[3]{z} + x^{(y+2)}\right)}$
7	$b = \lvert x - y \rvert (\sin^2 z + tg\, z)$	22	$b = (\sin z)^2 + \lvert x + y \rvert$
8	$b = \sqrt{y + \sqrt[3]{x}} - 1 + 2z$	23	$b = e^{2z} - \sqrt[3]{y \lvert x \rvert}$
9	$b = x(tg\, z + \cos^2 y)$	24	$b = e^{(x-1)} + \sin y$
10	$b = e^{\lvert x - y \rvert} (tg^2 z + 1)^x$	25	$b = \sqrt{\lvert z \rvert e^{-(y + x/2)}}$
11	$b = \cos^2 z + tg\, 2x + \lvert y \rvert$	26	$b = \dfrac{4y^2 e^{2x} \sin^2 z}{3z^3 + \ln x}$
12	$b = 5tg\, z - 4y^2 + \lvert xy \rvert$	27	$b = \dfrac{\sqrt{y \ln x} - z x^2}{1 + tg^2 x^2} x$
13	$b = (z - x)\dfrac{y - \ln z}{1 + (y - x)^2}$	28	$b = \dfrac{\lg\left(y + \sqrt{z + x^2}\right)}{y + x^2}$
14	$b = y^z + \sqrt{\lvert x \rvert + \lvert y \rvert}$	29	$b = \dfrac{x^2 + 4}{\sin^2 z^2 + x/2} y$
15	$b = \dfrac{\lg\left(\sqrt{x} + \sqrt{y} + 2\right)}{\lvert 2z \rvert}$	30	$b = \dfrac{\sin x + \sqrt{\lvert z - y \rvert}}{y(x - 2) + x^2}$

Контрольные вопросы

1. Что такое сервлет и что такое *JSP* ? Их сходство и различия.
2. Нарисуйте функциональную схему работы веб-приложения на основе *Java*-сервлетов и *JSP*.
3. Какие методы работают в течение жизненного цикла сервлета ?

4. С помощью каких классов или интерфейсов чаще всего создают сервлет ? Напишите код.

5. Что такое дескриптор развертывания, для чего он нужен ?

6. Из каких папок состоит веб-приложение, что в этих папках находится ?

7. Какие преимущества и какие недостатки у сервлетов ?

8. Для чего нужно межсервлетное взаимодействие, какими механизмами его можно обеспечить (назовите классы, методы) ?

9. Приведите механизмы отслеживания сессии.

10. Недостатки технологии сервлетов при реализации в многопоточном гетерогенном окружении.

Лабораторная работа № 5 «Разработка *Web*-приложения с использованием *JSP*-страниц»

Цель работы: научиться разрабатывать *Web*-приложения с использованием технологии *JSP*-страниц в *Java*

Теоретические сведения

Структура *JSP* страницы - это что-то среднее между сервлетом и *HTML* страницей. *JSP* тэги начинаются и заканчиваются угловой скобкой, также как и *HTML* коды, но теги также включают знаки процента, так что все *JSP* теги обозначаются так:

<% *JSP code here* %>.

За лидирующим знаком процента могут следовать другие символы, которые определяют точный тип *JSP* кода в тэге.

Вот чрезвычайно простой *JSP* пример, который использует стандартный библиотечный *Java* вызов для получения текущего времени в миллисекундах, которое затем делится на 1000, чтобы получить время в секундах. Так как используется *JSP* выражение (<%=), результат вычислений преобразуется в строку, а затем поместится в сгенерированную *Web* страницу:

```
//:! c15:jsp:ShowSeconds.jsp
<html><body>
<H1>The time in seconds is:
<%= System.currentTimeMillis()/1000 %></H1>
</body></html>
///:~
```

В *JSP* примерах этой книги восклицательный знак в первой "строке комментариев" означает, что первая и последняя строки не будут включаться в реальный файл кода, который помещен в дерево исходного кода этой книги.

Когда клиент создает запрос к *JSP* странице, *Web* сервер должен быть сконфигурирован, чтобы переправить запрос к *JSP* контейнеру, который затем вовлекает страницу. Как упомянуто ранее, при первом обращении запрашивается страница, генерируются компоненты, указанные на странице, и *JSP* контейнером компилируется один или несколько сервлетов. В приведенном выше примере сервлет будет содержать код конфигурации объекта *HTTPServletResponse*, производя объект *PrintWriter*'а (который всегда называется *out*), а затем включится расчет времени с помощью очень краткой инструкции, но среднестатистический *HTML* программист/*Web* дизайнер не имеет опыта в написании такого кода.

Неявные объекты

Сервлеты включают классы, которые предоставляют удобные утилиты, такие как *HttpServletRequest*, *HttpServletResponse*, *Session* и т. п. Объекты этих классов встроены в *JSP* спецификацию и автоматически доступны для использования в вашем *JSP* без написания дополнительных строк кода. Неявные объекты *JSP* детально перечислены в приведенной ниже таблице.

Область видимости каждого объекта может значительно варьироваться. Например, объект сессии имеет область видимости, которая превышает страницу, так как она может распространяться на несколько клиентских запросов и страниц. Объект приложения может предоставить обслуживание группе *JSP* страниц, которые вместе представляют *Web* приложение.

JSP директивы

Директивы являются сообщениями *JSP* контейнеру и обозначаются символом "@":

<%@ *directive* {*attr*="*value*"}* %>

Директивы ничего не посылают в выходной поток, но они важны для настройки атрибутов вашей *JSP* страницы и зависимостей *JSP* контейнера. Например, строка:

<%@ *page language*="*java*" %>

сообщает, что скриптовой язык, используемый внутри *JSP* страницы, это *Java*. Фактически, *JSP* спецификация только описывает семантику скриптов для языковых атрибутов, эквивалентных "*Java*". Смысл этой директивы состоит во встраивании гибкости в *JSP* технологию. В будущем, если вы выберите другой язык, скажем *Python* (хороший выбор для скриптов), то такой язык должен иметь поддержку *Java Run-time Environment*, выставляя наружу объектную модель *Java* технологии для скриптового окружения, особенно для неявных переменных, определенных выше, свойств *JavaBeans* и публичных методов.

Наиболее важными директивами являются директивы страницы. Они определяют несколько атрибутов страницы и взаимодействие этих атрибутов с *JSP* контейнером. Эти атрибуты включают: *language*, *extends*, *import*, *session*, *buffer*, *autoFlush*, *isThreadSafe*, *info* и *errorPage*. Например:

<%@ *page session*=”*true*” *import=&dqut*;*java.util*.*&dqut*; %>

Эта строка, прежде всего, указывает, что эта страница требует участие в *HTTP* сессии. Так как мы не установили директиву языка, *JSP* контейнер по умолчанию использует *Java* и неявную переменную скриптового языка с названием *session* типа *javax.servlet.http.HttpSession*. Если бы директива использовала *false*, то неявная переменная *session* будет недоступна. Если переменная *session* не определена, то по умолчанию считается "*true*".

Атрибут *import* описывает типы, которые доступны для скриптовой среды. Этот атрибут используется так же, как и в языке программирования *Java*, т. е., разделенный запятыми обычный список выражений *import*. Этот список импортируется транслируемой реализацией *JSP* страницы и доступен для скриптового окружения. Скажем еще раз, что в настоящее время это определено, только если значением директивы языка является "*java*".

Скриптовые элементы *JSP*

Как только вы использовали директивы для установки скриптового окружения, вы можете использовать скриптовые элементы. *JSP* 1.1 имеет три скриптовых языковых элемента - декларацию, скриплет и выражение. Декларация декларирует элементы, скриплеты являются фрагментами инструкций, а выражения являются полным языковым выражением. В *JSP* каждый скриптовый элемент начинается с "<%". Синтаксис каждого из них:

<%! *declaration* %>

<% *scriptlet* %>
<%= *expression* %>

Пробелы после "<%!", "<%", "<%=" и перед "%>" не обязательны.

Все эти теги основываются на *XML*; вы даже можете сказать, что *JSP* страница может быть отражена на *XML* документ. Эквивалентный синтаксис для скриптовых элементов, приведенных выше, может быть:

<*jsp*:*declaration*> *declaration* </*jsp*:*declaration*>
<*jsp*:*scriptlet*> *scriptlet* </*jsp*:*scriptlet*>
<*jsp*:*expression*> *expression* </*jsp*:*expression*>

Кроме тоге, есть два типа комментариев:

<%-- *jsp comment* --%>
<!-- *html comment* -->

Первая форма позволяет вам добавлять комментарии в исходный код *JSP*, которые ни в какой форме не появятся в *HTML* странице, посылаемой клиенту. Конечно, вторая форма комментариев не специфична для *JSP* - это обычный *HTML* комментарий. Интересно то, что вы можете вставлять *JSP* код внутрь *HTML* комментария и результат будет показан в результирующей странице.

Декларации используются для объявления переменных и методов в скриптовом языке (в настоящее время только в *Java*), используемых на *JSP* странице. Декларация должна быть законченным *Java* выражением и не может производить никакого вывода в выходной поток. В приведенном ниже примере *Hello.jsp* декларации для переменных *loadTime*, *loadDate* и *hitCount* являются законченными *Java* выражениями, которые объявляют и инициализируют новые переменные:

```
//:! c15:jsp:Hello.jsp
<%-- Этот JSP комментарий не появится в
сгенерированном html --%>
<%-- Это JSP директива: --%>
<%@ page import="java.util.*" %>
<%-- Эта декларации: --%>
<%!
  long loadTime= System.currentTimeMillis();
  Date loadDate = new Date();
  int hitCount = 0;
```

```
%>
<html><body>
<%-- Следующие несколько строк являются результатом
JSP выражений, вставленных в сгенерированный html;
знак '=' указывает на JSP выражение --%>
<H1>Эта страница была загружена <%= loadDate %> </H1>
<H1>Hello, world! It's <%= new Date() %></H1>
<H2>Here's an object: <%= new Object() %></H2>
<H2>This page has been up
<%= (System.currentTimeMillis()-loadTime)/1000 %>
seconds</H2>
<H3>Page has been accessed <%= ++hitCount %>
times since <%= loadDate %></H3>
<%-- "Скриплет", которые пишет на консоли сервера
и на странице клиента.
Обратите, что необходимо ставить ';': --%>
<%
  System.out.println("Goodbye");
  out.println("Cheerio");
%>
</body></html>
///:~
```

Когда вы запустите эту программу, вы увидите, что переменные *loadTime*, *loadDate* и *hitCount* содержат свои значения между обращениями к странице, так что они явно являются полями, а не локальными переменными.

В конце примера помещен скриплет, который пишет "*Goodbye*" на консоль *Web* сервера и "*Cheerio*" в неявный объект вывода *JspWriter*. Скриплет может содержать любые фрагменты кода, которые являются имеющими силу инструкциями *Java*. Скриплеты выполняются во время обработки запроса. Когда все фрагменты скриплета в данном *JSP* будут скомбинированы по порядку своего появления в *JSP* странице, они должны дать имеющую силу инструкцию, определенную для языка программирования *Java*. Будет ли скриплет производить вывод в выходной поток или нет, зависит только от кода скриплета. Вы должны знать, что скриплет может воздействовать на объекты, которые видимы для него.

JSP выражения можно найти вперемешку с *HTML* в среднем разделе *Hello.jsp*. Выражения должны быть законченными *Java* инструкциями, которые вычисляются, переводятся в строку и посылаются в вывод. Если результат инструкции не может быть переведен в строку (*String*), будет выброшено исключение *ClassCastException*.

Извлечение полей и значений

Следующий пример похож на приведенный ранее в разделе о сервлетах. При первом показе страницы она определяет, что у вас нет полей и возвращает страницу, содержащую форму с помощью того же самого кода, что и в примере с сервлетом, но в формате *JSP*. Когда вы отправляете форму с заполненными полями по тому же самому *JSP URL*'у, страница обнаруживает поля и отображает их. Это прелестная техника, поскольку она позволяет вам получить две страницы, одна из которых содержит форму для заполнения пользователем, а вторая содержит код ответа на эту страницу, в едином файле, таким образом, облегчается создание и поддержка:

```
//:! c15:jsp:DisplayFormData.jsp
<%-- Извлечение данных из HTML формы. --%>
<%-- Эта JSP также генерирует форму. --%>
<%@ page import="java.util.*" %>
<html><body>
<H1>DisplayFormData</H1><H3>
<%
  Enumeration flds = request.getParameterNames();
  if(!flds.hasMoreElements()) { // Нет полей %>
    <form method="POST"
    action="DisplayFormData.jsp">
<% for(int i = 0; i < 10; i++) {  %>
      Field<%=i%>: <input type="text" size="20"
      name="Field<%=i%>" value="Value<%=i%>"><br>
<% } %>
    <INPUT TYPE=submit name=submit
    value="Submit"></form>
<%} else {
    while(flds.hasMoreElements()) {
```

```
    String field = (String)flds.nextElement();
    String value = request.getParameter(field);
%>
    <li><%= field %> = <%= value %></li>
<%  }
  } %>
</H3></body></html>
///:~
```

Более интересное свойство этого примера заключается в том, что он демонстрирует, как код скриплета может быть перемешан с *HTML* кодом даже в том месте, где генерируется *HTML* внутри *Java* цикла. Это особенно удобно для построения форм любого рода, в противном случае необходимо было бы вставлять повторяющийся *HTML* код.

Атрибуты *JSP* страницы и область видимости

Просматривая документацию в *HTML* для сервлетов и *JSP*, вы найдете возможность, которая сообщает информацию о текущем запущенном сервлете или *JSP*. Следующий пример отображает некоторую часть этих данных:

```
//:! c15:jsp:PageContext.jsp
<%-- Просмотр атрибутов pageContext--%>
<%-- Обратите внимание, что вы можете включить любое количество кода
внутри тэгов скриплета --%>
<%@ page import="java.util.*" %>
<html><body>
Servlet Name: <%= config.getServletName() %><br>
Servlet container supports servlet version:
<% out.print(application.getMajorVersion() + "."
+ application.getMinorVersion()); %><br>
<%
  session.setAttribute("My dog", "Ralph");
  for(int scope = 1; scope <= 4; scope++) {  %>
    <H3>Scope: <%= scope %> </H3>
<%  Enumeration e =
```

```
    pageContext.getAttributeNamesInScope(scope);
   while(e.hasMoreElements()) {
    out.println("\t<li>" +
     e.nextElement() + "</li>");
   }
  }
%>
</body></html>
///:~
```

Этот пример также показывает использование встроенного *HTML* и записи в *out*, чтобы в результате получить *HTML* страницу.

Первая часть информации производит имя сервлета, которое, вероятнее всего, будет просто "*JSP*", но это зависит от вашей реализации. Вы можете также определить текущую версию контейнера сервлетов, используя объект *application*. И, наконец, после установки атрибута сессии в обычной области видимости отображаются "имена атрибутов". Обычно вы не используете область видимости в большинстве *JSP*; они показаны здесь просто, чтобы добавить интереса в этот пример. Есть три следующие атрибута области видимости: область видимости страницы (*scope* 1), область видимости запроса (*scope* 2), область видимости сессии (*scope* 3 - здесь доступен только один элемент - это "*My dog*", добавленный прямо перед циклом) и область видимости приложения (*scope* 4), основанная на объекте *ServletContext*. Есть один *ServletContext* на каждое "*Web* приложение" в каждой *Java* Машине. ("*Web* приложение" - это набор сервлетов и содержимого, установленного под определенным подмножеством *URL*'ов Сервера, таких как /*catalog*. Они устанавливаются с помощью конфигурационного файла.) В области видимости приложения вы увидите объекты, которые представляют пути для рабочего каталога и временного каталога.

Манипуляция сессиями в *JSP*

Сессии были введены в предыдущем разделе, посвященном сервлетам и также доступны в *JSP*. Следующие примеры исследуют сессионные объекты и позволяют вам манипулировать промежутком времени, прежде, чем сессия станет недействительной:

```
//:! c15:jsp:SessionObject.jsp
```

```
<%-- Получение и установка значений объекта сессии --%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H3><li>This session was created at
<%= session.getCreationTime() %></li></H1>
<H3><li>Old MaxInactiveInterval =
  <%= session.getMaxInactiveInterval() %></li>
<% session.setMaxInactiveInterval(5); %>
<li>New MaxInactiveInterval=
  <%= session.getMaxInactiveInterval() %></li>
</H3>
<H2>If the session object "My dog" is
still around, this value will be non-null:<H2>
<H3><li>Session value for "My dog" =
<%= session.getAttribute("My dog") %></li></H3>
<%-- Теперь добавим объект сессии "My dog" --%>
<% session.setAttribute("My dog",
                new String("Ralph")); %>
<H1>My dog's name is
<%= session.getAttribute("My dog") %></H1>
<%-- See if "My dog" wanders to another form --%>
<FORM TYPE=POST ACTION=SessionObject2.jsp>
<INPUT TYPE=submit name=submit
Value="Invalidate"></FORM>
<FORM TYPE=POST ACTION=SessionObject3.jsp>
<INPUT TYPE=submit name=submit
Value="Keep Around"></FORM>
</body></html>
///:~
```

Объект сессии предоставляется по умолчанию, так что он доступен без дополнительного кода. Вызов *getID*( ), *getCreationTime*( ) и *getMaxInactiveInterval*( ) используются для отображения информации об этом сессионном объекте.

Когда вы в первый раз получите эту сессию, вы увидите, что *MaxInactiveInterval* составляет, например, 1800 секунд (30 минут). Это зависит от конфигурации вашего контейнера *JSP*/сервлетов. *MaxInactiveInterval* сокращается до 5 секунд, чтобы сделать вещи

интереснее. Если вы перегрузите страницу до того, как истекут 5 секунд, то вы увидите:

Session value for "*My dog*" = *Ralph*

Но если вы подождете больший интервал, то "*Ralph*" станет *null*.

Чтобы посмотреть, как сессионная информация переносится на другие страницы, а также, чтобы посмотреть эффект становления объекта сессии недействительным по сравнению с простым вариантом, когда вы дали ему устареть, созданы две новые страницы. Первая (доступна при нажатии кнопки "*invalidate*" в *SessionObject.jsp*) читает сессионную информацию, а затем явно делает сессию недействительной:

```
//:! c15:jsp:SessionObject2.jsp
<%-- Объект сессии переноситься --%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<% session.invalidate(); %>
</body></html>
///:~
```

Чтобы экспериментировать с этим, обновите *SessionObject.jsp*, затем сразу же кликните на кнопку "*invalidate*", чтобы перейти к странице *SessionObject2.jsp*. В этом месте вы все еще будете видеть "*Ralph*" и сразу после этого (прежде, чем пойдет интервал в 5 секунд), обновите *SessionObject2.jsp*, чтобы увидеть, что сессия была успешно сделана недействительной и "*Ralph*" исчез.

Если вы вернетесь на *SessionObject.jsp*, обновите страницу так, чтобы вы снова имели 5-ти секундный интервал, затем нажмете кнопку "*Keep Around*", то вы попадете на следующую страницу *SessionObject3.jsp*, которая не делает сессию недействительной:

```
//:! c15:jsp:SessionObject3.jsp
<%-- Переход объекта сессии по страницам --%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
```

```
<FORM TYPE=POST ACTION=SessionObject.jsp>
<INPUT TYPE=submit name=submit Value="Return">
</FORM>
</body></html>
///:~
```

Задание на лабораторную работу

1. Создать *Web*-приложение на *Java* для работы с использованием технологии *JSP*-страниц.
2. Приложение должно обеспечивать ввод, редактирование, удаление данных, осуществлять выборки в соответствии с вариантом задания
3. Разработанное приложение должно использовать *Web*-интерфейс пользователя для осуществления всех действий
4. Данные хранить в БД, обращаться с использованием технологии *JDBC*.
5. Варианты заданий приведены в таблице 3.

Таблица 3

Варианты базы данных

Вариант	Описание
1	Цех, участок, Ф.И.О. Объем выполненной работы
2	УДК, Ф.И.О. автора, Наименование, Количество
3	Номер поезда, Наименование, Место отправления, Место прибытия, Категория (скорый.)
4	Организация, Ф.И.О., Год рождения, Занятое место
5	Страна, Площадь, Население, Континент, Столица
6	Марка автомобиля, Номер, Цвет, Год выпуска, Пробег
7	Порода собаки, Кличка, Отец, Мать, Дата рождения, ФИО хозяина
8	Вид овощей, Название сорта, Дата посадки, Дата уборки, Урожай
9	Дисциплина, Объем лекций, Объем лабораторных, Вид контроля, Группа
10	Дата, Температура, Давление, Облачность, Направление ветра
11	Наименование, Фирма, Стоимость, Количество, Дата

Окончание табл. 3

Вариант	Описание
12	Название группы, Страна, Альбом, Дата выпуска, Число продаж
13	Название вершины, Высота, Страна, Год покорения, Количество восхождений
14	Ф.И.О., Год рождения, Рост, Вес, Группа крови
15	Ф.И.О., Область деятельности, Год рождения, Страна, Число публикаций

Контрольные вопросы

1. Что такое сервлет и что такое *JSP* ? Их сходство и различия.
2. Нарисуйте функциональную схему работы веб-приложения на основе *Java*-сервлетов и *JSP*.
3. Какие методы работают в течение жизненного цикла сервлета?
4. С помощью каких классов или интерфейсов чаще всего создают сервлет ? Напишите код.
5. Что такое дескриптор развертывания, для чего он нужен?
6. Из каких папок состоит веб-приложение, что в этих папках находится?
7. Какие преимущества и какие недостатки у сервлетов ?
8. Для чего нужно межсервлетное взаимодействие, какими механизмами его можно обеспечить (назовите классы, методы) ?
9. Приведите механизмы отслеживания сессии с использованием *JSP*- страниц.
10. Преимущества *JSP*-страниц при реализации в многопоточном окружении.

Лабораторная работа № 6 «Использование шаблонов проектирования для разработки приложений»

Цель работы: научиться использовать шаблоны проектирования для разработки приложений

Теоретические сведения

Моделирование данных является важнейшим процессом при проектировании программного обеспечения (ПО). По этой причине, разработчики *CASE*-средств в своих продуктах вынуждены уделять моделированию данных повышенное внимание. Являясь признанным лидером в области объектных методологий, фирма *Rational Software Corporation*, тем не менее, до недавнего времени такого средства не имела. Основной причиной этого, по-видимому, является ориентация на язык *Unified Modeling Language* (*UML*), как универсальный инструмент моделирования. *UML* полностью покрывает потребности моделирования данных. Сложившаяся на протяжении десятилетий технология моделирования данных, традиции, система понятий и колоссальный опыт разработчиков не могли далее игнорироваться. Немаловажную роль здесь сыграла и необходимость формального контроля моделей данных, что является абсолютно необходимым при проектировании мало-мальски больших схем баз данных и что *UML* не обеспечивает в достаточной степени. И, наконец, последней причиной, побудившей специалистов *Rational Software Corporation* к созданию собственного средства моделирования данных, является требование построения эффективных физических моделей, прежде всего для конкретных СУБД - лидеров рынка.

Шаблон проектирования *MVC* (*Model-View-Controller*)

При использовании шаблона *MVC* поток выполнения приложения всегда обязан проходить через контроллер приложения. Контроллер направляет запросы – в данном случае *HTTP*(*S*)-запросы – к соответствующему обработчику. Обработчики запроса связаны с бизнес-моделью, и в итоге каждый разработчик приложения должен только обеспечить взаимодействие между запросом и бизнес-моделью. В результате реакции системы на запрос вызывается соответствующая *JSP*-страница, выполняющая в данной схеме роль представления.

В результате модель отделена от представления, и все связывающие их команды проходят через контроллер приложения. Приложение, соответствующее этим принципам, становится более понятным с точки зрения разработки, поддержки и

совершенствования, а отдельные его части довольно легко могут быть использованы повторно.

В оригинальной концепции была описана сама идея и роль каждого из элементов: модели, представления и контроллера. Но связи между ними были описаны без конкретизации. Кроме того, различали две основные модификации:

– Пассивная модель – модель не имеет никаких способов воздействовать на представление или контроллер, и используется ими в качестве источника данных для отображения. Все изменения модели отслеживаются контроллером и он же отвечает за перерисовку представления, если это необходимо. Такая модель чаще используется в структурном программировании, так как в этом случае модель представляет просто структуру данных, без методов их обрабатывающих.

– Активная модель – модель оповещает представление о том, что в ней произошли изменения, а представления, которые заинтересованы в оповещении, подписываются на эти сообщения. Это позволяет сохранить независимость модели как от контроллера, так и от представления.

Классической реализацией концепции *MVC* принято считать версию именно с активной моделью.

С развитием объектно-ориентированного программирования и понятия о шаблонах проектирования был создан ряд модификаций концепции *MVC*, которые при реализации у разных авторов могут отличаться от оригинальной. Так, например, Эриан Верми в 2004 году описал пример обобщенного *MVC*.

Концепция *MVC* позволяет разделить данные, представление и обработку действий пользователя на три отдельных компонента:

Модель (англ. *Model*). Модель предоставляет знания: данные и методы работы с этими данными, реагирует на запросы, изменяя своё состояние. Не содержит информации, как эти знания можно визуализировать.

Представление, вид (англ. *View*). Отвечает за отображение информации (визуализацию). Часто в качестве представления выступает форма (окно) с графическими элементами.

Контроллер (англ. *Controller*). Обеспечивает связь между пользователем и системой: контролирует ввод данных пользователем и использует модель и представление для реализации необходимой реакции.

Важно отметить, что как представление, так и контроллер зависят от модели. Однако модель не зависит ни от представления, ни от контроллера. Тем самым достигается назначение такого разделения: оно позволяет строить модель независимо от визуального представления, а также создавать несколько различных представлений для одной модели.

Для реализации схемы *Model-View-Controller* используется достаточно большое число шаблонов проектирования (в зависимости от сложности архитектурного решения), основные из которых «наблюдатель», «стратегия», «компоновщик».

Наиболее типичная реализация отделяет вид от модели путем установления между ними протокола взаимодействия, используя аппарат событий (подписка/оповещение). При каждом изменении внутренних данных в модели она оповещает все зависящие от неё представления, и представление обновляется. Для этого используется шаблон «наблюдатель». При обработке реакции пользователя вид выбирает, в зависимости от нужной реакции, нужный контроллер, который обеспечит ту или иную связь с моделью. Для этого используется шаблон «стратегия», или вместо этого может быть модификация с использованием шаблона «команда». А для возможности однотипного обращения с подобъектами сложно-составного иерархического вида может использоваться шаблон «компоновщик». Кроме того, могут использоваться и другие шаблоны проектирования, например, «фабричный метод», который позволит задать по умолчанию тип контроллера для соответствующего вида.

Запрос к серверу обрабатывается классом-контроллером ***ActionServlet*** в соответствии с настройками в файле ***web.xml***. Запрос на сервер выполняет пользователь нажатием кнопки *Submit*, введением *URL* в поле браузера, вызовом *submit*-формы на *JavaScript* и пр. Во время инициализации главный контроллер считывает (*parse*) конфигурационный файл ***struts-config.xml***, который однозначно определяет все соответствия и альтернативы для всех запросов данного приложения и упаковывает их в объект класса ***org.apache.struts.action.ActionMapping*.**

Для запроса контроллер находит соответствующие ему ***ActionForm***- и ***Action***-классы. Первым создается *ActionForm*, если такой объект еще не создавался и не находится ни в одной из областей видимости (сессия, запрос). Далее поля объекта *ActionForm* заполняются данными, которые пришли с запросом, т.е. для каждого

параметра, содержащегося в запросе, вызывается соответствующий *set*-метод. Например, если в запросе есть параметр ***login=goch***, то в ***ActionForm*** будет вызван метод ***setLogin()*** с передачей значения параметра. В *Struts* существует возможность заполнения структуры объектов. Пусть в ***ActionForm*** с помощью методов ***setCompany()*** и ***getCompany()*** передается объект класса *Company*, полем которого является коллекция объектов типа ***Employee***, для каждого из которых определено поле типа ***Address***. Тогда для заполнения поля адреса одного из работников достаточно присутствия параметра ***company.employees[n].address.phone***

Простое приложение

```
/* пример1: Action класс : LoginAction.java */
package jspservlet;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;

public class LoginAction extends Action {
    public ActionForward execute(ActionMapping mapping,
    ActionForm form, HttpServletRequest request,
    HttpServletResponse response)
    throws IOException, ServletException {
        ActionMessages errors = new ActionMessages();

        LoginForm actionForm = (LoginForm)form;
        String login = actionForm.getLogin();
        String password = actionForm.getPassword();
    if (login != null && password != null) {
        if (LoginLogic.checkLogin(login, password)) {
            return mapping.findForward("success");
        } else {
```

```
            errors.add(ActionMessages.GLOBAL_MESSAGE,
            new
        ActionMessage("error.login.incorrectLoginOrPassword"));
                        saveErrors(request, errors);
                                    }
            }
          //загрузка формы для логина
            return mapping.findForward("loginAgain");
        }
    }
```

Класс *LoginForm*, объект которого представляет форму, соответствующую странице ввода логина и пароля, выглядит следующим образом:

```
    /* пример : класс хранения информации, передаваемой из
login.jsp :
    LoginForm.java */
    package jspservlet;
    import javax.servlet.http.HttpServletRequest;
    import org.apache.struts.action.ActionForm;
    import org.apache.struts.action.ActionMapping;

    public class LoginForm extends ActionForm {
        private String login;
        private String password;
        //очищение полей формы
        public void reset(ActionMapping mapping,
                    HttpServletRequest request) {
            super.reset(mapping, request);
            this.password = null;
        }
        public String getLogin() {
            return login;
        }
        public String getPassword() {
            return password;
        }
        public void setLogin(String login) {
```

```
            this.login = login;
        }
        public void setPassword(String password) {
            this.password = password;
        }
    }
```

Для данного приложения файл ресурсов может быть создан в виде:

```
    # пример : файл ресурсов :  ApplicationResources.properties
    # header и footer, которые будут использоваться для
    # обрамления ошибок, выдаваемых тегом <errors/>.
    errors.header=<ul>
    errors.footer=</ul>
    errors.prefix=<li>
    errors.suffix=</li>
    # разметка для элемента списка ошибок из <errors/>,
    # который указывает, что логин или пароль неверны.
    error.login.incorrectLoginOrPassword=<li>incorrect login or
password</li>

    # текстовая информация на login.jsp
    jsp.login.title=Login
    jsp.login.header=Login
    jsp.login.field.login=Login
    jsp.login.field.password=Password
    jsp.login.button.submit=Enter

    # текстовая информация на main.jsp
    jsp.main.title=Welcome
    jsp.main.header=Welcome
    jsp.main.hello=Hello

    # текстовая информация на error.jsp
    jsp.error.title=Error
    jsp.error.header=Error
    jsp.error.returnToLogin=Return to login page
```

Имя и месторасположение этого файла настраиваются в *struts-config.xml*, который является основным конфигурационным файлом для приложения, построенного на основе *Struts*.

пример : конфигурация action, forward, resource и т.д. : struts-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache
Software Foundation//DTD Struts Configuration 1.2//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
    <data-sources>
<!--источники данных к БД. Как правило, это организация пула соединений. Пример приведен в конце раздела-->
    </data-sources>
<!-- ========== Form Bean Definitions -->
        <form-beans>
    <form-bean  name="loginForm"
    type="jspservlet.LoginForm" />
  </form-beans>

<!-- ============== Global Exception Definitions (если есть)-->
    <global-exceptions>
        </global-exceptions>
<!-- ============== Global Forward Definitions(если есть) -->
        <global-forwards>
            </global-forwards>

<!-- ================ Action Mapping Definitions -->
        <action-mappings>
            <!-- Action для процесса логина -->
<action name="loginForm"
          path="/login"
          scope="request"
<!-- Задается область видимости формы. Часто необходимо, чтобы форма лежала в сессии, а не в запросе. -->
          type="jspservlet.LoginAction"
          validate="false">
```

Получить данный пул из класса, реализующего *org.apache.struts.action.Action*, можно с помощью метода *getDataSource*(*HttpServletRequest request*)).

Оставшиеся *JSP*-страницы были изменены несущественно:

```
<%-- пример : страница, вызываемая после прохождения идентификации : main.jsp --%>
<%@ page errorPage="error.jsp" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<html:html locale="true">
<head>
    <title><bean:message key="jsp.main.title"/></title>
    <html:base/>
</head>
<body>
<h3><bean:message key="jsp.main.header"/></h3>
<hr/>
<bean:message key="jsp.main.hello"/>,
    <bean:write name="loginForm" property="login"/>
<hr/>
<a href="login.do">
<bean:message
                key="jsp.error.returnToLogin"/></a>
<html:errors/>
</body>
</html:html>
```

Задание на лабораторную работу

1. Создать *Web*-приложение на *Java* для работы с использованием шаблонов проектирования (*MVC*).
2. Приложение должно обеспечивать ввод, редактирование, удаление данных, осуществлять выборки в соответствии с вариантом задания.
3. Разработанное приложение должно использовать *Web*-интерфейс пользователя для осуществления всех действий
4. Данные хранить в БД, обращаться с использованием технологии *JDBC*.

Варианты заданий:

1. ***Student***: *id*, Фамилия, Имя, Отчество, Дата рождения, Адрес, Телефон,

Создать массив объектов. Вывести:

a) список студентов заданного факультета;

b) списки студентов для каждого факультета и курса;

c) список студентов, родившихся после заданного года;

d) список учебной группы.

2. ***Customer***: *id*, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Номер банковского счета.

Создать массив объектов. Вывести:

a) список покупателей в алфавитном порядке;

b) список покупателей, у которых номер кредитной карточки находится в заданном интервале.

3. ***Patient***: *id*, Фамилия, Имя, Отчество, Адрес, Телефон, Номер медицинской карты, Диагноз.

Создать массив объектов. Вывести:

a) список пациентов, имеющих данный диагноз;

b) список пациентов, номер медицинской карты у которых находится в заданном интервале.

4. ***Abiturient***: *id*, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки.

Создать массив объектов. Вывести:

a) список абитуриентов, имеющих неудовлетворительные оценки;

b) список абитуриентов, средний балл у которых выше заданного;

c) выбрать заданное число *n* абитуриентов, имеющих самый высокий средний балл (вывести также полный список абитуриентов, имеющих полупроходной балл).

5. ***Book***: *id*, Название, Автор(ы), Издательство, Год издания, Количество страниц, Цена, Переплет.

Создать массив объектов. Вывести:

a) список книг заданного автора;

b) список книг, выпущенных заданным издательством;

c) список книг, выпущенных после заданного года.

6. ***House***: *id*, Номер квартиры, Площадь, Этаж, Количество комнат, Улица, Тип здания, Срок эксплуатации.

Создать массив объектов. Вывести:

a) список квартир, имеющих заданное число комнат;

b) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;

c) список квартир, имеющих площадь, превосходящую заданную.

7. ***Phone***: *id*, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Дебет, Кредит, Время городских и междугородных разговоров.

Создать массив объектов. Вывести:

a) сведения об абонентах, у которых время внутригородских разговоров превышает заданное;

b) сведения об абонентах, которые пользовались междугородной связью;

c) сведения об абонентах в алфавитном порядке.

8. ***Car***: *id*, Марка, Модель, Год выпуска, Цвет, Цена, Регистрационный номер.

Создать массив объектов. Вывести:

a) список автомобилей заданной марки;

b) список автомобилей заданной модели, которые эксплуатируются больше *n* лет;

c) список автомобилей заданного года выпуска, цена которых больше указанной.

9. ***Product***: *id*, Наименование, *UPC*, Производитель, Цена, Срок хранения, Количество.

Создать массив объектов. Вывести:

a) список товаров для заданного наименования;

b) список товаров для заданного наименования, цена которых не превосходит заданную;

c) список товаров, срок хранения которых больше заданного.

10. ***Train***: Пункт назначения, Номер поезда, Время отправления, Число мест (общих, купе, плацкарт, люкс).

Создать массив объектов. Вывести:

a) список поездов, следующих до заданного пункта назначения;

b) список поездов, следующих до заданного пункта назначения и отправляющихся после заданного часа;

c) список поездов, отправляющихся до заданного пункта назначения и имеющих общие места.

11. ***Bus***: Фамилия и инициалы водителя, Номер автобуса, Номер маршрута, Марка, Год начала эксплуатации, Пробег.

Создать массив объектов. Вывести:

a) список автобусов для заданного номера маршрута;

b) список автобусов, которые эксплуатируются больше 10 лет;

c) список автобусов, пробег у которых больше 100000 км.

12. ***Airlines***: Пункт назначения, Номер рейса, Тип самолета, Время вылета, Дни недели.

Создать массив объектов. Вывести:

a) список рейсов для заданного пункта назначения;

b) список рейсов для заданного дня недели;

c) список рейсов для заданного дня недели, время вылета для которых больше заданного.

13. ***Abiturient***: *id*, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки.

Создать массив объектов. Вывести:

a) список абитуриентов, имеющих неудовлетворительные оценки;

b) список абитуриентов, средний балл у которых выше заданного;

c) выбрать заданное число *n* абитуриентов, имеющих самый высокий средний балл (вывести также полный список абитуриентов, имеющих полупроходной балл).

14. ***Book***: *id*, Название, Автор(ы), Издательство, Год издания, Количество страниц, Цена, Переплет.

Создать массив объектов. Вывести:

a) список книг заданного автора;

b) список книг, выпущенных заданным издательством;

c) список книг, выпущенных после заданного года.

15. ***House***: *id*, Номер квартиры, Площадь, Этаж, Количество комнат, Улица,Тип здания, Срок эксплуатации.

Создать массив объектов. Вывести:

a) список квартир, имеющих заданное число комнат;

b) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;

c) список квартир, имеющих площадь, превосходящую заданную.

Контрольные вопросы

1. Каким образом строятся модели программных систем ?
2. Необходимость проектирования.
3. Разбиение системы на компоненты.
4. Какие уровни сложности системы вы знаете ?
5. Насколько эффективно использование *Java* для проектирования программных систем ?
6. Преимущества *Java* для проектирования программных систем ?
7. Что такое *UML* ?
8. Связь модели классов и диаграмм классов
9. Модель *MVC*. Преимущества и недостатки.
10. Шаблоны проектирования программных систем.

Лабораторная работа № 7 «Использование *RMI* при разработке распределенных систем»

Цель работы: научиться использовать *RMI* при разработке распределенных систем

Теоретические сведения

Распределенное приложение, построенное с помощью *Java RMI*, создано из интерфейсов и классов. Интерфейсы определяют методы, а

классы реализуют методы, определенные в интерфейсах, а также, возможно, определяют и новые дополнительные методы. В распределенном приложении предусматривается размещение некоторых реализаций на различных виртуальных машинах. Объекты, имеющие методы, которые могут вызываться между различными виртуальными машинами, являются удаленными объектами – *remote objects*.

Объект становится удаленным объектом при реализации удаленного интерфейса – *remote interface*, который имеет следующие характеристики:

1. Удаленный интерфейс расширяет интерфейс *java.rmi.Remote*;

2. Каждый метод интерфейса объявляет *java.rmi.RemoteException* в своем условии *throws*, помимо любых других исключений, специфичных для конкретного приложения.

При передаче объекта от одной виртуальной машины к другой *RMI* рассматривает удаленный объект – *remote object* не так, как обычный не удаленный объект. Вместо того, чтобы делать копию реализации объекта в принимающей виртуальной машине, *RMI* передает удаленный *stub* – заглушку для удаленного объекта. Этот *stub* действет как локальное представление, или заместитель (*proxy*) для удаленного объекта, а для вызывающего является удаленной ссылкой. Вызывающий вызывает метод для локального *stub*, который отвечает за обработку вызова метода удаленного объекта.

Переданный *stub* для удаленного объекта реализует тот же набор удаленных интерфейсов, который реализует удаленный объект. Это позволяет для *stub* быть "передатчиком" к любому из интерфейсов, которые реализует удаленный объект. Однако это также означает, что только те методы, которые определены в удаленном интерфейсе, могут быть вызваны на принимающей виртуальной машине.

Рассмотрим создание простейшего приложения на примере *Hello World*. Сервер предоставляет один метод, принимающий в качестве параметра имя и возвращающий строку «*Hello* имя».

1. Создаем интерфейс *IHelloWorld*

```
package chstu.ds.lab1;
import java.rmi.Remote;
import java.rmi.RemoteException;
//интерфейс должен расширять Remote
```

```
public interface IHelloWorld extends Remote {
        //      и генерировать RemoteException
public String SyaHello(String name) throws RemoteException;
}
```

2. Создаем его реализацию:

```
package chstu.ds.lab1;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class HelloImpl extends UnicastRemoteObject implements
IHelloWorld {
        /* UnicastRemoteObject предоставляет такую реализацию
        многих
         * методов класса java.lang.Object (equals, hashCode,
         * toString), подходящие для удаленных объектов */
        public HelloImpl() throws RemoteException{
                super();
        }

        public String SyaHello(String name) throws RemoteException{
                return "Hello "+name;
        }
}
```

3. Создаем сервер

```
package chstu.ds.lab1;
import java.rmi.*;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class HelloServer {
        public static void main(String[] args) {
        // задаем файл политики безопасности
        System.setProperty("java.security.policy","D:\\rmi.policy");
        // Указываем сервер на котором регистрироваться и имя
сервиса
        String name = "rmi://localhost/HelloService";
        if (System.getSecurityManager() == null) {
```

```
        System.setSecurityManager(new RMISecurityManager());
       }
        try {
        // создаем RMI Registry. Если его не создавать,
        // то требуется запускать отдельно. RMI Registry является
        // простым сервисом именований и позволяет клиенту
        // получить ссылку на удаленный объект по имени

LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
        IHelloWorld hw = new HelloImpl();
        // регистрируем объект в RMI Registry
        Naming.rebind(name, hw);
          System.out.println("ComputeEngine bound");
       } catch (Exception e) {
          System.err.println("ComputeEngine exception: " +
                        e.getMessage());
          e.printStackTrace();
       }
        }}
```

4. Создаем клиентский класс

```
package chstu.ds.lab1;
import java.rmi.*;
public class HelloClient {
public static void main(String[] args) {
              try {
          // адрес RMI Registry и имя сервиса
              String name = "rmi://localhost/HelloService";
          // получаем клиентскую заглушку
              IHelloWorld hw = (IHelloWorld) Naming.lookup(name);
              // вызываем метод удаленного объекта
          System.out.println(hw.SyaHello("Vasya"));
       } catch (Exception e) {
          System.err.println("ComputePi exception: " +
                        e.getMessage());
          e.printStackTrace();
       }
        }
```

}

5. Запускаем сервер
6. Запускаем клиент.

Задание на лабораторную работу

Разработать серверное приложение с использованием технологии *RMI* на языке программирования *java*. Сервер выполняет действия в соответствии с таблицей. Клиент подключается к серверу, передает необходимые данные и получает результат**.** Варианты заданий приведены в таблице 4.

Таблица 4

Варианты заданий

№	Приложение	Функции сервера
1	Целочисленный калькулятор	Операции: сложение, вычитание, умножение, целочисленное деление, НОД (наибольший общий делитель)
2	Калькулятор чисел с плавающей точкой	Операции: сложение, вычитание, умножение, деление квадратный корень $\sqrt{x}$
3	Математический калькулятор №1	Операции: $sin\ x$, $cos\ x$, e^x, $log_{10}\ x$, $log_2 x$
4	Математический калькулятор №2	Операции: $sin\ x$, $cos\ x$, $tg\ x$, $\sqrt[y]{x}$, x^y,
5	Редактор строки	Операции: сложение, замена одной подстроки другой, замена строчных букв прописными (в т.ч. для русских букв)
6	Редактор строки	Операции: «переворот» текста, перевод в транслитерацию, замена прописных букв строчными (в т.ч. для русских букв)

Продолжение табл. 4

№	Приложение	Функции сервера
7	Работа с комплексными числами	Операции: сложение, вычитание, умножение
8	Восьмеричный калькулятор	Операции: Сложение, вычитание, умножение, остаток от деления
9	Троичный калькулятор	Операции: Сложение, вычитание, умножение, остаток от деления
10	Конвертер целых чисел №1	Преобразование **в** системы счисления: 16-тиричную, 8-ричную, 3-ичную, 2-ичную
11	Конвертер целых чисел №2	Преобразование **из** систем счисления: 16-тиричной, 8-ричной, 3-ичной, 2-ичной
12	Перевод числа от 0 до 99 в текстовый вид	Перевод на языки: русский, белорусский, английский
13	Конвертер единиц длины	Перевод длины из метров в: километры, мили, ярды, футы, дюймы
14	Проверка делимости чисел	Проверка делимости целых чисел на: 2, 3, 5, 10
15	Преобразование текста №1	Замена всех: цифр на символ «*», пробелов на символ «_», русских букв на символ «#»
16	Преобразование текста №2	Преобразование текста: каждое новое слово с большой буквы, а остальной текст – маленькими (и для русских букв), каждое новое предложение с большой буквы, а остальной текст – маленькими (и для русских букв).

Окончание табл. 4

17	Преобразование текста №3	Замена в тексте: все гласные буквы становятся большими, все согласные буквы становятся большими, все знаки препинания дублируются (удваиваются).
18	Конвертер единиц объёма	Перевод объёма из литров в: галлоны, кубические футы, кубические дюймы
19	Перевод единиц времени	Перевод отрезка времени, заданного в виде числа суток в: число часов, число минут, число секунд
20	Логические операции	Операции с логическими операндами (Истина, Ложь): «И», «ИЛИ», «Исключающее ИЛИ».

Контрольные вопросы

1. Понятия и архитектура распределенной системы и приложений. Требования к распределенным системам
2. Программная реализация удаленного вызова процедур *RMI*
3. Веб-службы и их использование
4. Протоколы взаимодействия веб-служб
5. Реализация интерфейса *Remote* с использованием *RMI*.
6. Реализация интерфейса *Local* с использованием *RMI*.
7. Реализация интерфейса *RemoteHome* с использованием *RMI*.
8. Реализация интерфейса *LocalHome* с использованием *RMI*.
9. Алгоритм взаимодействия с использованием технологии *RMI*.
10. Недостатки технологии *RMI*.

Литература

1. Дейтел Х.М., Дейтел П.Д. Технология программирования на *Java*2. Книга 1. Графика, *JavaBeans*, интерфейс пользователя. - М.: Бином, 2003. *ISBN*: 5-9518-0017-X.
2. Дейтел Х.М., Дейтел П.Д. Технология программирования на *Java*2. Книга 2. Распределенные приложения. - М.: Бином, 2003. *ISBN*: 5-9518-0051-*X*.
3. Дейтел Х.М., Дейтел П.Д. Технология программирования на *Java*2. Книга 3. Корпоративные системы, сервлеты, *JSP*, *web*-сервисы. - М.: Бином, 2003. *ISBN*: 5-9518-0034-*X*.
4. Дейтел Х.М., Дейтел П.Д. Как программировать на *Java*. Книга 1. Основы программирования. - М.: Бином, 2003. *ISBN*: 5-9518-0015-3.
5. Дейтел Х.М., Дейтел П.Д. Как программировать на *Java*. Книга 2. Файлы, сети, базы данных. - М.: Бином, 2006. *ISBN*: 5-9518-0127-3.
6. Ноутон П., Шилдт Г. *Java*2. Наиболее полное руководство. – СПб.: *BHV*, 2001. *ISBN*: 0-07-211976-4, 5-94157-012-0.
7. Биллиг В.А. Основы программирования на *Java*/ – М.: Изд-во «Интернет-университет информационных технологий – ИНТУИТ.ру», 2006. – 488с.
8. Шилдт Г.С. *Java*: Учебный курс. – СПб.: Питер, 2002. – 512с.
9. Шилдт Г.С. Полный справочник по *Java*. – М.: Издательский дом «Вильямс», 2004. – 752с.

Стефановский Игорь Леонидович
Семенченя Татьяна Сергеевна

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА КОРПОРАТИВНЫХ ИНФОРМАЦИОННЫХ СИСТЕМ НА ОСНОВЕ ТЕХНОЛОГИИ *JEE*

Практикум
по выполнению лабораторных работ
по одноименной дисциплине для студентов
специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)»
дневной и заочной форм обучения

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 31.03.21.

Рег. № 33Е.
http://www.gstu.by