Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерного проектирования

Кафедра инженерной психологии и эргономики

А. А. Быков, Е. А. Мельникова, К. Д. Яшин

ТЕХНОЛОГИИ ВИРТУАЛЬНОЙ РЕАЛЬНОСТИ. ПОСОБИЕ К ПРАКТИЧЕСКИМ ЗАНЯТИЯМ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве пособия для специальности 1-58 01 01 «Инженерно-психологическое обеспечение информационных технологий»

Минск БГУИР 2015

УДК 004.946(076)
ББК 32.973.26-018.2я73
Б95

Р е ц е н з е н т ы:

кафедра робототехнических систем
Белорусского национального технического университета
(протокол №2 от 23.09.2013);

главный научный сотрудник государственного научного учреждения
«Объединенный институт проблем информатики
Национальной академии наук Беларуси»,
доктор технических наук, доцент Л. Д. Черемисинова

Быков, А. А.

Б95 Технологии виртуальной реальности. Пособие к практическим занятиям : пособие / А. А. Быков, Е. А. Мельникова, К. Д. Яшин. – Минск : БГУИР, 2015. – 64 с.

ISBN 978-985-543-059-0.

Содержит краткие теоретические сведения и методические указания к выполнению практических занятий.

Практические задания предназначены для студентов и преподавателей, владеющих теоретическим материалом по темам предлагаемых практических работ.

УДК 004.946(076)
ББК 32.973.26-018.2я73

ISBN 978-985-543-059-0

СОДЕРЖАНИЕ

Введение

Термином «виртуальная реальность» обозначают новое направление развития информационных технологий. Они позволяют создавать у людей иллюзию наблюдения и даже ощущения некоторого искусственного мира, создаваемого при помощи средств информатики и кибернетики. Технологии виртуальной реальности находят в последние годы все более широкое применение во всех направлениях человеческой деятельности.

Одним из способов реализации технологий виртуальной реальности является создание приложений с использованием платформы Java. Язык Java – универсальное средство, обеспечивающее связь пользователей с любыми источниками информации, независимо от того, где она расположена – на web-сервере, в базе данных, хранилище данных и т. д. Этот хорошо разработанный объектно-ориентированный язык программирования имеет встроенные средства, позволяющие решать задачи повышенной сложности, такие как: работа с сетевыми ресурсами, управление базами данных, динамическое наполнение web-страниц, многопоточность приложений.

В пособии представлены задания, нацеленные на выполнение исследовательской работы или решение практических задач. Оно поможет студентам в практическом освоении технологий создания приложений с использованием платформы Java. В пособии предусмотрено двенадцать тем для практических занятий с вариантами заданий для них.

Тема 1. Введение в современное программирование

Цель: исследовать средства и технологии современного программирования.

Теория и примеры выполнения задания

Современное программирование основано на быстром создании каркаса разрабатываемого приложения с использованием средств проектирования и быстрой разработки (RAD), библиотек и каркасов (Frameworks), технологий передачи и обработки данных. Современные программные системы должны соответствовать промышленным стандартам разработки ПО, а получаемый продукт – сложившимся стандартам архитектуры ПО.

Java – объектно-ориентированный язык, легкий в изучении и позволяющий создавать программы, которые могут исполняться на любой платформе без каких-либо доработок (кросс-платформенность). Язык Java похож на упрощенный C или C++ с добавлением garbage collector – автоматического сборщика «мусора» (механизм освобождения памяти, которая больше не используется программой). Java ориентирована на Интернет, и самое ее распространенное применение – небольшие программы, называющиеся сервлеты, которые запускаются в ответ на HTTP-запросы к web-серверу.

Наиболее распространенной технологией создания современных корпоративных приложений является технология Java 2 Platform, Enterprise Edition (J2EE), основанная на платформе Java. J2EE – это не конкретный продукт, а набор спецификаций, устанавливающих правила, которых следует придерживаться поставщикам конкретной реализации платформы J2EE, а также разработчикам корпоративных приложений. Он обладает многими возможностями Java 2 Platform, Standard Edition (J2SE):

- переносимостью («пишем один раз, используем везде»);
- JDBC API для доступа к базам данных;
- технологией CORBA для взаимодействия с существующими ресурсами предприятия;
- обеспечением защиты данных.

Разработанная на этой базе Java 2 Platform, Enterprise Edition поддерживает компоненты Enterprise JavaBeans, Java Servlets API, JavaServer Pages и технологию XML. Для поддержки Enterprise JavaBeansTM, бизнес-компонентов Java Servlets и JavaServer PagesTM спецификация J2EE, устанавливает несколько стандартных служб для использования J2EE-компонентами (рисунок 1).

Рисунок 1 – Спецификация Java 2 Platform

Таким образом, революция Java и открытых API связана с эволюцией в существующих продуктах, таких как базы данных, приложения, почта и web-серверы. Широкая доступность продуктов для запуска серверных Java-приложений сильно увеличивает ее конкурентоспособность на рынке.

Среда Spring была разработана в конце 2002 года для упрощения разработки J2EE-приложений. Spring успешно выполняла эту задачу, предоставляя такие понятные и удобные Java-инфраструктуры и функциональные возможности, как Spring Security, Spring MVC, управление транзакциями, Spring-пакеты и Spring-интеграция. Разработчики Spring решили еще больше повысить производительность программистов. Для этого было создано средство разработки под названием Spring Roo.

Основой для начала работы является выбор и настраивание инструмента. Для программирования – это среда быстрой разработки приложений. Перед началом работы необходимо установить следующее программное обеспечение:

- JDK (Java Development Kit) – набор инструментов командной строки для компиляции и запуска Java-кода;
- SpringSource Tool Suite – набор плагинов для популярной IDE Eclipse с открытым исходным кодом для Java-разработки;
- Tomcat – распространенный сервер J2EE-приложений.

После установки и настройки для создания учебного приложения «PetClinic» (клиника домашних животных) необходимо:

1) добавить в PATH директории [путь к sts]\spring-roo\bin; [путь к sts]\apache-maven\bin;

2) открыть командную строку;

3) создать папку для проекта, например petclinic, и перейти в нее;

4) скопировать файл учебного проекта [путь к sts]\spring-roo\samples\clinic.roo в созданную вами папку petclinic;

5) запустить командную строку roo с командой создания учебного проекта:

roo script – file clinic.roo

6) настроить использование прокси-сервера Maven, открыть [путь к sts]\apache – maven – 3.0.4\conf\settings.xml и найти секцию proxies, указать настройки прокси для сети БГУИР;

7) собрать проект и запустить на web-сервере jetty

mvn jetty:run

8) открыть в браузере http://localhost:8080/petclinic, вы увидите приветственную страницу (рисунок 2).

Рисунок 2 – Приветственная страница SpringSource Tool Suite

Далее можно открыть проект и проанализировать его структуру. Для этого необходимо запустить STS и импортировать созданный проект File → import → Maven/Existing Maven Project, указать путь к petclinic.

Приложение можно запускать непосредственно из редактора Run → Run on Server (рисунки 3, 4).

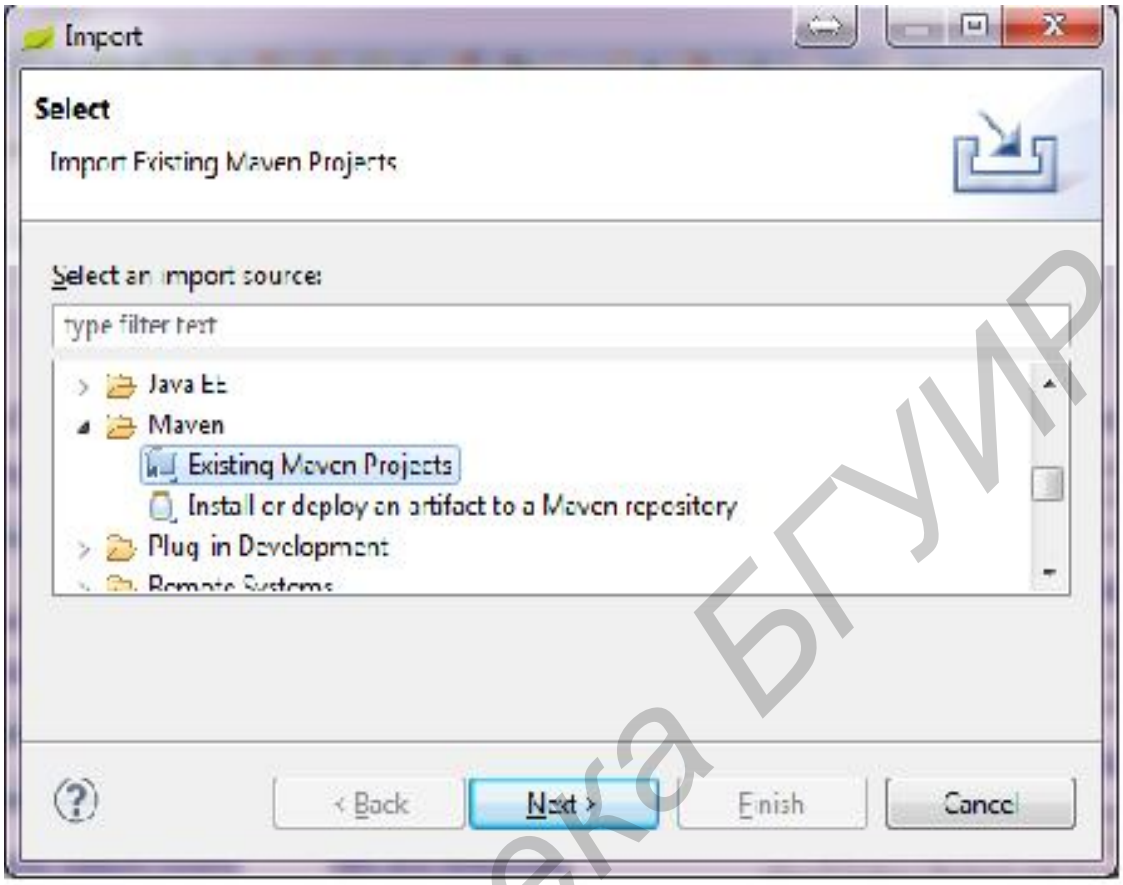

Рисунок 3 – Пример запуска приложения непосредственно из редактора Run → Run on Server

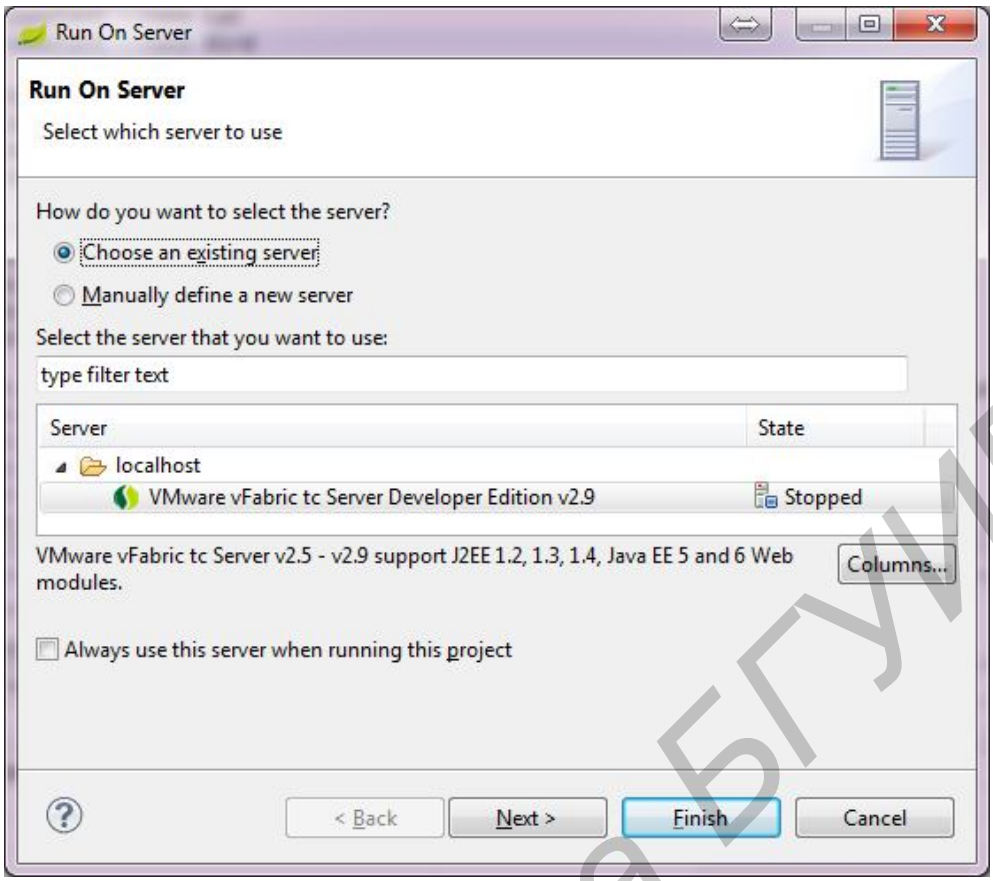

Рисунок 4 – Запуск приложения в SpringSource Tool Suite

Вместе с STS также поставляется сервер для работы с приложениями. Мы можем использовать этот сервер или добавить новый.

Модель учебного приложения состоит из следующих сущностей: Vet (ветеринар), Owner (владелец), Pet (домашнее животное), Visit (визит).

Задание для самостоятельной работы

Настроить интегрированную среду разработки Eclipse, подготовить и запустить учебное приложение «PetClinic». Согласно своему варианту подготовить модель данных для учебного проекта. Спроектировать и описать элементы модели и отношения между элементами.

Варианты заданий

1. Университет. Студент; преподаватель; предмет; оценка.
2. Футбольный чемпионат. Страна; клуб; игроки; матчи.
3. Сеть супермаркетов. Супермаркет; продавцы; продукты; поставщики.
4. Фирма экстремального спуска с гор. Средство спуска (сноуборд, лыжи и т. д.); мероприятия; услуги (душ, чай, успокоительное); клиент.
5. Продажа земельных участков на планетах. Планета; клиент; участок; расписание рейсов на планету.
6. Сеть зоопарков. Зоопарк; животное; рабочий; услуги (чистка, мойка, расчесывание, кормление).

7. Форум. Рубрика; тема; сообщение; пользователи.

8. Фруктовая компания. Страна; фрукт; получатель; способ доставки (стоимость за километр).

9. Компания «Пилорама». Лесник (поставляет дерево); тип дерева; цех деревообработки; выпускаемая продукция.

10. Столовая. Блюда; продукты; меню на день; клиент; заказ.

11. Фирма «Машина будущего». Средство передвижения (велосипед, самокат, и т. д.); поставщик; заказ транспортного средства; клиент.

12. Фирма проката свадебных платьев и аксессуаров. Платье; свадьба; заказ; клиент.

Тема 2. Язык Java

Цель: исследовать технологии написания программ с использованием языка программирования Java.

Теория и примеры выполнения задания

Ни одна программа, написанная на одном из языков высокого уровня, к которым относится и язык Java, так называемый исходный модуль, не может быть сразу же выполнена. Ее сначала надо откомпилировать, т. е. перевести в последовательность машинных команд. Исходный модуль, написанный на Java, не может избежать этих процедур, но здесь проявляется главная особенность технологии Java – программа компилируется сразу в машинные команды, но не какого-то конкретного процессора, а в команды так называемой виртуальной машины Java (JVM, Java Virtual Machine). Виртуальная машина Java – это совокупность команд вместе с системой их выполнения.

Итак, на первом этапе программа, написанная на языке Java, переводится компилятором в байт-коды. Эта компиляция не зависит от типа какого-либо конкретного процессора и архитектуры некоего конкретного компьютера. Она может быть выполнена один раз сразу же после написания программы. Байт-коды записываются в одном или нескольких файлах, могут храниться во внешней памяти или передаваться по сети. Это особенно удобно благодаря небольшому размеру файлов с байт-кодами. Затем полученные в результате компиляции байт-коды можно выполнять на любом компьютере, имеющем систему, реализующую JVM. При этом не важен ни тип процессора, ни архитектура компьютера.

Кроме реализации JVM для выполнения байт-кодов на компьютере еще нужно иметь набор функций, вызываемых из байт-кодов и динамически компонующихся с байт-кодами. Этот набор оформляется в виде библиотеки классов Java, состоящей из одного или нескольких пакетов. Каждая функция может быть записана байт-кодами, но, поскольку она будет храниться на конкретном компьютере, ее можно записать прямо в системе команд этого компьютера, избежав тем самым интерпретации байт-кодов. Такие функции

называют «родными методами» (native methods). Применение «родных методов» ускоряет выполнение программы.

Разработчики технологии Java распространяют набор необходимых программных инструментов для полного цикла работы с этим языком программирования (компиляции, интерпретации, отладки), включающий и богатую библиотеку классов под названием JDK (Java Development Kit). Есть наборы инструментальных программ и других фирм. Например, большой популярностью пользуется JDK фирмы IBM.

Набор JDK упаковывается в самораспаковывающийся архив. Скачав его из Интернета (http://java.sun.com/produets/jdk) и получив компакт-диск, вам остается только запустить файл с архивом на выполнение. Откроется окно установки, в котором вам будет предложено выбрать каталог (directory) установки. Вы можете согласиться с предлагаемым каталогом. Если вы указали собственный каталог, то проверьте после установки значение переменной PATH, набрав в командной строке окна MS-DOS Prompt (или окна Command Prompt в Windows NT/2000/XP) команду set. Переменная PATH должна содержать полный путь к подкаталогу bin этого каталога. Если нет, то добавьте этот путь, например C:\jdkl.6\bin. Надо определить и специальную переменную CLASSPATH, содержащую пути к архивным файлам и каталогам с библиотеками классов. Системные библиотеки Java 2 подключаются автоматически без переменной CLASSPATH.

Исходные файлы программ на языке java хранятся в файлах в текстовом формате с расширением *.java. Для компиляции и запуска необходимо использовать утилиты, входящие в состав JDK. Пусть для примера именем файла будет MyProgram.java, а сам файл сохранен в текущем каталоге. После создания этого файла из командной строки вызывается компилятор javac и ему передается исходный файл как параметр:

```
javac MyProgram.java
```

Компилятор создает в том же каталоге по одному файлу на каждый класс, описанный в программе, называя каждый файл именем класса с расширением class. Допустим, в нашем примере имеется только один класс, названный MyProgram, тогда получаем файл с именем MyProgram.class, содержащий байт-коды. Для запуска программы необходимо использовать утилиту java. Ей необходимо передать название пакетов, к которым относится класс MyProgram через точки и название класса. Допустим программа находится в пакете по умолчанию, в тексте не указан пакет. В этом случае формат запуска из командной строки будет иметь вид

```
java MyProgram
```

Для редактирования исходного кода и запуска программ удобно использовать интегрированные среды, такие как Eclipse. Для создания Java-программы необходимо:

1. Открыть STS Eclipse и создать Java-проект: File → New Java Project, ввести в открывшемся диалоге название проекта (рисунок 5).

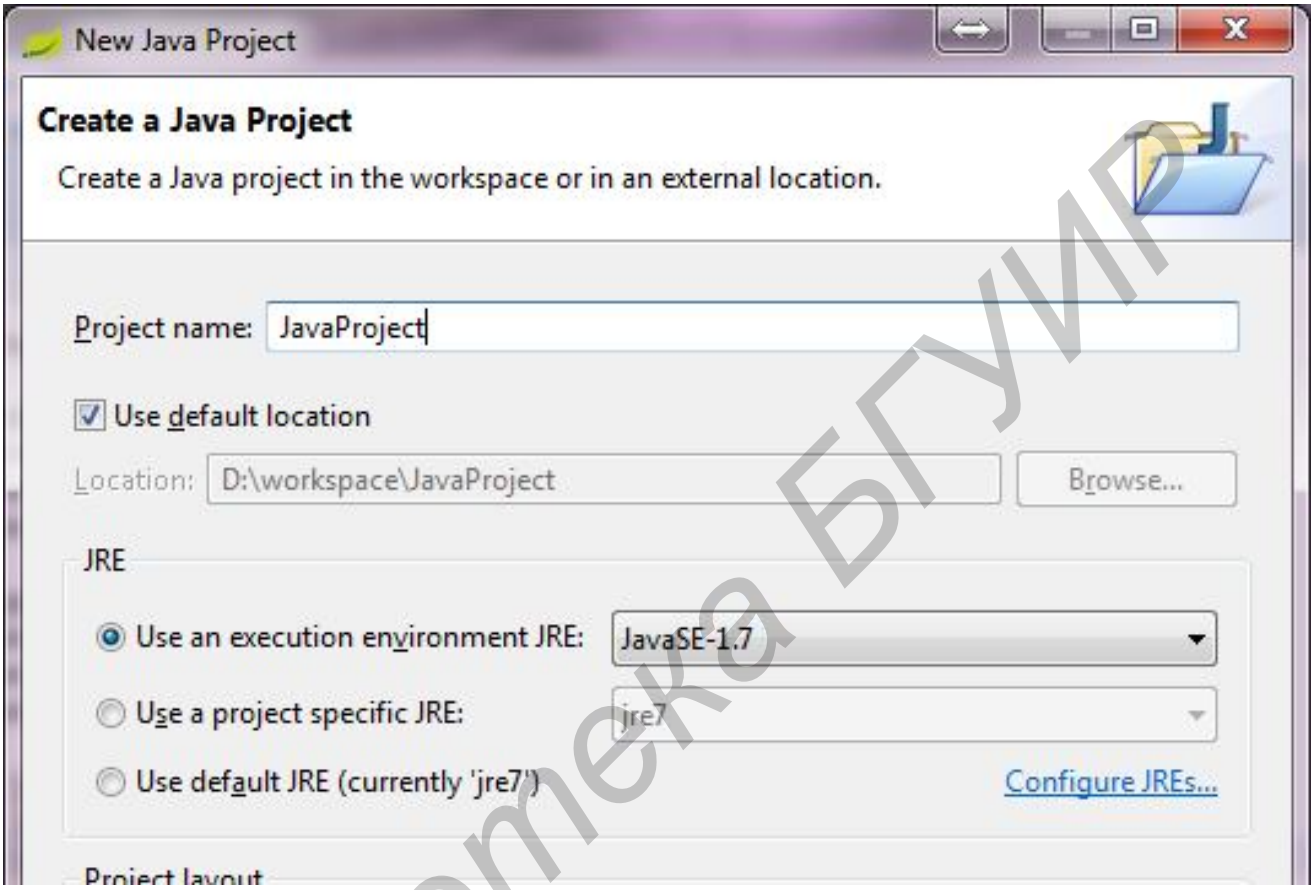

Рисунок 5 – Окно STS Eclipse

2. Добавить класс с методом main. Для этого отметить соответствующий пункт в диалоге (рисунок 6).

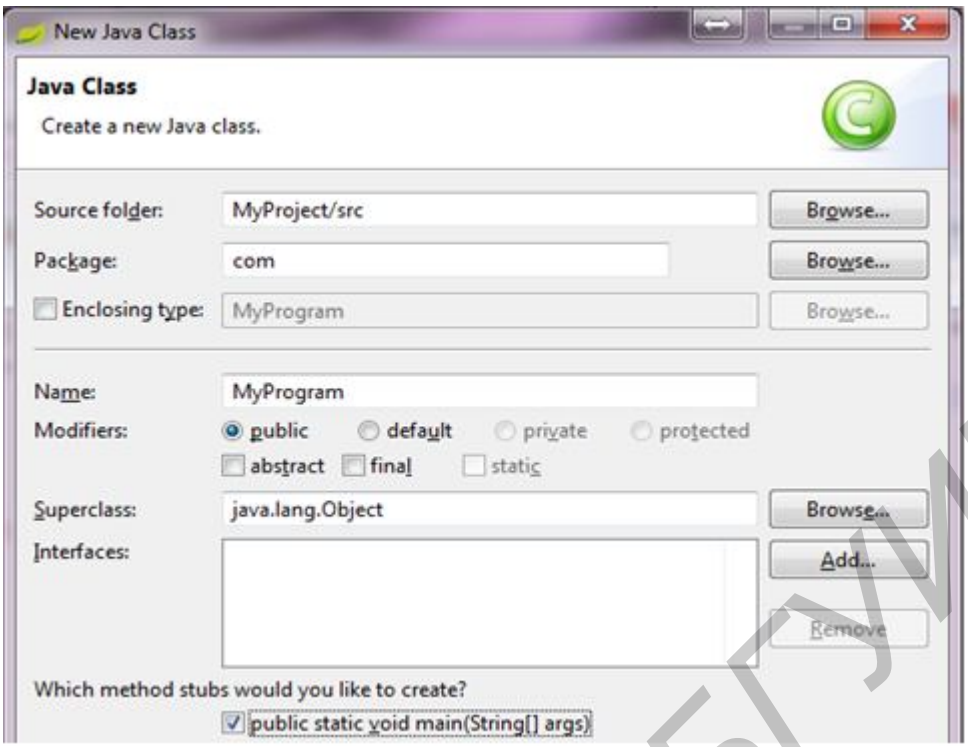

Рисунок 6 – Добавление класса в STS Eclipse

3. Можно добавить в созданный метод main код, выводящий строку в консоль: System.out.println(“Hello, World!”).

4. Запустить класс как приложение. Вызвать контекстное меню над созданным классом и выбрать Run As → Run As Java Application. Результат работы программы можно увидеть в консоли среды STS Eclipse. Созданная настройка запуска приложения должна выглядеть следующим образом (рисунок 7).

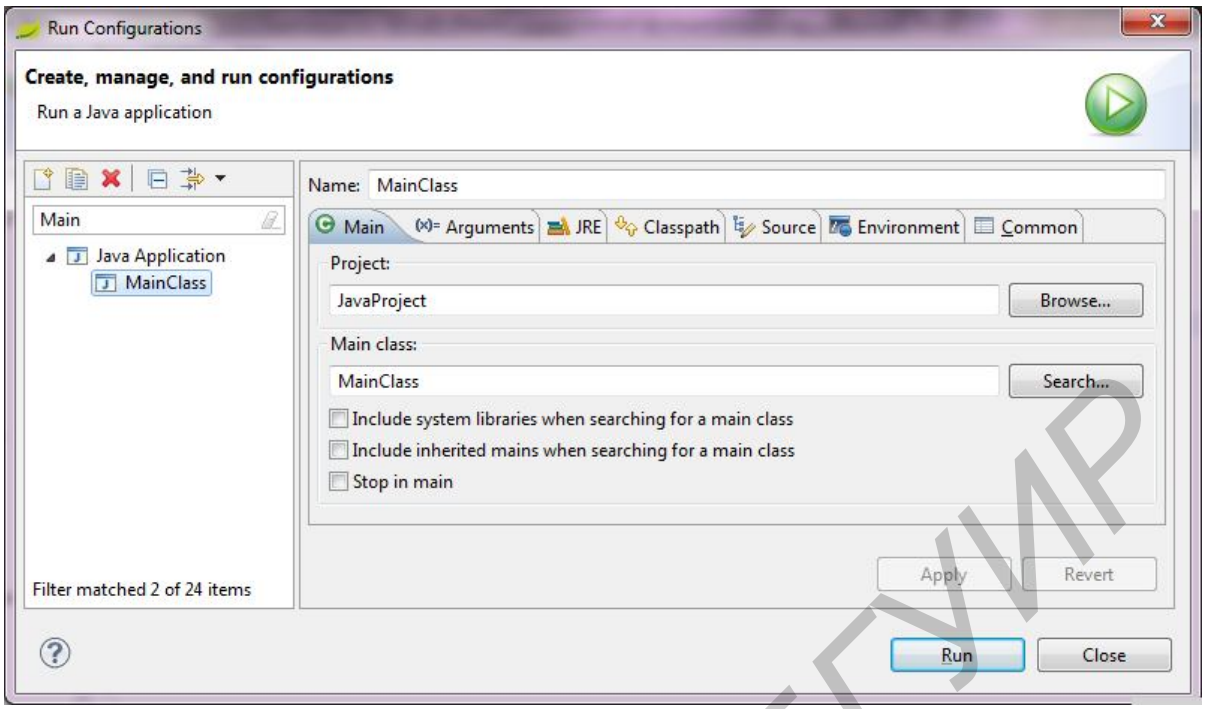

Рисунок 7 – Запуск приложения в STS Eclipse

В учебном примере классы модели данных находятся в пакете com.springsource.petclinic.domain (рисунок 8).

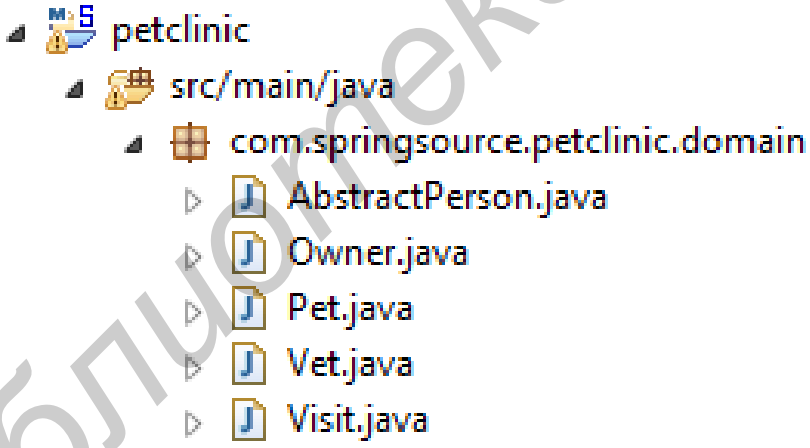

Рисунок 8 – Путь к классам модели в учебном примере

Эти классы описывают модель нашего web-приложения. Каждый класс реализует методы для сравнения объектов класса equals и для вывода информации в виде строки toString. Тип домашнего животного Pet.type имеет перечисляемый тип PetType { Dog, Cat, Bird; }.

Задание для самостоятельной работы

Разработать классы для хранения объектов модели согласно выданному варианту по теме 1, реализовать отношения между объектами. Использовать

массивы или коллекции из пакета java.utils. Информация из объектов должна выводиться в консоль при запуске приложения при помощи метода toString, объекты должны сортироваться при помощи метода compareTo и сравниваться методом equals. Обратите внимание, некоторые характеристики могут принимать только ограниченный набор значений, следовательно, необходимо использовать enum.

Тема 3. Работа со строками и текстом на языке Java

Цель: исследовать средства обработки текстовой информации с использованием платформы Java.

Теория и примеры выполнения задания

Очень большое место в обработке информации занимает работа с текстами. Как и многое другое, текстовые строки в языке Java являются объектами. Они представляются экземплярами класса String или класса StringBuffer .

Символы в строках хранятся в кодировке Unicode, в которой каждый символ занимает два байта. Тип каждого символа – char .

Перед работой со строкой ее следует создать. Это можно сделать разными способами.

Самый простой способ создать строку – это организовать ссылку типа String на строку-константу:

String si = "Это строка";

Объекты класса StringBuffer – это строки переменной длины. Только что созданный объект имеет буфер определенной емкости (capacity), по умолчанию достаточной для хранения шестнадцати символов. Емкость можно задать в конструкторе объекта. Как только буфер начинает переполняться, его емкость автоматически увеличивается, чтобы вместить новые символы.

Задача разбора введенного текста – парсинг (parsing) – вечная задача программирования наряду с сортировкой и поиском. Написана масса программ-парсеров (parser), разбирающих текст по различным признакам. В пакет java.util входит простой класс StringTokenizer, облегчающий разбор строк.

В разборе строки на слова активно участвуют следующие методы:

1) метод nextToken () возвращает в виде строки следующее слово;

2) логический метод hasMoreTokens () возвращает true, если в строке еще есть слова, и false, если слов больше нет;

3) метод countTokens () возвращает число оставшихся слов;

4) метод nextToken (String newDeiimeters) позволяет «на ходу» менять разделители. Следующее слово будет выделено по новым разделителям newDeiimeters; новые разделители действуют далее вместо старых разделителей, определенных в конструкторе или предыдущем методе nextToken ();

5) оставшиеся два метода nextEiement () и hasMoreEiements () реализуют интерфейс Enumeration. Они просто обращаются к методам nextToken () и hasMoreTokens().

Также можно манипулировать строкой, разбив ее на элементы путем использования регулярных выражений. Регулярные выражения – эта система обработки текста, основанная на специальной системе записи образцов для поиска. Образец (pattern), задающий правило поиска, по-русски также иногда называют шаблоном, маской. Сейчас регулярные выражения используются многими текстовыми редакторами и утилитами для поиска и изменения текста на основе выбранных правил. Язык программирования Java также поддерживает регулярные выражения для работы со строками.

Основными классами для работы с регулярные выражения являются класс java.util.regex.Pattern и класс java.util.regex.Matcher.

Класс java.util.regex.Pattern применяется для определения регулярных выражений, для которого ищется соответствие в строке, файле или другом объекте, представляющем собой некоторую последовательность символов. Для определения шаблона применяются специальные синтаксические конструкции. О каждом соответствии можно получить больше информации с помощью класса java.util.regex.Matcher.

Если в строке, проверяемой на соответствие, необходимо, чтобы в какой-либо позиции находился один из символов некоторого символьного набора, то такой набор (класс символов) можно объявить, используя одну из конструкций, представленных в таблице 1.

Таблица 1 – Способы определения классов символов

Шаблон	Описание
[abc]	a, b или c
[^abc]	Символ, исключая a, b и c
[a-z]	Символ между a и z
[a-d[m-p]]	Либо между a и d, либо между m и p
[e-z&&[dem]]	e либо m (конъюкция)

Кроме стандартных классов символов существуют предопределенные классы символов (таблица 2).

Таблица 2 – Дополнительные способы определения классов символов

Шаблон	Описание
1	2
.	Любой символ
\d	[0-9]
\D	[^0-9]
\s	[\t\n\x0B\f\r]

Продолжение таблицы 2

1	2
\S	[^ \s]
\w	[a-zA-Z_0-9]
\W	[^\w]
\p{javaLowerCase}	То же, что и Character.isLowerCase ()
\p{javaUpperCase}	То же, что и Character.isUpperCase ()

При создании регулярного выражения могут использоваться логические операции (таблица 3).

Таблица 3 – Способы задания логических операций

Шаблон	Описание
XY	После X следует Y
X|Y	X либо Y
(X)	X

Скобки, кроме их логического назначения, также используются для выделения групп. Для определения регулярных выражений недостаточно одних классов символов, т. к. в шаблоне часто нужно указать количество повторений. Для этого существуют квантификаторы (таблица 4).

Таблица 4 – Квантификаторы

Шаблон	Описание
X?	X один раз или ни разу
X*	X нуль или более раз
X+	X один или более раз
X{n}	X n раз
X{n,}	X n или более раз
X{n,m}	X от n до m

Для выполнения практического задания необходимо реализовать сохранение объектов в текст. Ранее мы создали модель и определили связи, теперь необходимо сохранить состояние объектов модели в строку и восстановить информацию из строки. Для этого нам потребуется создать парсер, работающий с текстом. Для этого необходимо:

1) создать в каждом объекте модели соответствующий метод, таким образом, каждый класс будет отвечать сам за сохранение своих данных;

2) определить формат сохранения данных модели, что особенно важно для массива коллекций объектов. Это могут быть строки в квадратных скобках, разделенные запятыми;

3) определить связи между объектами (подчиненные объекты должны сохранить ссылку на главный). В учебном проекте PetClinic, например, это объект Owner и объекты Pet. Поле id – поле, точно определяющее объект;

4) прочитать сохраненную информацию (сперва следует прочитать информацию для восстановления объектов, а затем восстановить связи).

На следующем занятии результат нашей работы будет выводиться в файл.

Задание для самостоятельной работы

Разработать функцию чтения и сохранения состояния объектов модели согласно полученному варианту по теме 1 в виде массива текстовых данных. Активно использовать средства для работы с текстом платформы Java, такие как StringTokenizer, метод split, PrintWriter (Charset). Разработать разметку для сохранения данных модели, выполнить чтение и запись для массивов и коллекций.

Тема 4. Система ввода/вывода платформы Java

Цель: изучить систему ввода/вывода платформы Java и средств для работы с потоками данных.

Теория и примеры выполнения задания

Для того чтобы отвлечься от особенностей конкретных устройств ввода/вывода, в Java употребляется понятие потока (stream). Считается, что в программу идет входной поток (input stream) символов Unicode или просто байт, воспринимаемый в программе методами read(). Из программы методами write() или prin(), println() выводится выходной поток (output stream) символов или байт. При этом неважно, куда направлен поток: на консоль, на принтер, в файл или в сеть, методы write() и print() ничего об этом не знают.

Можно представить себе поток как трубу, по которой в одном направлении последовательно «текут» символы или байты, один за другим. Методы read(), write(), print(), println() взаимодействуют с одним концом трубы, другой конец соединяется с источником или приемником данных конструкторами классов, в которых реализованы эти методы.

Конечно, полное игнорирование особенностей устройств ввода/вывода сильно замедляет передачу информации. Поэтому в Java все-таки выделяется файловый ввод/вывод, вывод на печать, сетевой поток.

Три потока определены в классе system статическими полями in, out и err. Их можно использовать без всяких дополнительных определений. Они называются соответственно стандартным вводом (stdin), стандартным выводом (stdout) и стандартным выводом сообщений (stderr). Эти стандартные потоки могут быть соединены с разными конкретными устройствами ввода и вывода.

Потоки out и err – это экземпляры класса PrintStream, организующего выходной поток байт. Эти экземпляры выводят информацию на консоль методами print(), println() и write(), которых в классе PrintStream имеется около двадцати для разных типов аргументов.

Поток err предназначен для вывода системных сообщений программы: трассировки, сообщений об ошибках или просто о выполнении каких-то этапов программы. Такие сведения обычно заносятся в специальные журналы, log-

файлы, а не выводятся на консоль. В Java есть средства переназначения потока, например, с консоли в файл.

Поток in – это экземпляр класса InputStream. Он назначен на клавиатурный ввод с консоли методами read(). Класс InputStream абстрактный, поэтому реально используется какой-то из его подклассов.

Понятие потока оказалось настолько удобным и облегчающим программирование ввода/вывода, что в Java предусмотрена возможность создания потоков, направляющих символы или байты не на внешнее устройство, а в массив или из массива, т. е. связывающих программу с областью оперативной памяти. Более того, можно создать поток, связанный со строкой типа string находящейся опять-таки в оперативной памяти. Кроме того, можно создать канал (pipe) обмена информацией между подпроцессами.

Еще один вид потока – поток байт, составляющих объект Java. Его можно направить в файл или передать по сети, а потом восстановить в оперативной памяти. Эта операция называется сериализацией (serialization) объектов.

Методы организации потоков собраны в классы пакета Java.io. Кроме классов, организующих поток, в пакет Java.io входят классы с методами преобразования потока, например, можно преобразовать поток байт, образующих целые числа, в поток этих чисел.

В Java есть целых четыре иерархии классов для создания, преобразования и слияния потоков:

1) Reader – абстрактный класс, в котором собраны самые общие методы символьного ввода;

2) Writer – абстрактный класс, в котором собраны самые общие методы символьного вывода;

3) InputStream – абстрактный класс с общими методами байтового ввода;

4) OutputStream – абстрактный класс с общими методами байтового вывода.

Классы входных потоков Reader и InputStream определяют по четырем методам ввода:

1) read() – возвращает один символ или байт, взятый из входного потока, в виде целого значения типа int; если поток уже закончился, возвращает -1;

2) read (char[] buf) – заполняет заранее определенный массив buf символами из входного потока; в классе InputStream массив типа byte[] заполняет байтами; метод возвращает фактическое число взятых из потока элементов или -1, если поток уже закончился;

3) read (char[] buf, int offset, int len) – заполняет часть символьного или байтового массива buf, начиная с индекса offset, число взятых из потока элементов равно len; метод возвращает фактическое число взятых из потока элементов или -1;

4) skip (long n) «проматывает» поток с текущей позиции на n символов или байт вперед. Эти элементы потока не вводятся методами read(). Метод возвращает реальное число пропущенных элементов, которое может отличаться от n, например, поток может закончиться.

Первые три метода выбрасывают IOException, если произошла ошибка ввода/вывода.

Текущий элемент потока можно пометить методом mark (int n), а затем вернуться к помеченному элементу методом reset o, но не более чем через n элементов. Не все подклассы реализуют эти методы, поэтому перед расстановкой пометок следует обратиться к логическому методу marksupported(), который возвращает true, если реализованы методы расстановки и возврата к пометкам.

Классы выходных потоков Writer и OutputStream определяют по три почти одинаковых метода вывода:

1) write (char[] buf) – выводит массив в выходной поток, в классе Outputstream массив имеет тип byte[];

2) write (char[] buf, int offset, int len) – выводит len элементов массива buf, начиная с элемента с индексом offset;

3) write (int elem) в классе Writer – выводит шестнадцать, а в классе OutputStream восемь младших бит аргумента elem в выходной поток.

В классе Writer есть еще два метода:

1) write (String s) – выводит строку s в выходной поток;

2) write (String s, int offset, int len) – выводит len символов строки s, начиная с символа с номером offset.

Многие подклассы классов Writer и OutputStream осуществляют буферизованный вывод. При этом элементы сначала накапливаются в буфере и оперативной памяти, а затем выводятся в выходной поток после того, как буфер заполнится. Это удобно для выравнивания скоростей вывода из программы и вывода потока, но часто надо вывести информацию в поток еще до заполнения буфера. Для этого предусмотрен метод flush(). Данный метод сразу же выводит все содержимое буфера в поток. Наконец, по окончании работы с потоком его необходимо закрыть методом closed.

Классы, входящие в иерархии потоков ввода/вывода, показаны на рисунках 9 и 10.

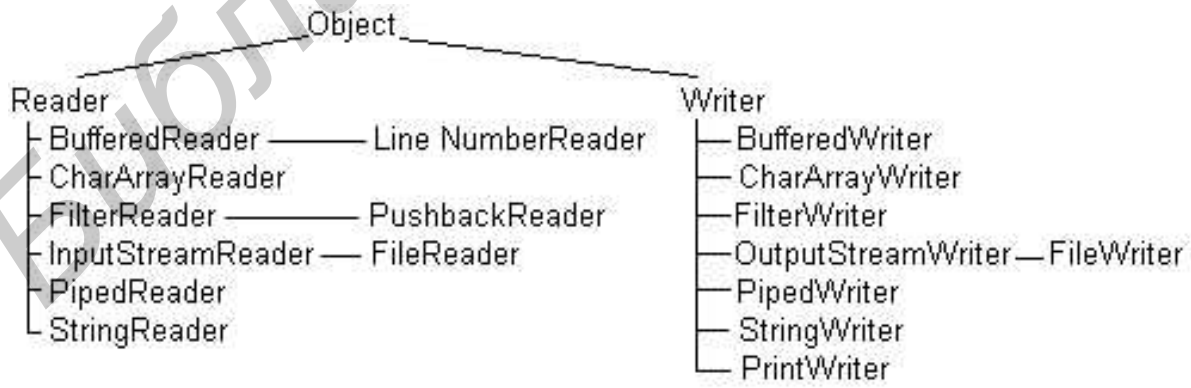

Рисунок 9 – Иерархия символьных потоков

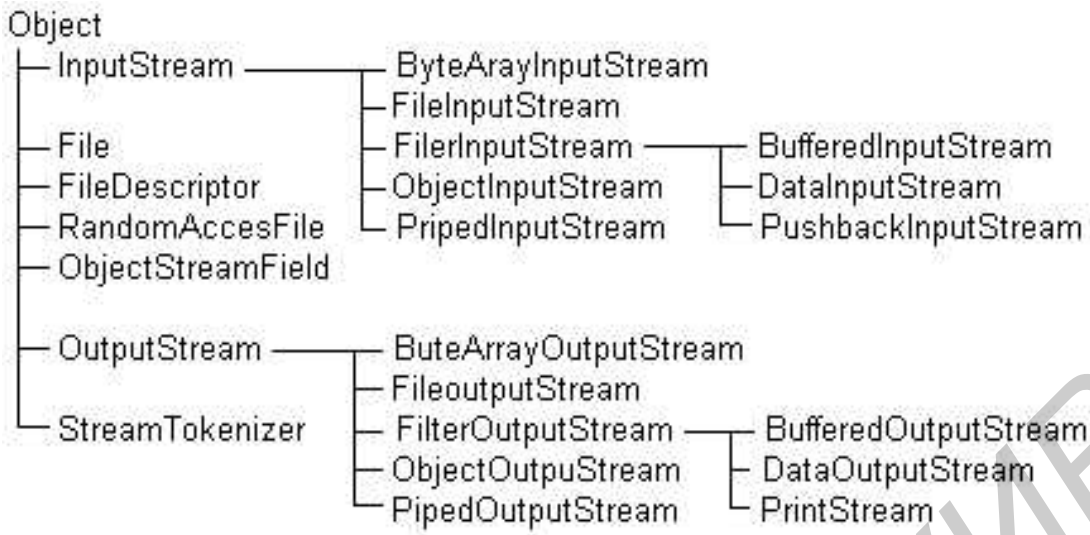

Рисунок 10 – Классы байтовых потоков

Все классы пакета java.io можно разделить на две группы: классы, создающие поток (data sink), и классы, управляющие потоком (data processing). Классы, создающие потоки, в свою очередь, можно разделить на пять групп:

1) классы, создающие потоки, связанные с файлами:
   - FileReader;
   - FileInputStream;
   - FileWriter;
   - FileOutputstream;
   - RandomAccessFile;
2) классы, создающие потоки, связанные с массивами:
   - CharArrayReader;
   - ByteArrayInputStream;
   - CharArrayWriter;
   - ByteArrayOutputStream;
3) классы, создающие каналы обмена информацией между подпроцессами:
   - PipedReader;
   - PipedInputStream;
   - PipedWriter;
   - PipedOutputStream;
4) классы, создающие символьные потоки, связанные со строкой:
   - StringReader;
   - StringWriter;
5) классы, создающие байтовые потоки из объектов Java:
   - ObjectInputStream;
   - ObjectOutputStream.

Классы, управляющие потоком, получают в своих конструкторах уже имеющийся поток и создают новый, преобразованный поток. Можно представлять их себе как «переходное кольцо», после которого идет труба другого диаметра.

Четыре класса созданы специально для преобразования потоков:

1) FilterReader;

2) FilterlnputStream;

3) FilterWriter;

4) FilterOutputStream.

Сами по себе эти классы бесполезны – они выполняют тождественное преобразование. Их следует расширять, переопределяя методы ввода/вывода. Но для байтовых фильтров есть полезные расширения, которым соответствуют некоторые символьные классы.

Четыре класса выполняют буферизованный ввод/вывод:

1) BufferedReader;

2) BufferedlnputStream;

3) BufferedWriter;

4) BufferedOutputStream.

Два класса преобразуют поток байт, образующих восемь простых типов Java, в эти самые типы:

1) DatalnputStream;

2) DataOutputStream.

Два класса содержат методы, позволяющие вернуть несколько символов или байт во входной поток:

1) PushbackReader;

2) PushbacklnputStream.

Два класса связаны с выводом на строчные устройства – экран дисплея, принтер:

1) PrintWriter;

2) PrintStream.

Два класса связывают байтный и символьный потоки:

1) InputStreamReader – преобразует входной байтовый поток в символьный поток;

2) OutputStreamWriter – преобразует выходной символьный поток в байтовый поток.

Класс StreamTokenizer позволяет разобрать входной символьный поток на отдельные элементы (tokens).

Из управляющих классов выделяется класс SequenceInputStream, сливающий несколько потоков, заданных в конструкторе, в один поток и класс.

LineNumberReader «умеет» читать выходной символьный поток построчно. Строки в потоке разделяются символами '\n' и/или '\r'.

Для вывода на консоль использовался метод println() класса InputStream, никогда не определяя экземпляры этого класса. Использовалось статическое поле out класса System, которое является объектом класса PrintStream. Исполняющая система Java связывает это поле с консолью.

Консоль является байтовым устройством, и символы Unicode перед выводом на консоль должны быть преобразованы в байты. Для символов Latin1 с кодами '\u0000' – '\u00FF' при этом просто откидывается нулевой старший байт и выводятся байты '\0x00' – '\0xFF'. Для кодов кириллицы, которые лежат в диапазоне '\u0400' – '\u04FF' кодировки Unicode, и других национальных алфавитов производится преобразование по кодовой таблице, соответствующей установленной на компьютере локале.

Трудности с отображением кириллицы возникают, если вывод на консоль производится в кодировке, отличной от локали. Именно так происходит в русифицированных версиях MS Windows NT/2000. Обычно в них устанавливается локаль с кодовой страницей CP1251, а вывод на консоль происходит в кодировке CP866. В этом случае надо заменить PrintStream, который не может работать с символьным потоком, на PrintWriter и «вставить переходное кольцо» между потоком символов Unicode и потоком байт system и out, выводимых на консоль, в виде объекта класса OutputStreamWriter. В конструкторе этого объекта следует указать нужную кодировку.

Класс PrintStream буферизует выходной поток. Второй аргумент его конструктора true вызывает принудительный сброс содержимого буфера в выходной поток после каждого выполнения метода println(). Но после print() буфер не сбрасывается! Для сброса буфера после каждого print() надо писать flush().

Поскольку файлы в большинстве современных операционных систем понимаются как последовательность байт, для файлового ввода/вывода создаются байтовые потоки с помощью классов FileInputStream и FileOutputStream. Это особенно удобно для бинарных файлов, хранящих байт-коды, архивы, изображения, звук.

Очень много файлов содержат тексты, составленные из символов. Несмотря на то, что символы могут храниться в кодировке Unicode, эти тексты чаще всего записаны в байтовых кодировках. Поэтому и для текстовых файлов можно использовать байтовые потоки. В таком случае со стороны программы придется организовать преобразование байт в символы и обратно.

Чтобы облегчить это преобразование, в пакет java.io введены классы FineReader и FileWriter. Они организуют преобразование потока: со стороны программы потоки символьные, со стороны файла – байтовые. Это происходит потому, что данные классы расширяют классы InputStreamReader и OutputStreamWriter, а значит, содержат «переходное кольцо» внутри себя. Несмотря на различие потоков, использование классов файлового ввода/вывода очень похоже.

В конструкторах всех четырех файловых потоков задается имя файла в виде строки типа string или ссылка на объект класса File. Конструкторы не только создают объект, но и отыскивают файл и открывают его.

При неудаче выбрасывается исключение класса FileNotFoundException, но конструктор класса FileWriter выбрасывает более общее исключение IOException.

После открытия выходного потока типа FileWriter или FileQutputStream содержимое файла, если он был не пуст, стирается. Для того чтобы можно было делать запись в конец файла, и в том и в другом классе предусмотрен конструктор с двумя аргументами. Если второй аргумент равен true, то происходит дозапись в конец файла, если false – файл заполняется новой информацией. По окончании работы с файлом поток следует закрыть методом close().

Преобразование потоков в классах FileReader и FileWriter выполняется по кодовым таблицам, установленным на компьютере локали. Для правильного ввода кирилицы надо применять FileReader, а не FileInputStream. Если файл содержит текст в кодировке, отличной от локальной кодировки, то придется вставлять «переходное кольцо» вручную.

Операции ввода/вывода по сравнению с операциями в оперативной памяти выполняются очень медленно. Для компенсации в оперативной памяти выделяется некоторая промежуточная область – буфер, в которой постепенно накапливается информация. Когда буфер заполнен, его содержимое быстро переносится процессором, буфер очищается и снова заполняется информацией.

Пример буфера – почтовый ящик, в котором накапливаются письма. Мы бросаем в него письмо и уходим по своим делам, не дожидаясь приезда почтовой машины. Почтовая машина периодически очищает почтовый ящик, перенося сразу большое число писем. Представьте себе город, в котором нет почтовых ящиков, и толпа людей с письмами в руках дожидается приезда почтовой машины.

Классы файлового ввода/вывода не занимаются буферизацией. Для этой цели есть четыре специальных класса BufferedXxx, перечисленных ранее.

Ранее были разработаны алгоритмы сохранения состояния модели в строку, теперь требуется определить способ записи информации в файл. Для того чтобы сохранить строку с состоянием объектов в файловый поток, используются потоки ввода/вывода. Для этого необходимо:

1) создать ссылку на файл File и проверить, есть ли уже такой файл;

2) открыть файловый поток и направить строку с состоянием объектов в этот поток.

Для чтения открываем файловый поток и направляем его в соответствующий метод восстановления объектов.

То, что используются файловые потоки, позволяет сохранять состояние объекта не только в файл, но и в любой другой поток, например по сети.

Задание для самостоятельной работы

Разработать сохранение текстовых данных модели в один или несколько файлов. Сохранение данных должно производиться централизовано, избегая дублирование кода в программе. Предусмотреть возможность изменения алгоритма сохранения при помощи потоков разного типа (запись в файл, запись по сети). Использовать ранее разработанные алгоритмы чтения и записи.

Тема 5. Пакет Java.utils

Цель: исследовать пакет Java.utils для упрощения разработки программ, средств обработки исключительных ситуаций в программе.

Теория и примеры выполнения задания

При программировании в Java операций над группой однотипных объектов важно выбирать наиболее эффективную структуру данных (класс) для хранения этих объектов. В языке Java определены специальные классы для хранения однотипных объектов, которые называются коллекциями, определяющими такие структуры, как список, множество, очередь.

Выбор определенного класса для работы с коллекциями определяет набор методов, которые будут доступны для объекта этого класса. Например, если используется список, который определяет интерфейс List, то существует богатый выбор для его реализации: ArrayList, LinkedList, Vector, Stack. Конкретный выбор реализации списка сказывается на эффективности манипуляций с объектами списка. Так, ArrayList хранит элементы в виде массива, а значит, доступ и замена будут выполняться относительно быстро. В то же время LinkedList хранит элементы в виде связанного списка, что влечет за собой относительно медленный поиск элементов и быструю операцию добавления/удаления.

Интерфейс Collection содержит набор общих методов, которые используются в большинстве коллекций. Рассмотрим основные из них:

1) add(Object item) – добавляет в коллекцию новый элемент, если элементы коллекции каким-то образом упорядочены, новый элемент добавляется в конец коллекции;
2) clear() – удаляет все элементы коллекции;
3) contains(Object obj) – возвращает true, если объект obj содержится в коллекции и false, если нет;
4) isEmpty() – проверяет, пуста ли коллекция;
5) remove(Object obj) – удаляет из коллекции элемент obj, возвращает false, если такого элемента в коллекции не нашлось;
6) size() – возвращает количество элементов коллекции.

Интерфейс List описывает упорядоченный список. Элементы списка пронумерованы, начиная с нуля, и к конкретному элементу можно обратиться по целочисленному индексу. Интерфейс List является наследником интерфейса

Collection, поэтому содержит все его методы и добавляет к ним несколько своих:

1) add(int index, Object item) – вставляет элемент item в позицию index, при этом список раздвигается (все элементы, начиная с позиции index, увеличивают свой индекс на единицу);

2) get(int index) – возвращает объект, находящийся в позиции index;

3) indexOf(Object obj) – возвращает индекс первого появления элемента obj в списке;

4) lastIndexOf(Object obj) – возвращает индекс последнего появления элемента obj в списке;

5) add(int index, Object item) – заменяет элемент, находящийся в позиции index объектом item;

6) subList(int from, int to) – возвращает новый список, представляющий собой часть данного (начиная с позиции from до позиции to включительно).

Интерфейс Set описывает множество. Элементы множества не упорядочены. Множество не может содержать двух одинаковых элементов. Интерфейс Set унаследован от интерфейса Collection, но никаких новых методов не добавляет. Изменяется только смысл метода add(Object item) – он не добавляет объект item, если он уже присутствует во множестве.

Интерфейс Queue описывает очередь. Элементы могут добавляться в очередь только с одного конца, а извлекаться с другого (аналогично очереди в магазине). Интерфейс Queue также унаследован от интерфейса Collection. Специфическими для очереди являются следующие методы:

1) poll() – возвращает первый элемент и удаляет его из очереди;

2) peek() – возвращает первый элемент очереди, не удаляя его;

3) offer(Object obj) – добавляет в конец очереди новый элемент и возвращает true, если вставка удалась.

Vector (вектор) – набор упорядоченных элементов, к каждому из которых можно обратиться по индексу. По сути эта коллекция представляет собой обычный список. Класс Vector реализует интерфейс List, основные методы которого приведены выше. К этим методам добавляется еще несколько. Например, метод firstElement() позволяет обратиться к первому элементу вектора, метод lastElement() – к его последнему элементу. Метод removeElementAt(int pos) удаляет элемент в заданной позиции, а метод removeRange(int begin, int end) удаляет несколько подряд идущих элементов. Все эти операции можно было бы осуществить комбинацией базовых методов интерфейса List, так что функциональность принципиально не меняется.

Класс ArrayList – аналог класса Vector. Он представляет собой список и может использоваться в тех же ситуациях. Основное отличие состоит в том, что он не синхронизирован и одновременная работа нескольких параллельных процессов с объектом этого класса не рекомендуется. В обычных же ситуациях он работает быстрее.

Stack – коллекция, объединяющая элементы в стек. Стек работает по принципу LIFO («последним пришел – первым ушел»). Элементы кладутся в

стек «друг на друга», причем взять можно только «верхний» элемент, т. е. тот, который был положен в стек последним. Для стека характерны операции, реализованные в следующих методах класса Stack:

1) push(Object item) – помещает элемент на вершину стека;
2) pop() – извлекает из стека верхний элемент;
3) peek() – возвращает верхний элемент, не извлекая его из стека;
4) empty() – проверяет, не пуст ли стек;
5) search(Object item) – ищет «глубину» объекта в стеке. Верхний элемент имеет позицию «1», находящийся под ним – «2» и т. д. Если объекта в стеке нет, возвращает позицию «–1».

Класс Stack является наследником класса Vector, поэтому имеет все его методы (и, разумеется, реализует интерфейс List). Однако, если в программе нужно моделировать именно стек, рекомендуется использовать только пять вышеперечисленных методов.

Преимущество использования массивов и коллекций заключается не только в том, что можно поместить в них произвольное количество объектов и извлекать их при необходимости, но и в том, что все эти объекты можно комплексно обрабатывать. Например, вывести на экран все шашки, содержащиеся в списке checkers. В случае массива мы пользуемся следующим циклом:

```
for (int i = 1; i < array.length; i++){
        // обрабатываем элемент array[i]
}
```

Имея дело со списком, мы можем поступить аналогичным образом, только вместо array[i] писать array.get(i). Но мы не можем поступить так с коллекциями, элементы которых не индексируются (например, очередью или множеством). А в случае индексированной коллекции надо хорошо знать особенности ее работы: как определить количество элементов, как обратиться к элементу по индексу, может ли коллекция быть разреженной (т. е. могут ли существовать индексы, с которыми не связано никаких элементов) и т. д.

Для навигации по коллекциям в Java предусмотрено специальное архитектурное решение, получившее свою реализацию в интерфейсе Iterator. Идея заключается в том, что к коллекции «привязывается» объект, единственное назначение которого – выдать все элементы этой коллекции в некотором порядке, не раскрывая ее внутреннюю структуру.

Интерфейс Iterator имеет всего три метода:

1) next() – возвращает очередной элемент коллекции, к которой «привязан» итератор, и делает его текущим; порядок перебора определяет сам итератор;
2) hasNext() – возвращает true, если перебор элементов еще не закончен;
3) remove() – удаляет текущий элемент.

Интерфейс Collection помимо рассмотренных ранее методов, имеет метод iterator(), который возвращает итератор для данной коллекции, готовый к ее обходу. С помощью такого итератора можно обработать все элементы любой коллекции следующим простым способом:

```
Iterator iter = coll.iterator(); // coll - коллекция
while (iter.hasNext()) {
        // обрабатываем объект, возвращаемый методом iter.next()
}
```

Для коллекций, элементы которых проиндексированы, определен более функциональный итератор, позволяющий двигаться как в прямом, так и в обратном направлении, а также добавлять в коллекцию элементы. Такой итератор имеет интерфейс ListIterator, унаследованный от интерфейса Iterator и дополняющий его следующими методами:

1) previous() – возвращает предыдущий элемент и делает его текущим;
2) hasPrevious() – возвращает true, если предыдущий элемент существует (т. е. текущий элемент не является первым элементом для данного итератора);
3) add(Object item) – добавляет новый элемент перед текущим элементом;
4) set(Object item) – заменяет текущий элемент;
5) nextIndex() и previousIndex() – служат для получения индексов следующего и предыдущего элементов соответственно.

В интерфейсе List определен метод ListIterator(), возвращающий итератор ListIterator для обхода данного списка.

Интерфейс Map из пакета Java.util описывает коллекцию, состоящую из пар «ключ – значение», которые широко используются для хранения настроек в файлах конфигурации (например /etc/services). У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения (Map). Интерфейс Map содержит следующие методы, работающие с ключами и значениями:

1) boolean containsKey (Object key) – проверяет наличие ключа key;
2) boolean containsValue (Object value) – проверяет наличие значения value;
3) Set entry Set() – представляет коллекцию в виде множества, каждый элемент которого – пара из данного отображения, с которой можно работать методами вложенного интерфейса Map.Entry;
4) Object get(Object key) – возвращает значение, отвечающее ключу key;
5) Set key S et() – представляет ключи коллекции в виде множества;
6) Object put (Object key, Object value) – добавляет пару « key-value », если такой пары не было, и заменяет значение ключа key, если такой ключ уже есть в коллекции;
7) void putAll (Map m) – добавляет к коллекции все пары из отображения m;
8) collection values() – представляет все значения в виде коллекции.

В интерфейс Map вложен интерфейс Map.Entry. Этот интерфейс описывает методы работы с парами, полученными методом entrySet() из объекта типа Map.

Методы getKey() и getValue() позволяют получить ключ и значение пары, метод setValue (Object value) меняет значение в данной паре.

Для того чтобы упростить работу с массивами объектов, в созданной модели будем использовать коллекции. Коллекции нужны для того, чтобы автоматически резервировать память хранения, выполнять операции сортировки, добавления и удаления. Для этого следует:

1) определить типы коллекций, которые нужны. Какую коллекцию использовать – зависит от логики работы программы. Для хранения неповторяющихся объектов лучше использовать Set, для хранения несортированных данных лучше подходит List, коллекция Map используется для быстрого доступа к объектам коллекции по ключам;

2) добавить в классы модели методы давления и удаления зависимых объектов, также необходимо добавить конструкторы с параметрами для создания и одновременной инициализации объектов;

3) определить порядок для хранения объектов. Для этого необходимо создать компаратор для коллекции, объекты которой должны храниться в определенном порядке.

Коллекция хранит объекты в определенном порядке, а также требует сортировать их после каждой операции добавления и удаления. Все это замедляет ее работу. Такой вариант использования коллекции оправдан, если в приложении часто выполняются операции чтения списка и редко изменяются данные. Альтернативное решение – хранить несортированную коллекцию и добавить метод, который сортирует коллекцию каждый раз перед тем, как возвращает список объектов. Такой подход оправдан, если в программе часто добавляются и удаляются объекты и редко требуется сортированный список.

Задание для самостоятельной работы

Разработать бизнес-методы обработки данных согласно выданному варианту. Использовать готовые алгоритмы сортировки; итераторы для поиска и верификации информации; разработать классы для обработки исключительных ситуаций и в случае обработки некорректных данных, используя класс Exception.

Варианты заданий

1. Университет. Подсчет среднего балла для студента при вводе. При отчислении студента отправлять ему сообщение на почту. Отчисление студента производится автоматически, когда у него четыре неудовлетворительные оценки.

2. Футбольный чемпионат. После каждого матча проставлять очки клубу. Отсылать поздравление на почту президента клуба. В случае проигрыша – отсылка всем игрокам уведомления об уменьшении зарплаты в данном месяце.

3. Сеть супермаркетов. При уменьшении количества продуктов до критической массы автоматически отсылается заказ (почтой) поставщику продукта. Обновлять стоимость товаров списком новых цен при поставке новой партии. Сортировать продавцов по результативности.

4. Фирма экстремального спуска с гор. Подсчитывать общую стоимость пакета услуг клиента, снимать деньги со счета за предоставление услуги и отправлять уведомления о произведенных операциях клиенту. Сортировать мероприятия по поступившим на счет фирмы деньгам.

5. Продажа земельных участков на планетах. Подсчитывать остаток свободного места на планете после вычитания площади все занятых участков. При уничтожении планеты астероидом или захвате внеземными цивилизациями (изменение статуса планеты) отсылать сообщения всем владельцам участков на планете. Выводить ближайшие рейсы между выбранными планетами.

6. Сеть зоопарков. При добавлении нового животного связывать его с работником, у которого наименьшее количество животных такого вида. Также отсылать работнику сообщение на почту о добавлении животного. Выводить все зоопарки города.

7. Форум. Фильтровать добавленные сообщения согласно справочнику некультурных слов. Отправлять сообщения пользователю о непринятии сообщения. Позволять создавать тему пользователю, если у него более десяти сообщений.

8. Фруктовая компания. Не добавлять информацию о доставке, если время хранения фруктов меньше, чем время доставки. Рассчитывать стоимость доставки. Отправлять сообщение получателю о подтверждении доставки.

9. Компания «Пилорама». Не позволять смешивать в поставках хвойные и лиственные породы древесины. На каждые десять кубометров древесины лесник должен добавлять одиннадцатый, который будет бесплатным для покупателя. Отправлять сообщение о подтверждении получателю.

10. Столовая. Не позволять смешивать мясные и рыбные блюда в одном заказе. В один заказ не принимать блюда из сельди и молочных продуктов. После формирования меню отправлять информацию о нем клиентам по почте.

11. Фирма «Машина будущего». Подсчитывать стоимость транспортного средства (стоимость деталей и сборки). Выводить количество транспортных средств, которое может быть произведено из доступных деталей. Отправлять уведомление клиенту о заказе.

12. Фирма «Шпионаж». Не позволять вербовать агенту более восьми людей, т. к. за ними становится трудно следить. При переводе кого-либо из персонала в статус «рассекречен» отправлять сообщение в штаб-квартиру об угрозе раскрытия агента. Подсчитывать эффективность сбора документов агентами и их работы.

Тема 6. Библиотека Swing

Цель: изучить технологии создания пользовательских интерфейсов с использованием библиотеки Swing.

Теория и примеры выполнения задания

Библиотека Swing используется для создания десктопных приложений на Java. Графический пользовательский интерфейс (GUI) – основной способ взаимодействия конечных пользователей с Java-приложением. Для разработки прикладного программного обеспечения на языке Java, а точнее графического интерфейса приложений, обычно используются пакеты AWT и Swing.

AWT (для доступа загружается пакет Java.awt) содержит набор классов, позволяющих выполнять графические операции и создавать оконные элементы управления.

Swing (для доступа загружается пакет Javax.swing) содержит новые классы, в основном аналогичные AWT. К именам классов добавляется J (JButton, JLabel и др).

На данный момент основные классы для построения визуальных интерфейсов содержатся в пакете Swing. Из пакета AWT используются классы для обработки сообщений. Простейшее графическое приложение приведено ниже.

```
import javax. swing.*;
public final class HelloWorld implements Runnable {
  public static void main(String[] args) {
 //Swing имеет собственный управляющий поток (т. н. dispatching thread),
 //который работает параллельно с основным (в котором выполняется main())
 //потоком. Если основной поток закончит работу (метод main завершится),
 //поток, отвечающий за работу Swing-интерфейса, может продолжать свою работу.
 //И даже если пользователь закрыл все окна, программа продолжит свою работу
 //(до тех пор, пока жив данный поток). Начиная с Java 6, когда все
 //компоненты уничтожены, управляющий поток останавливается автоматически.
 //Запускаем весь код, работающий в управляющем потоке, даже инициализацию:

 SwingUtilities.invokeLater (new HelloWorld());
}

public void run() {
 // Создаем окно с заголовком "Hello, World!"
 Frame f = new JFrame ("Hello, World!");

// Ранее практиковалось следующее: создавался listener и регистрировался
// на экземпляре главного окна, который реагировал на windowClosing()
// принудительной остановкой виртуальной машины вызовом System.exit()
// Теперь же есть более «правильный» способ – задать реакцию на закрытие окна.
// Данный способ уничтожает текущее окно, но не останавливает приложение. Тем
// самым приложение будет работать, пока не будут закрыты все окна.
 f.setDefaultCloseOperation (JFrame. DISPOSE_ON_CLOSE );
```

```
// однако можно задать и так:
// f.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

// Добавляем на панель окна нередактируемый компонент с текстом.
// f.getContentPane().add (new JLabel("Hello, World!")); - старый стиль
 f.add(new JLabel("Hello World"));

// pack() «упаковывает» окно до оптимального размера
// всех расположенных в нем компонентов.
 f.pack();
// Показать окно
 f.setVisible(true);
}}
```

Базовым классом всей библиотеки визуальных компонентов Swing является JComponent. Это суперкласс других визуальных компонентов. Он является абстрактным классом, поэтому в действительности вы не можете создать JComponent, но он содержит сотни функций, которые каждый компонент Swing может использовать как результат иерархии классов. Класс JComponent обеспечивает инфраструктуру окрашивания для всех компонентов. Он знает, как обрабатывать все нажатия клавиш на клавиатуре. Подклассы JComponent должны только прослушивать определенные клавиши. Класс JComponent также содержит метод add(), который позволяет добавить другие объекты класса JComponent, так можно добавить любой Swing-компонент к любому другому для создания вложенных компонентов (например, JPanel, содержащую JButton, или даже более причудливые комбинации, например, JMenu, содержащее JButton).

Самым простым и в то же время основным визуальным компонентом в библиотеке Swing является JLabel, или «метка». К методам этого класса относится установка текста, изображения, выравнивания и других компонентов, которые описывает метка:

1) get/setText() позволяет получить или установить текст в метке;
2) get/setIcon() – изображение в метке;
3) get/setHorizontalAlignment – горизонтальную позицию текста;
4) get/setVerticalAlignment() – вертикальную позицию текста;
5) get/setDisplayedMnemonic() – мнемонику (подчеркнутый символ) для метки;
6) get/setLabelFor() – компонент, к которому присоединена данная метка; когда пользователь нажимает комбинацию клавиш Alt + мнемоника, фокус перемещается на указанный компонент.

Основным активным компонентом в Swing является JButton. Методы, используемые для изменения свойств JButton, аналогичны методам JLabel (вы обнаружите, что они аналогичны для большинства Swing-компонентов). Они управляют текстом, изображениями и ориентацией:

1) get/setText() позволяет получить или установить текст в кнопке;

2) get/setIcon() – изображение в кнопке;

3) get/setHorizontalAlignment() – горизонтальную позицию текста;

4) get/setVerticalAlignment() – вертикальную позицию текста;

5) get/setDisplayedMnenomic() – мнемонику (подчеркнутый символ), которая в комбинации с кнопкой Alt вызывает нажатие кнопки.

Класс JFrame является контейнером, позволяющим добавлять к себе другие компоненты для их организации и предоставления пользователю. JFrame выступает в качестве моста между независимыми от конкретной операционной системы Swing-частями и реальной операционной системой, на которой они работают. JFrame регистрируется как окно и таким образом получает многие свойства окна операционной системы: минимизация/максимизация, изменение размеров и перемещение. Хотя при выполнении лабораторной работы достаточно считать JFrame палитрой, на которой вы размещаете компоненты. Перечислим некоторые из методов, которые вы можете вызвать в JFrame для изменения его свойств:

1) get/setTitle() позволяет получить или установить заголовок фрейма;

2) get/setState() – состояние фрейма (минимизировать, максимизировать и т. д.);

3) is/setVisible() – видимость фрейма, другими словами, отображение на экране;

4) get/setLocation() – месторасположение в окне, где фрейм должен появиться;

5) get/setSize() – размер фрейма;

6) add() позволяет добавить компоненты к фрейму.

При построении визуальных приложений в Java нельзя просто случайно разместить их на экране и ожидать от них немедленной работы. Компоненты необходимо разместить в определенные места, реагировать на взаимодействие с ними, обновлять их на основе этого взаимодействия и заполнять данными. Для эффективной работы с визуальными компонентами необходима установка следующих трех архитектурных составляющих Swing:

- схем (layout). Swing содержит множество схем, которые представляют собой классы, управляющие размещением компонентов в приложении и тем, что должно произойти с ними при изменении размеров окна приложения или при удалении или добавлении компонентов;

- событий (event). Программа должна реагировать на нажатия клавиатуры, кнопки мыши и на все остальное, что пользователь может сделать;

- моделей (model). Для более продвинутых компонентов (списки, таблицы, деревья) и даже для некоторых более простых, например, JComboBox, модели – это самый эффективный способ работы с данными. Они удаляют большую часть работы по обработке данных из самого компонента (вспомните MVC) и предоставляют оболочку для общих объектных классов данных (например Vector и ArrayList).

Особое внимание в связи с необходимостью изображения динамических сцен на визуальных компонентах необходимо уделить классу Graphics 2D.

Для создания интерфейса для нашего приложения нам необходимо:

1) создать окно программы;

2) добавить несколько закладок для разделения работы с различными объектами;

3) добавить графические элементы для добавления, редактирования и удаления объектов;

4) редактировать у связанных объектов ссылки на главные объекты при помощи элементов ComboBox.

Задания для самостоятельной работы

Разработать программный интерфейс API для дальнейших действий с ранее разработанными методами обработки данных модели. В состав интерфейса должны входить методы сохранения и чтения данных, получения списков объектов, отсортированных и отфильтрованных согласно передаваемому критерию, бизнес-методы модели согласно полученному варианту по теме 1.

Разработать пользовательский интерфейс с помощью библиотеки Swing. Для этого необходимо использовать библиотеку готовых графических элементов и Layout для компоновки графических форм для работы с данными.

Тема 7. Сетевые средства Java

Цель: исследовать стандарты обмена данными в компьютерных сетях, сетевые средства платформы Java.

Теория и примеры выполнения задания

Когда число компьютеров в учреждении превышает десяток, тогда в них вставляются сетевые карты, протягиваются кабели и компьютеры объединяются в сеть. Сначала все компьютеры в сети равноправны, они делают одно и то же – это одноранговая (peer-to-peer) сеть. Далее покупается компьютер с большими и быстрыми жесткими дисками, и все файлы учреждения начинают храниться на данных дисках – этот компьютер становится файл-сервером, предоставляющим услуги хранения, поиска, архивирования файлов. Затем покупается дорогой и быстрый принтер. Компьютер, связанный с ним, становится принт-сервером, предоставляющим услуги печати. Далее появляются графический сервер, вычислительный сервер, сервер базы данных. Остальные компьютеры становятся клиентами этих серверов. Такая архитектура сети называется архитектурой клиент-сервер (client-server).

Язык Java делает сетевое программирование простым благодаря наличию специальных средств и классов. Рассмотрим некоторые виды сетевых приложений. Интернет-приложения включают web-браузер, e-mail, сетевые новости, передачу файлов и telnet. Основные используемые протоколы – TCP и IP. Приложения клиент-сервер используют компьютер, выполняющий специальную программу-сервер, которая предоставляет услуги другим программам – клиентам. Клиент – это программа, получающая услуги от сервера. Клиент-серверные приложения основаны на использовании, в первую очередь, прикладных протоколов стека TCP/IP, таких как:

1) HTTP – Hypertext Transfer Protocol (WWW);
2) NNTP – Network News Transfer Protocol (группы новостей);
3) SMTP – Simple Mail Transfer Protocol (рассылка почты);
4) POP3 – Post Office Protocol (получение почты с сервера);
5) FTP – File Transfer Protocol (протокол передачи файлов);
6) TELNET – удаленное управление компьютерами.

Каждый компьютер, работающий по протоколам стека TCP/IP имеет уникальный сетевой адрес. IP-адрес – это 32-битовое число, обычно записываемое как четыре числа, разделенные точками, каждое из которых изменяется от 0 до 255. IP-адрес может быть временным и выделяться динамически для каждого подключения или быть постоянным, как для сервера. Обычно при подключении к компьютеру вместо числового IP-адреса используются символьные имена (например – www.example.com), называемые доменными именами. Специальная программа DNS (Domain Name Sever) преобразует имя домена в числовой IP-адрес. Получить IP-адрес в программе можно с помощью объекта класса InetAddress из пакета Java.net.

```
import java.net.*;
public class MyLocal {
  public static void main(String[] args) {
    InetAddress myIP = null;
    try {
      myIP = InetAddress.getLocalHost();
    } catch (UnknownHostException e) {
    System. out.println( " ошибка доступа ->" + e);
  }
  System. out.println( " Мой IP ->" + myIP);
}
```

Следующая программа демонстрирует, как получить IP-адрес из имени домена с помощью сервера имен доменов (DNS), к которому обращается метод getByName().

```
import java.net.*;
public class IPfromDNS {

public static void main(String[] args) {
        InetAddress omgtu = null;
        try {
                omgtu = InetAddress.getByName("omgtu.ru");
                }
        catch (UnknownHostException e) {
                System.out.println( " ошибка доступа ->" + e);
                }
        System.out.println( "IP- адрес ->" + omgtu );
        }
}
```

Будет выведено: IP-адрес –>omgtu.ru/195.69.204.35

Сокеты – это сетевые разъемы, через которые осуществляются двунаправленные поточные соединения между компьютерами. Сокет определяется номером порта и IP-адресом. При этом IP-адрес используется для идентификации компьютера, номер порта – для идентификации процесса, работающего на компьютере. Когда одно приложение знает сокет другого, создается сокетное соединение. Клиент пытается соединиться с сервером, инициализируя сокетное соединение. Соединение с помощью сокетов устанавливается следующим образом.

1) сервер создает сокет, прослушивающий порт сервера;

2) клиент тоже создает сокет, через который связывается с сервером, сервер начинает устанавливать (accept) связь с клиентом;

3) устанавливая связь, сервер создает новый сокет, прослушивающий порт с другим, новым номером, и сообщает этот номер клиенту;

4) клиент посылает запрос на сервер через порт с новым номером.

После этого соединение становится совершенно симметричным – два сокета обмениваются информацией, а сервер через старый сокет продолжает прослушивать прежний порт, ожидая следующего клиента.

В Java сокет – это объект класса Socket из пакета Java.io. В классе шесть конструкторов, в которые разными способами заносится адрес хоста и номер порта.

Рассмотрим пример получения файла с сервера по максимально упрощенному протоколу HTTP.

1. Клиент посылает серверу запрос на получение файла строкой POST filename HTTP/1.l\n\n, где filename – строка с путем к файлу на сервере.

2. Сервер анализирует строку, отыскивает файл с именем filename и возвращает его клиенту. Если имя файла filename заканчивается наклонной чертой /, то сервер понимает его как имя каталога и возвращает файл index.html, находящийся в этом каталоге.

3. Перед содержимым файла сервер посылает строку вида HTTP/1.1 code OK\n\n, где code – это код ответа, одно из чисел которого будет иметь следующие цифры: 200 – запрос удовлетворен, файл посылается; 400 – запрос не понят; 404 – файл не найден.

4. Сервер закрывает сокет и продолжает слушать порт, ожидая следующего запроса.

5. Клиент выводит содержимое полученного файла в стандартный вывод System, out или выводит код сообщения сервера в стандартный вывод сообщений System, err.

6. Клиент закрывает сокет, завершая связь.

Закрытие потоков ввода/вывода вызывает закрытие сокета, а закрытие сокета закрывает потоки.

Сокетное соединение с сервером создается с помощью объекта класса Socket. При этом указывается IP-адрес сервера и номер порта (80 для HTTP). Если указано имя домена, то Java преобразует его с помощью DNS-сервера к IP-адресу

```
try {
  Socket socket = new Socket("localhost", 8030);
} catch (IOException e) {
  System. out.println( " ошибка : " + e);
}.
```

Сервер ожидает сообщения клиента и должен быть запущен с указанием определенного порта. Объект класса ServerSocket создается с указанием конструктору номера порта и ожидает сообщения клиента с помощью метода accept(), который возвращает сокет клиента:

```
Socket socket = null ;
try {
  ServerSocket server = new ServerSocket(8030);
  socket = server.accept();
} catch (IOException e) {
  System. out.println( " ошибка : " + e);
}.
```

Клиент и сервер после установления сокетного соединения могут получать данные из потока ввода и записывать данные в поток вывода с помощью методов getInputStream() и getOutputStream() или к PrintStream, для того чтобы программа могла трактовать поток как выходные файлы.

В следующем примере для посылки клиенту строки «привет!» сервер вызывает метод getOutputStream() класса Socket. Клиент получает данные от сервера с помощью метода getInputStream(). Для разъединения клиента и сервера после завершения работы сокет закрывается с помощью метода close() класса Socket. В следующем примере сервер посылает клиенту строку «привет!», после чего разрывает связь.

```
// передача клиенту строки : MyServerSocket. java
import java.io.*;
import java.net.*;
public class MyServerSocket {
  public static void main(String[] args) throws Exception {
    Socket s = null;
    try { // посылка строки клиенту
      ServerSocket server = new ServerSocket(8030);
      s = server.accept();
      PrintStream ps = new PrintStream(s.getOutputStream());
      ps.println( " привет !" );
      ps.flush();
      s.close(); // разрыв соединения
    } catch (IOException e) {
      System. out.println( " ошибка : " + e);
    }
  }
}
/* получение клиентом строки : MyClientSocket. java */
import java.io.*;
import java.net.*;
public class MyClientSocket {
  public static void main(String[] args) {
    Socket socket = null;
    try {// получение строки клиентом
      socket = new Socket( " имя _ компьютера " , 8030);
      BufferedReader dis = new BufferedReader(new InputStreamReader(
                                                socket.getInputStream()));
      String msg = dis.readLine();
      System.out.println(msg);
    } catch (IOException e) {
```

```
            System. out.println( " ошибка : " + e);
        }
      }
    }
```

Аналогично клиент может послать данные серверу через поток вывода с помощью метода getOutputStream(), а сервер может получать данные с помощью метода getInputStream().

Если необходимо протестировать подобный пример на одном компьютере, можно выступать одновременно в роли клиента и сервера, используя статические методы getLocalHost() класса InetAddress для получения динамического IP-адреса компьютера, который выделяется при входе в Интернет.

Для того чтобы с приложением работало несколько людей, необходимо организовать работу приложения на основе архитектуры клиент-сервер, а для этого требуется передача информации по сети. В прошлой лабораторной работе мы сделали графический интерфейс, позволяющий редактировать данные модели. Целью данной работы является создание клиент-серверной архитектуры приложения. Необходимо реализовать две основные операции: чтение данных и сохранение данных. Код, который записывает данные в поток, у нас уже есть, нам необходимо реализовать запросы к серверной части. Серверная часть – это приложение, которое открывает Socket. Клиентская часть выполняет соединение с сокетом сервера, выполняет запрос и читает ответ сервера. Для выполнения этих операций необходимо произвести следующие действия:

1) создать приложение «Сервер», которое создаст серверный Socket для приема данных на определенном порту и будет ждать входящее соединение;

2) доработать наше приложение для того, чтобы по команде сохранения объекта выполнялся запрос к приложению «Сервер». Перед передачей данных нужен идентификатор команды которая должна быть выполнена;

3) после выполнения получения запроса серверная часть должна выполнить команду «Отправить ответ». Ответом может быть код ошибки или сообщение об ошибке;

4) клиентская часть после получения ответа в случае ошибки должна вывести пользователю сообщение об ошибке или подтвердить, что данные сохранились правильно.

Задание для самостоятельной работы

Разработать взаимодействие модулей приложения по сети, используя стандарты обмена данными по сети TCP/IP. Для организации обмена данными между модулями приложения предусмотреть создание и настройку сокетов (адресов и портов). Реализовать обмен данными объектов модели по сети (использовать ранее разработанную запись данных в поток).

Тема 8. Потоки выполнения

Цель: исследовать средства параллельной обработки данных платформы Java.

Теория и примеры выполнения задания

Основное понятие в современных операционных системах – процесс (process). Можно понимать под процессом выполняющуюся (runnable) программу, но надо помнить о том, что у процесса есть несколько состояний. Процесс может в любой момент перейти к выполнению машинного кода другой программы, а также «заснуть» (sleep) на некоторое время, приостановив выполнение программы. Он может быть выгружен на диск. Количество состояний процесса и их особенности зависят от операционной системы.

Все современные операционные системы многозадачные (multitasking), они запускают и выполняют сразу несколько процессов. Одновременно может работать браузер, текстовый редактор, музыкальный проигрыватель. На экране дисплея открываются несколько окон, каждое из которых связано со своим работающим процессом.

Реализацию многопоточной архитектуры проще всего представить себе для системы, в которой есть несколько центральных вычислительных процессоров. В этом случае для каждого из них можно выделить задачу, которую он будет выполнять. В результате несколько задач будут обслуживаться одновременно.

Однако возникает вопрос – каким же тогда образом обеспечивается многопоточность в системах с одним центральным процессором, который, в принципе, выполняет лишь одно вычисление в один момент времени? В таких системах применяется процедура квантования времени (time-slicing). Время разделяется на небольшие интервалы. Перед началом каждого интервала принимается решение, какой именно поток выполнения будет отрабатываться на протяжении этого кванта времени. За счет частого переключения между задачами эмулируется многопоточная архитектура.

На самом деле, как правило, и для многопроцессорных систем применяется процедура квантования времени. Дело в том, что даже в мощных серверах приложений процессоров не так много (редко бывает больше десяти), а потоков исполнения запускается гораздо больше. Например, операционная система Windows без единого запущенного приложения инициализирует десятки, а то и сотни потоков. Квантование времени позволяет упростить управление выполнением задач на всех процессорах.

Среди начинающих программистов бытует мнение, что многопоточные программы работают быстрее. Рассмотрев способ реализации многопоточности, можно утверждать, что такие программы работают на самом деле медленнее. Действительно, для переключения между задачами на каждом интервале требуется дополнительное время, а ведь они (переключения) происходят довольно часто. Если бы процессор, не отвлекаясь, выполнял

задачи последовательно, одну за другой, он завершил бы их заметно быстрее. Стало быть, преимущества заключаются не в этом.

Первый тип приложений, который выигрывает от поддержки многопоточности, предназначен для задач, где действительно требуется выполнять несколько действий одновременно. Например, будет вполне обоснованно ожидать, что сервер общего пользования станет обслуживать несколько клиентов одновременно. Можно легко представить себе пример из сферы обслуживания, когда имеется несколько потоков клиентов и желательно обслуживать их все одновременно.

Другой пример – активные игры или подобные приложения. Необходимо одновременно опрашивать клавиатуру и другие устройства ввода, чтобы реагировать на действия пользователя. В то же время необходимо рассчитывать и перерисовывать изменяющееся состояние игрового поля.

Понятно, что в случае отсутствия поддержки многопоточности для реализации подобных приложений потребовалось бы реализовывать квантование времени вручную. Условно говоря, одну секунду проверять состояние клавиатуры, а следующую – пересчитывать и перерисовывать игровое поле. Если сравнить две реализации time-slicing, одну – на низком уровне, выполненную средствами, как правило, операционной системы, другую – выполняемую вручную, на языке высокого уровня, мало подходящего для таких задач, то становится понятным первое и, возможно, главное преимущество многопоточности: она обеспечивает наиболее эффективную реализацию процедуры квантования времени, существенно облегчая и укорачивая процесс разработки приложения. Код переключения между задачами на Java выглядел бы куда более громоздко, чем независимое описание действий для каждого потока.

Следующее преимущество проистекает из того, что компьютер состоит не только из одного или нескольких процессоров. Вычислительное устройство – лишь один из ресурсов, необходимых для выполнения задач. Всегда есть оперативная память, дисковая подсистема, сетевые подключения, периферия и т. д. Предположим, пользователю требуется распечатать большой документ и скачать большой файл из сети. Очевидно, что обе задачи требуют совсем незначительного участия процессора, а основные необходимые ресурсы, которые будут задействованы на пределе возможностей, у них разные – сетевое подключение и принтер. Значит, если выполнять задачи одновременно, то замедление от организации квантования времени будет незначительным, процессор легко справится с обслуживанием обеих задач. В то же время, если каждая задача по отдельности занимала, скажем, два часа, то вполне вероятно, что и при одновременном исполнении потребуется не более тех же двух часов, а сделано при этом будет гораздо больше.

Если же задачи в основном загружает процессор (например математические расчеты), то их одновременное исполнение займет в лучшем случае столько же времени, что и последовательное, а то и больше.

Третье преимущество появляется из-за возможности более гибко управлять выполнением задач. Предположим, пользователь системы, не поддерживающей многопоточность, решил скачать большой файл из сети или произвести сложное вычисление, что занимает, скажем, два часа. Запустив задачу на выполнение, он может внезапно обнаружить, что ему нужен не этот, а какой-нибудь другой файл (или вычисление с другими начальными параметрами). Однако если приложение занимается только работой с сетью (вычислениями) и не реагирует на действия пользователя (не обрабатываются данные с устройств ввода, таких как клавиатура или мышь), то он не сможет быстро исправить ошибку. Получается, что процессор выполняет большее количество вычислений, но при этом приносит гораздо меньше пользы.

Процедура квантования времени поддерживает приоритеты (priority) задач. В Java приоритет представляется целым числом. Чем больше число, тем выше приоритет. Строгих правил работы с приоритетами нет, каждая реализация может вести себя по-разному на разных платформах. Однако есть общее правило – поток с более высоким приоритетом будет получать большее количество квантов времени на исполнение и таким образом сможет быстрее выполнять свои действия и реагировать на поступающие данные.

В описанном примере представляется разумным запустить дополнительный поток, отвечающий за взаимодействие с пользователем. Ему можно поставить высокий приоритет, т. к. в случае бездействия пользователя этот поток практически не будет занимать ресурсы машины. В случае же активности пользователя следует, как можно быстрее произвести необходимые действия, чтобы обеспечить максимальную эффективность работы пользователя.

Рассмотрим еще одно свойство потоков. Раньше, когда рассматривались однопоточные приложения, завершение вычислений однозначно приводило к завершению выполнения программы. Теперь же приложение должно работать до тех пор, пока есть хоть один действующий поток исполнения. В то же время часто бывают нужны обслуживающие потоки, которые не имеют никакого смысла, если они остаются в системе одни. Например, автоматический сборщик мусора в Java запускается в виде фонового (низкоприоритетного) процесса. Его задача – отслеживать объекты, которые уже не используются другими потоками, и затем уничтожать их, освобождая оперативную память. Понятно, что работа одного потока garbage collector не имеет никакого смысла.

Такие обслуживающие потоки называют демонами (daemon), это свойство можно установить любому потоку. В итоге приложение выполняется до тех пор, пока есть хотя бы один поток недемон.

Поток выполнения в Java представляется экземпляром класса Thread. Для того, чтобы написать свой поток исполнения, необходимо наследоваться от этого класса и переопределить метод run(). Например,

```
public class MyThread extends Thread {
  public void run() {
```

```
    // некоторое долгое действие, вычисление
    long sum=0;
    for (int i=0; i<1000; i++) {
      sum+=i;
    }
    System.out.println(sum);
  }
}
```

Метод run() содержит действия, которые должны выполняться в новом потоке исполнения. Чтобы запустить его, необходимо создать экземпляр класса-наследника и вызвать унаследованный метод start(), который сообщает виртуальной машине, что требуется запустить новый поток исполнения и начать выполнять в нем метод run().

```
MyThread t = new MyThread();
t.start();
```

В результате чего на консоли появится результат:

```
499500
```

Когда метод run() завершен (в частности, встретилось выражение return), поток выполнения останавливается. Однако ничто не препятствует записи бесконечного цикла в этом методе. В результате поток не прервет своего исполнения и будет остановлен только при завершении работы всего приложения.

Описанный подход имеет один недостаток. Поскольку в Java множественное наследование отсутствует, требование наследоваться от Thread может привести к конфликту. Если еще раз посмотреть на приведенный выше пример, станет понятно, что наследование производилось только с целью переопределения метода run(). Поэтому предлагается более простой способ создать свой поток исполнения. Достаточно реализовать интерфейс Runnable, в котором объявлен только один метод – уже знакомый void run(). Запишем пример, приведенный выше, с помощью этого интерфейса:

```
public class MyRunnable implements Runnable {
  public void run() {
    // некоторое долгое действие, вычисление
    long sum=0;
    for (int i=0; i<1000; i++) {
      sum+=i;
    }
    System.out.println(sum);
  }
}
```

Также незначительно меняется процедура запуска потока:

```
Runnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
```

Если раньше объект, представляющий сам поток выполнения, и объект с методом run(), реализующим необходимую функциональность, были объединены в одном экземпляре класса MyThread, то теперь они разделены. Какой из двух подходов удобней, решается в каждом конкретном случае. Runnable не является полной заменой классу Thread, поскольку создание и запуск самого потока исполнения возможно только через метод Thread.start().

Рассмотрим, как в Java можно назначать потокам приоритеты. Для этого в классе Thread существуют методы getPriority() и setPriority(), а также объявлены три константы:

MIN_PRIORITY
MAX_PRIORITY
NORM_PRIORITY

Из названия понятно, что их значения описывают минимальное, максимальное и нормальное (по умолчанию) значения приоритета.

Рассмотрим следующий пример:

```
public class ThreadTest implements Runnable {
  public void run() {
    double calc;
    for (int i=0; i<50000; i++) {
      calc=Math.sin(i*i);
      if (i%10000==0) {
        System.out.println(getName()+
          " counts " + i/10000);
      }
    }
  }

  public String getName() {
     return Thread.currentThread().getName();
  }

  public static void main(String s[]) {
    // Подготовка потоков
    Thread t[] = new Thread[3];
    for (int i=0; i<t.length; i++) {
      t[i]=new Thread(new ThreadTest(),
                  "Thread "+i);
    }
    // Запуск потоков
    for (int i=0; i<t.length; i++) {
      t[i].start();
```

```
            System.out.println(t[i].getName()+
                               " started");
        }
    }
}
```

Обратите внимание, что конструктору класса Thread передается два параметра. К реализации Runnable добавляется строка. Это имя потока, которое используется только для упрощения его идентификации. Имена нескольких потоков могут совпадать. Если его не задать, то Java генерирует простую строку вида «Thread-» и номер потока (вычисляется простым счетчиком). Именно это имя возвращается методом getName(). Его можно сменить с помощью метода setName().

Статический метод currentThread позволяет в любом месте кода получить ссылку на объект класса Thread, представляющий текущий поток исполнения. Результат работы такой программы будет иметь следующий вид:

```
Thread 0 started
Thread 1 started
Thread 2 started
Thread 0 counts 0
Thread 1 counts 0
Thread 2 counts 0
Thread 0 counts 1
Thread 1 counts 1
Thread 2 counts 1
Thread 0 counts 2
Thread 2 counts 2
Thread 1 counts 2
Thread 2 counts 3
Thread 0 counts 3
Thread 1 counts 3
Thread 2 counts 4
Thread 0 counts 4
Thread 1 counts 4
```

Мы видим, что все три потока были запущены один за другим и начали проводить вычисления. Видно также, что потоки исполняются без определенного порядка, случайным образом. Тем не менее в среднем они движутся с одной скоростью, никто не отстает и не догоняет.

Введем в программу работу с приоритетами, расставим разные значения для разных потоков и посмотрим, как это скажется на выполнении. Изменяется только метод main().

```
public static void main(String s[]) {
// Подготовка потоков
   Thread t[] = new Thread[3];
   for (int i=0; i<t.length; i++) {
      t[i]=new Thread(new ThreadTest(),
                  "Thread "+i);
      t[i].setPriority(Thread.MIN_PRIORITY +
      (Thread.MAX_PRIORITY -
      Thread.MIN_PRIORITY)/t.length*i);
   }

// Запуск потоков
   for (int i=0; i<t.length; i++) {
      t[i].start();
      System.out.println(t[i].getName()+
                           " started");
   }
}
```

Формула вычисления приоритетов позволяет равномерно распределить все допустимые значения для всех запускаемых потоков. На самом деле константа минимального приоритета имеет значение «1», максимального – «10», нормального – «5». Так что в простых программах можно явно пользоваться этими величинами и указывать в качестве, например, пониженного приоритета значение «3». Результатом работы будет следующий вид потоков.

```
Thread 0 started
Thread 1 started
Thread 2 started
Thread 2 counts 0
Thread 2 counts 1
Thread 2 counts 2
Thread 2 counts 3
Thread 2 counts 4
Thread 0 counts 0
Thread 1 counts 0
Thread 1 counts 1
Thread 1 counts 2
Thread 1 counts 3
Thread 1 counts 4
Thread 0 counts 1
Thread 0 counts 2
Thread 0 counts 3
Thread 0 counts 4
```

Потоки, как и раньше, стартуют последовательно. Но затем мы видим, что чем выше приоритет, тем быстрее отрабатывает поток. Тем не менее весьма показательно, что поток с минимальным приоритетом (Thread 0) все же получил возможность выполнить одно действие раньше, чем отработал поток с более высоким приоритетом (Thread 1). Это говорит о том, что приоритеты не делают систему однопоточной, выполняющей единовременно лишь один поток с наивысшим приоритетом. Напротив, приоритеты позволяют одновременно работать над несколькими задачами с учетом их важности.

Если увеличить параметры метода (выполнять 500 000 вычислений, а не 50 000, и выводить результаты каждого 1000-го вычисления, а не 10 000-го), то можно будет наглядно увидеть, что все три потока имеют возможность выполнять свои действия одновременно, просто более высокий приоритет позволяет выполнять их чаще.

Наконец, перейдем к рассмотрению трех методов класса Object, завершая описание механизмов поддержки многопоточности в Java. Каждый объект в Java имеет не только блокировку для synchronized-блоков и методов, но и так называемый wait-set – набор потоков исполнения. Любой поток может вызвать метод wait() любого объекта и таким образом попасть в его wait-set. При этом выполнение такого потока приостанавливается до тех пор, пока другой поток не вызовет у этого же объекта метод notifyAll(), который пробуждает все потоки из wait-set. Метод notify() пробуждает один случайно выбранный поток из данного набора.

Однако применение этих методов связано с одним важным ограничением: любой из них может быть вызван потоком у объекта только после установления блокировки на этот объект, т. е. либо внутри synchronized-блока со ссылкой на этот объект (в качестве аргумента), либо обращения к методам должны быть в синхронизированных методах класса самого объекта. Рассмотрим пример:

```
public class WaitThread implements Runnable {
    private Object shared;

    public WaitThread(Object o) {
      shared=o;
}

public void run() {
  synchronized (shared) {
    try {
      shared.wait();
    } catch (InterruptedException e) {}
    System.out.println("after wait");
  }
}
```

```
public static void main(String s[]) {
  Object o = new Object();
  WaitThread w = new WaitThread(o);
  new Thread(w).start();
  try {
    Thread.sleep(100);
  } catch (InterruptedException e) {}
  System.out.println("before notify");
  synchronized (o) {
    o.notifyAll();
  }
  }
}
```

Результатом программы будет:

```
before notify
after wait
```

Обратите внимание, что метод wait(), как и sleep(), требует обработки InterruptedException, т. е. его выполнение также можно прервать методом interrupt(). В заключение рассмотрим более сложный пример для трех потоков:

```
public class ThreadTest implements Runnable {
    final static private Object shared=new Object();
    private int type;
    public ThreadTest(int i) {
      type=i;
    }

public void run() {
  if (type==1 || type==2) {
    synchronized (shared) {
      try {
        shared.wait();
      } catch (InterruptedException e) {}
      System.out.println("Thread "+type+" after wait()");
  }
} else {
```

```
    synchronized (shared) {
      shared.notifyAll();
      System.out.println("Thread "+type+" after notifyAll()");
      }
    }
  }

  public static void main(String s[]) {
    ThreadTest w1 = new ThreadTest(1);
    new Thread(w1).start();
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {}
    ThreadTest w2 = new ThreadTest(2);
    new Thread(w2).start();
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {}
    ThreadTest w3 = new ThreadTest(3);
    new Thread(w3).start();
    }
  }
```

Результатом работы программы будет:

```
Thread 3 after notifyAll()
Thread 1 after wait()
Thread 2 after wait()
```

Рассмотрим, что произошло. Во-первых, был запущен поток 1, который тут же вызвал метод wait() и приостановил свое выполнение. Затем то же самое произошло с потоком 2. Далее начинает выполняться поток 3.

Сразу обращает на себя внимание следующий факт. Еще поток 1 вошел в synchronized-блок, а стало быть установил блокировку на объект shared. Но, судя по результатам, это не помешало и потоку 2 зайти в synchronized-блок, а затем и потоку 3. Причем для последнего это просто необходимо, иначе как можно «разбудить» потоки 1 и 2? Можно сделать вывод, что потоки, прежде чем приостановить выполнение после вызова метода wait(), отпускают все занятые блокировки.

Вызываем метод notifyAll(). Как уже было сказано, все потоки из wait-set возобновляют свою работу. Однако, чтобы корректно продолжить исполнение, необходимо вернуть блокировку на объект, ведь следующая команда также находится внутри synchronized-блока.

Получается, что даже после вызова notifyAll() все потоки не могут сразу возобновить работу. Лишь один из них сможет вернуть себе блокировку и продолжить работу. Когда он покинет свой synchronized-блок и отпустит объект, второй поток возобновит свою работу и т. д. Если по какой-то причине объект так и не будет освобожден, поток так никогда и не выйдет из метода wait(), даже если будет вызван метод notifyAll(). В рассмотренном примере потоки один за другим смогли возобновить свою работу.

Кроме того, определен метод wait() с параметром, который задает период тайм-аута, по истечении которого поток сам попытается возобновить свою работу. Но начать ему придется все равно с повторного получения блокировки.

Для нашего приложения выполнение открытки необходимо потому, что в случае если два приложения клиента обращаются одновременно, сервер не способен обрабатывать их запросы параллельно. Одному из клиентов будет отправлена ошибка. Нам необходимо будет организовать работу серверной части так, чтобы подключение и запросы к серверу выполнялись параллельно. Сложностью является то, что место где хранятся данные только одно. Для решения проблемы синхронизации нескольких потоков используется алгоритм семафора. Таким образом, нам необходимо будет создавать новые потоки и синхронизировать их. Для этого надо выполнить следующие действия:

1) создать поток, который будет отслеживать все входящие подключения к серверному сокету. Этот поток будет создавать новые потоки обработки;

2) класс для сохранения данных на серверной части объявить как Syncronized, что тем самым предотвратит совместное использование класса несколькими потоками одновременно, создаст очередь и гарантирует последовательное использование класса только одним потоком обработки.

Синхронизированный блок требуется минимизировать, потому что чем больше работы будет выполняться синхронизировано, тем дольше будет выполняться обработка каждой команды, а значит, медленнее будет работать серверное приложение.

Задание для самостоятельной работы

Реализовать параллельную обработку запросов к серверной части приложения для работы с несколькими клиентами по сети. Использовать ранее полученную функциональность. Реализовать сервис рассылки сообщений согласно полученному варианту заданий по теме 1. Использовать средства параллельной обработки данных платформы Java, класс Thread и интерфейс Runable.

Тема 9. Работа с БД через JDBC

Цель: исследовать технологии JDBC для работы с БД платформы Java.

Теория и примеры выполнения задания

Большинство информации хранится не в файлах, а в базах данных. Приложение должно уметь связываться с базой данных для получения из нее информации или для помещения информации в базу данных. Дело здесь осложняется тем, что СУБД (системы управления базами данных) сильно отличаются друг от друга и совершенно по-разному управляют базами данных. Каждая СУБД предоставляет свой набор функций для доступа к базам данных, и приходится для каждой СУБД писать свое приложение.

Фирма SUN разработала набор интерфейсов и классов, названный JDBC, предназначенный для работы с базами данных. Эти интерфейсы и классы составили пакет Java.sqi, входящий в J2SDK Standard Edition, и его расширение Javax.sql, входящее в J2SDK Enterprise Edition.

Подключение Java-программы к реляционной СУБД с помощью JDBC выполняется в три этапа (рисунок 11):

1) установка связи между Java-программой и диспетчером базы данных;

2) передача SQL-команды в базу данных с помощью объекта Statement;

3) чтение полученных результатов из базы данных и использование их в программе.

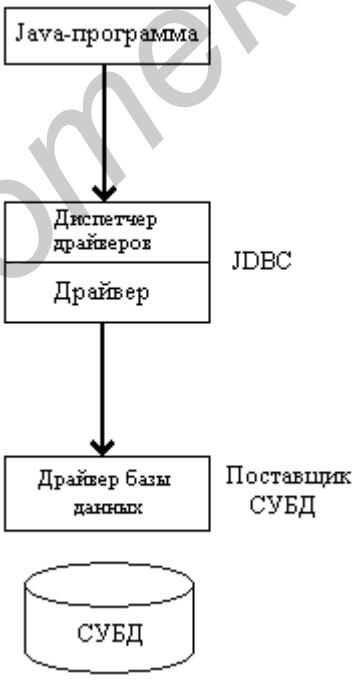

Рисунок 11 – Схема подключения Java-программы к базе данных

Пакет JDBC предназначен для работы с разнообразными диспетчерами СУБД от различных разработчиков. Для подключения к базе данных среда выполнения Java должна загрузить соответствующий драйвер указанной базы

данных. Загрузка и выгрузка таких драйверов осуществляется с помощью класса DriverManager.

Класс DriverManager имеет структуру данных, которая содержит как сами драйверы в виде объектов Driver, так и информацию о них.

Драйверы JDBC обычно создаются поставщиками СУБД. Их работа заключается в обработке JDBC-подключений и команд, поступающих от Java-приложения, и в генерации машинно-зависимых вызовов по отношению к базе данных.

Далеко не все поставщики СУБД предлагают драйверы JDBC, но, как правило, ими всегда поставляются драйверы ODBC (Open Database Connectivity), которые удовлетворяют стандарту Microsoft. При работе с СУБД на платформе Windows поставщик используемой СУБД почти всегда предлагает собственный драйвер ODBC. Поэтому проблем с подключением Java-приложения к базе данных в среде Windows обычно не возникает, чего, к сожалению, нельзя сказать о других платформах.

Загрузка драйвера может производиться как из программы, так и из командной строки. Для загрузки драйвера-моста JDBC-ODBC из командной строки необходимо ввести следующую команду:

```
Java -Djdbc.drivers=sun.odbc.JdbcOdbcDriver MyApplication
```

Для загрузки драйвера-моста JDBC-ODBC из программы необходимо ввести

```
try{
  Class theDriver = sun.odbc.JdbcOdbcDriver.class;
} catch(ClassNotFoundException e){
  System.err.println("Драйвер JDBC/ODBC не найден");
}
```

После регистрации драйвера с помощью диспетчера драйверов его можно применять для подключения к базе данных. Для этого диспетчеру следует сообщить о создании нового подключения. В ответ на это диспетчер драйверов вызовет соответствующий драйвер и возвратит ссылку на установленное подключение. Для создания подключения необходимо указать место расположения базы данных, а также (как правило, для большинства баз данных) учетное имя и пароль, как показано ниже.

```
Connection myConnection = DriverManager.getConnection(
"jdbc:odbc:mydataSource",
"username",
"password");
```

После получения запроса getConnection() диспетчер драйверов анализирует значение адреса URL для JDBC и, в свою очередь, передает его каждому зарегистрированному драйверу. Затем подключение будет установлено с помощью того драйвера, который первым опознает данный адрес URL для JDBC и сообщит о готовности к подключению. Если ни один из драйверов не сможет его опознать, диспетчер драйверов инициирует обработку исключительной ситуации SQLException с выдачей сообщения об отсутствии подходящего драйвера (No suitable driver).

Объект Statement предназначен для хранения SQL-команд. При пересылке объекта Statement базе данных с помощью установленного подключения СУБД запустит заданную SQL-команду и возвратит результат ее выполнения в виде объекта ResultSet. Для извлечения результатов запроса в виде объекта ResultSet следует использовать следующий код:

```
ResultSet theSet = theStatement.executeQuery("SELECT * FROM *");
```

На этом практическом занятии нам необходимо создать структуру таблиц, подключиться к базе данных, добавить операции сохранения редактирования и удаления данных. Для создания структуры таблиц используется язык Data Definition Language (DDL), часть языка SQL.

Для подключения к базе данных по технологии JDBC требуется указать четыре параметра: строку подключения, логин, пароль и названия драйвера. В предыдущей работе мы сделали серверную часть, теперь нам необходимо, чтобы данные сохранялись в базу данных выполнением запроса SQL. Мы можем использовать те же самые методы сохранения данных модели, однако теперь сохранения должны вестись не в строку, а в виде запроса.

Задание для самостоятельной работы

Реализовать хранение данных модели в БД. Использовать ранее разработанную структуру модели данных. Для хранения данных использовать базу данных MySQL. В методах обработки запросов использовать язык запросов SQL и классы Statement и ResultSet. Создать файл для образования структуры и заполнения начальными данными таблиц.

Тема 10. Технологии RMI и EJB

Цель: исследовать средства распределенной обработки данных EJB и технологии RMI.

Теория и примеры выполнения задания

Традиционный подход к выполнению кода на любой машине по сети был утомителен и подвержен ошибкам при реализации. Лучший способ представить эту проблему – это думать, что какой-то объект живет на другой машине, вы можете посылать сообщения удаленному объекту и получать результат, будто бы этот объект живет на вашей машине. Говоря простым языком, это в

точности то, что позволяет делать удаленный вызов методов (Remote Method Invocation (RMI)) в Java.

RMI делает тяжелым использование интерфейсов. Когда вы хотите создать удаленный объект, вы помечаете, что лежащую в основе удаленного объекта реализацию, нужно передавать через интерфейс. Таким образом, когда клиент получает ссылку на удаленный объект, на самом деле он получаете ссылку на интерфейс, который выполняет соединение с определенным местом кода, общающимся по сети. Когда вы создаете удаленный интерфейс, вы должны следовать следующей инструкции:

1. Удаленный интерфейс должен быть публичным – public (он не может иметь «доступ на уровне пакета», также он не может быть «дружественным»). В противном случае клиенты будут получать ошибку при попытке загрузки объекта, реализующего удаленный интерфейс.

2. Удаленный интерфейс должен расширять интерфейс java.rmi.Remote.

3. Каждый метод удаленного интерфейса должен объявлять java.rmi.RemoteException в своем предложении throws в добавок к любым исключениям, специфичным для приложения.

4. Удаленный объект, передаваемый как аргумент или возвращаемое значение (либо напрямую, либо как к части локального объекта), должен быть объявлен как удаленный интерфейс, а не реализация класса.

Технологию EJB (Enterprise Java Beans) можно рассматривать с двух точек зрения: как фреймворк и как компонент. С точки зрения компонента EJB – это всего-лишь надстройка над POJO-классом, описываемая с помощью аннотации. Существует три типа компонентов EJB:

1) session beans – используется для описания бизнес-логики приложения;

2) message-driven beans – также используется для бизнес-логики;

3) entities – используется для хранения данных.

С точки зрения фреймворка, EJB – это технология, предоставляющая множество готовых решений (управление транзакциями, безопасность, хранение информации и т. п.) для вашего приложения.

Перед тем как продолжить обзор основ EJB, остановимся на основе любого приложения – архитектуре. Существует две основные архитектуры при разработке enterprise-приложений:

1) традиционная слоистая архитектура (traditional layered architecture);

2) domain-driven design (DDD).

Обе эти архитектуры предпологают разделение приложения на функциональные слои, каждый из которых используется для решения задач определенного плана.

Например, традиционная слоситая архитектура предполагает разделение приложения на четыре базовых слоя: слой презентации, бизнес-логики, хранения данных и непосредственно слой самой базы данных.

Обычно слой презентации реализуется через web-приложение (т. е. используя JSP, JSF, GWT и т. п.) или web-сервис (что дает возможность написания клиента, например, на C#). В нем реализовано взаимодействие с

пользователем: формы для получения запросов от пользователя и средства для предоставления ему запрошенной информации. Слой бизнес-логики является основой для enterprise-приложения. В нем описываются бизнес-процессы, производится поиск, авторизация и множество других процессов. Слой бизнес-логики использует механизмы слоя хранения данных. Слой хранения данных отличается от слоя базы данных тем, что в первом описываются высокоуровневые объектно-ориентированные механизмы для работы с сущностями БД, в то время как второй – это и есть непосредственно база данных (Oracle, MySQL и т. п.)

Архитектура DDD предполагает, что объекты обладают бизнес-логикой, а не являются простой репликацией объектов БД. Многие программисты не любят наделять объекты логикой и создают отдельный слой, называемый service layer или application layer. Он похож на слой бизнес-логики традиционной слоистой архитектуры с тем лишь отличием, что он намного тоньше. Как уже было сказано выше, существует три типа компонентов EJB: session beans, message-driven beans и entities.

Session beans вызываются пользователем для совершения какой-либо бизнес-операции. Существует два типа session beans: stateless и stateful.

- stateful beans автоматически сохраняют свое состояние между разными клиентскими вызовами. Типичным примером stateful beans является корзина в интернет-магазине;

- stateless beans используются для реализации бизнес-процессов, которые могут быть завершены за одну операцию. Также на основе stateless beans проектируются web-сервисы.

Message-driven beans так же, как и session beans, используются для бизнес-логики. Отличие в том, что клиенты никогда не вызывают MDB напрямую. Обычно сервер использует MDB в асинхронных запросах.

Одним из главных достоинств EJB3 стал новый механизм работы с persistence – возможность автоматически сохранять объекты в реляционной БД, используя технологию объектно-реляционного маппинга (ORM).

В контексте EJB3 persistence-провайдер – это ORM-фреймворк, который поддерживает EJB3 Java Persistence API (JPA). JPA определяет стандарт:

1) для конфигурации маппинга сущностей приложения и их отображения в таблицах БД;

2) EntityManager API – стандартный API для CRUD-операций над сущностями (create, read, update, delete);

3) Java Persistence Query Language (JPQL) – для поиска и получения данных приложения.

Можно сказать, что session beans – это «глаголы» приложения, в то время как entities – это «существительные».

EntityManager – это инерфейс, который связывает класс сущности приложения и его представление в БД. EntityManager знает, как нужно добавлять сущности в базу, обновлять их, удалять, а также предоставляет механизмы для настройки производительности, кэширования, транзакций и т. д.

JPQL – это похожий на SQL язык запросов.

Рассмотрим реализацию этих сущностей. В EJB3 мы используем POJO (Plain Old Java Objects), POJI (Plain Old Java Interfaces) и аннотации. Если с первыми двумя все понятно, то про аннотации стоит поговорить отдельно. Аннотация записывается так:

@<имя аннотации>(<список параметров-значение>)

Аннотации могут быть следующих видов:

1) Stateless говорит контейнеру, что класс будет stateless session bean. Для него контейнер обеспечит безопасность потоков и менеджмент транзакций. Дополнительно вы можете добавить другие свойства, например, прозрачное управление безопасностью и перехватчики событий;

2) Local относится к интерфейсу и говорит, что bean, реализующий интерфейс, доступен локально;

3) Remote относится к интерфейсу и говорит, что bean доступен через RMI (Remote Method Invocation);

4) EJB применятся в коде, где мы используем bean;

5) Stateful говорит контейнеру, что класс будет stateful session bean;

6) Remove – опциональная аннотация, которая используется с stateful beans. Метод, помеченный как Remove, говорит контейнеру, что после его исполнения нет больше смысла хранить bean, т. е. его состояние сбрасывается. Это бывает критично для производительности;

7) Entity говорит контейнеру, что класс будет сущностью БД;

8) Table(name="...") указывает таблицу для маппинга;

9) Id, Column – параметры маппинга;

10) WebService говорит, что интерфейс или класс будет представлять web-сервис.

В качестве session bean может выступать обычный класс Java, удовлетворяющий следующим условиям:

- иметь как минимум один метод;
- не должен быть абстрактным;
- иметь конструктор по умолчанию;
- методы не должны начинаться с «ejb» (например ejbCreate, ejbDoSomething).

Для stateful beans существует еще одно условие: свойства класса должны быть объявлены примитивами или реализовывать интерфейс Serializable.

У stateless и MDB beans существует два события жизненного цикла, которые мы можем перехватить: создание и удаление бина. Метод, который будет вызываться сразу после создания бина, помечается аннотацией javax.annotation.PostConstruct, а перед его удалением – javax.annotation.PreDestroy. Stateful beans обладают помимо рассмотреных выше еще двумя событиями: при активизации события (javax.ejb.PostActivate) и при деактивизации (javax.ejb.PrePassivate).

Один бин может содержать множество клиентских методов. Этот момент является важным для производительности, т. к. контейнер помещает

экземпляры stateless beans в общее хранилище и множество клиентов могут использовать один экземпляр бина. В отличие от stateless stateful beans инстанцируются для каждого пользователя отдельно.

Интерфейс может быть помечен как Local, что сделает классы, реализующие этот интерфейс, классами локальной бизнес-логики. Локальные интерфейсы не требуют никаких дополнительных действий при реализации.

В противном случае интерфейс может быть помечен как Remote, что обеспечит возможность работы RMI. Обычно такой интерфейс расширяет интерфейс Remote, но это необязательно. Рассмотрим пример функциональности интерфейсов Local и Remote из «EJB 3 in Action».

```
public interface BidManager{
   void addBid(Bid bid);
   List<Bid> getBids(Item item);
}

@Local
public interface BidManagerLocal extends BidManager {
   void cancelBid(Bid bid);
}

@Remote
public interface BidManagerRemote extends BidManagerLocal {}

@WebService
public interface BidManagerWS extends BidManager {}
```

При создании enterprise-приложений часто возникает необходимость записывать лог вызываемых методов (в целях отладки или для лога безопасности), а также контролировать доступ пользователей к отдельным частям приложения. Для этого используются перехватчики – объекты, методы которых вызываются автоматически при вызове метода EJB-бина. Объект-перехватчик является POJO, за тем лишь исключением, что метод, который должен вызываться автоматически, аннотируется @AroundInvoke, например:

```
public class MyLogger {
   @AroundInvoke
   public Object logMethodEntry (InvocationContext invocationContext) throws
Exception {
      System.out.println("Entering methid:" +
invocationContext.getMethod().getName() );
         return invocationContext.proceed();
   }
}
```

Обратите внимание на возвращаемое значение и параметр функции. В данном случае мы говорим контейнеру, что после вызова перехватчика можно вызывать метод, который он перехватил, или следующий за ним перехватчик. Однако мы могли сгенерировать исключение или не вызывать метод proceed() и тогда метод EJB-бина не выполнился бы.

Использовать перехватчики можно двумя путями: указать его применение через аннотации для каждого класса или метода в отдельности или указать перехватчик по умолчанию для определенных (или всех) бинов.

Чтобы перехватчик применился только к определенному методу, перед декларацией метода можно написать аннотацию @Interceptors(MyLogger.class). Чтобы перехватчик работал для всех методов бина, эту же аннотацию можно было бы написать перед декларацией класса, например:

```
@Interceptors(MyLogger.class)
@Stateless
public class MyClass { ... }
```

Можно указывать несколько перехватчиков, тогда их перечисляют через запятую, например:

```
@Interceptors(MyLogger1.class, MyLogger2.class)
```

Порядок вызова перехватчиков никак нельзя задать через аннотации, но его можно изменять, если описывать их через дескриптор развертывания. Перехватчик описывается так:

```
<assembly-descriptor>
   <interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>com.example.MyLogger</interceptor-class>
   </interceptor-binding>
</assembly-descriptor>
```

Сначала вызывается перехватчик по умолчанию, потом специфичный для класса, а за ним для метода. Если вы не хотите, чтобы для вашего метода вызывался перехватчик по умолчанию, – используйте аннотацию.

```
@ExcludeDefaultInterceptors или @ExcludeClassInterceptors.
```

На прошлом занятии мы реализовали сохранение данных в базу данных, наше приложение расположено на нескольких компьютерах, данные от клиента передаются по сети. Теперь требуется правильно описать и настроить серверную часть для удаленных вызовов ее команд. Для этого необходимо использовать архитектуру EJB:

1) описать основные методы серверной части в виде интерфейса;

2) правильно оформить реализацию методов серверной части, сделать удаленный интерфейс и зарегистрировать сервисный код в JNDI;

3) метод на клиентской части, который выполняет запросы к серверу, будет использовать этот удаленный интерфейс.

Задание для самостоятельной работы

Разработать удаленные вызовы бизнес-методов модели согласно полученному варианту задания. Обработка данных модели должна производиться распределенно. Использовать ранее разработанные бизнес-методы. Реализовать вызов удаленных методов согласно стандарту RMI. Оформить бизнес-методы в виде распределенных модулей EJB.

Тема 11. Технологии JSP и JavaBeans

Цель: исследовать технологии создания динамических страниц JSP и JavaBeans.

Теория и примеры выполнения задания

Сервлеты – это компоненты приложений J2EE, выполняющиеся на стороне сервера, способные обрабатывать клиентские запросы и динамически генерировать ответы на них. Наибольшее распространение получили сервлеты, обрабатывающие клиентские запросы по протоколу HTTP. Сервлет может применяться, например, для создания серверного приложения, получающего от клиента запрос, анализирующего его и делающего выборку данных из базы данных, а также пересылающего клиенту страницу HTML, сгенерированную с помощью JSP на основе полученных данных.

Все сервлеты реализуют общий интерфейс Servlet. Для обработки HTTP-запросов можно воспользоваться в качестве базового класса абстрактным классом HttpServlet. Базовая часть классов JSDK помещена в пакет Javax.servlet. Однако класс HttpServlet и все, что с ним связано, располагаются на один уровень ниже в пакете Javax.servlet.http.

Жизненный цикл сервлета начинается с его загрузки в память контейнером сервлетов при старте либо в ответ на первый запрос. Далее происходят инициализация, обслуживание запросов и завершение существования.

Первым вызывается метод init(). Он дает сервлету возможность инициализировать данные и подготовиться для обработки запросов. Чаще всего в этом методе программисты помещают код, кэширующий данные фазы инициализации.

После этого сервлет можно считать запущенным, он находится в ожидании запросов от клиентов. Появившийся запрос обслуживается методом service() сервлета, а все параметры запроса упаковываются в объект ServletRequest, который передается в качестве первого параметра методу

service(). Второй параметр метода – объект ServletResponse. В этот объект упаковываются выходные данные в процессе формирования ответа клиенту. Каждый новый запрос приводит к новому вызову метода service(). В соответствии со спецификацией JSDK метод service() должен уметь обрабатывать сразу несколько запросов, т. е. быть синхронизирован для выполнения в многопоточных средах. Если же нужно избежать множественных запросов, сервлет должен реализовать интерфейс SingleThreadModel, который не содержит ни одного метода и только указывает серверу об однопоточной природе сервлета. При обращении к такому сервлету каждый новый запрос будет ожидать в очереди, пока не завершится обработка предыдущего запроса.

После завершения выполнения сервлета контейнер сервлетов вызывает метод destroy(), в теле которого следует помещать код освобождения занятых сервлетом ресурсов.

Интерфейсом Servlet предусмотрена реализация еще двух методов: getServletConfig() и getServletInfo(). Первый возвращает объект типа ServletConfig, содержащий параметры конфигурации сервлета, а второй – строку, описывающую назначение сервлета.

При разработке сервлетов в качестве базового класса в большинстве случаев используют не интерфейс Servlet, а класс HttpServlet, отвечающий за обработку запросов HTTP.

Класс HttpServlet имеет реализованный метод service(), служащий диспетчером для других методов, каждый из которых обрабатывает методы доступа к ресурсам. В спецификации HTTP определены следующие методы: GET, HEAD, POST, PUT, DELETE, OPTIONS и TRACE. Наиболее часто употребляются методы GET и POST, с помощью которых на сервер передаются запросы, а также параметры для их выполнения.

При использовании метода GET (по умолчанию) параметры передаются как часть URL, значения могут выбираться из полей формы или передаваться непосредственно через URL. При этом запросы кэшируются и имеют ограничения на размер. При использовании метода POST (method=POST) параметры (поля формы) передаются в содержимом HTTP-запроса и упакованы согласно полю заголовка Content-Type (по умолчанию в формате: <имя>=<значение>&<имя>=<значение>&).

Однако форматы упаковки параметров могут быть самые разные, например, в случае передачи файлов с использованием формы enctype="multipart/form-data".

В задачу метода service() класса HttpServlet входит анализ полученного через запрос метода доступа к ресурсам и вызов метода, имя которого сходно с названием метода доступа к ресурсам, но перед именем добавляется префикс do: doGet() или doPost(). Кроме этих методов могут использоваться и следующие: doHead(), doPut(), doDelete(), doOptions() и doTrace(). Разработчик должен переопределить нужный метод, разместив в нем функциональную логику.

Ранее мы сделали десктопный интерфейс, теперь сделаем web-интерфейс для нашего приложения. Будем использовать web-сервер, а также сделаем код, генерирующий разметку и дизайн для нашего web-интерфейса. Для этого необходимо:

1) использовать технологию JSP для генерации разметки html на платформе Java;

2) создать структуру страниц;

3) создать контроллер для обработки HTTP-запросов (однако часть сервера по прежнему может быть оформлена в виде EJB). Задача контроллера будет состоять в подготовка данных для страница JSP.

Широко использующийся архитектурой создания web-приложений является архитектура Model 2.

Задание для самостоятельной работы

Разработать вывод информации приложения согласно архитектуре клиент-сервер с использованием технологии автоматического формирования HTML-страниц. Использовать технологию JSP для создания пользовательских тегов и библиотеку JSTL. Реализовать разделение средств формирования разметки файлов для изменения стиля и функциональности на стороне клиента для получаемых web-страниц приложения.

Тема 12. Средства для обеспечения безопасности

Цель: исследовать средства для обеспечения безопасности для клиент-серверной архитектуры приложений.

Теория и примеры выполнения задания

HTTP является протоколом, не использующим сессии. Наиболее общий метод отслеживания сессий связан с наличием «Cookies», что является интегрированной частью стандарта Internet.

API сервлетов (версии 2.0 и выше) имеет класс Cookie. Этот класс объединяет все детали HTTP-заголовка и позволяет устанавливать различные атрибуты Cookie. Использовать Cookie просто, поэтому есть смысл добавлять его в объекты ответов. Конструктор получает имя Cookie в качестве первого аргумента, а значение – в качестве второго. Cookies извлекаются путем вызова метода getCookies( ) объекта HttpServletRequest, который возвращает массив из объектов cookie. Класс Session из API сервлета использует класс Cookie, чтобы сделать эту работу. Однако всем объектам Session необходим уникальный идентификатор некоторого вида, хранимый на стороне клиента и передающийся на сервер. Web-сайты могут также использовать другие типы слежения за сессиями, но эти механизмы более сложны для реализации, т. к. они не инкапсулированы в API сервлетов (следовательно, нужно писать их руками, чтобы разрешить ситуации, когда клиент отключает Cookies).

Для обеспечения безопасности нашего web-приложения определим набор страниц, которые должны быть общедоступны, и страниц, которые должны быть доступны только после аутентификации. Архитектура REST предполагает, что сервер не должен хранить состояние, связанное с клиентом. Решением этой задачей является создание сессии для клиентов. После успешной аутентификации для клиента добавляется Cookie-идентификатор и на серверной стороне создается объект сессии. Контроллер добавляет необходимые данные в объект сессии а JSP-страница визуализирует сохраненную в сессию информацию.

Задание для самостоятельной работы

Разработать регистрацию пользователей с использованием страниц JSP, а также конфигурации сервера Tomcat.

Для этого необходимо выполнить следующие действия:

1) создать код по управлению аутентификацией и форму для ввода логина и пароля пользователем;

2) для каждого контроллера, выполнение которого должно быть защищено, добавить проверку аутентификации пользователя. В случае отсутствия разрешения добавить перенаправление на страницу аутентификации;

3) добавить метод для выхода пользователя из системы. Этот метод должен удалять аутентификационные данные из сессии.

Реализовать две группы пользователей для работы с разработанным приложением: для первой группы пользователей должны быть доступны операции чтения и отображения информации, для второй – должны быть доступны все операции, в том числе редактирования и удаления данных.

ЛИТЕРАТУРА

1 Интернет-технологии и распределенная обработка данных : электронный учеб.-метод. комплекс [Электронный ресурс]. – 2007. – Режим доступа : http://bsuir.by.
2 Основы интернет-технологий : электронный учеб.-метод. комплекс [Электронный ресурс]. – 2007. – Режим доступа : http://bsuir.by.
3 Лабораторный практикум по курсу «Разработка информационных систем в WWW» для студентов экономических специальностей БГУИР. В 2 ч. Ч. 1 / А. В. Лепеш [и др.]. – Минск : БГУИР, 2002. – 56 с.
4 Эккель, Б. Философия Java. Библиотека программиста / Б. Эккель. – СПб. : Питер, 2001. – 880 с.
5 Ноутон, П. Java 2 / П. Ноутон; пер. с англ. – СПб. : BHV-Санкт-Петербург, 2001. –1072 с.
6 Симкин, С. Программирование на JAVA: путеводитель / С. Симкин ; пер. с англ. – Киев : DiaSott. 1996. – 736 с.
7 Вебер, Д. Технология Java в подлиннике / Д. Вебер. – СПб. : BHV-Санкт-Петербург, 1997. – 1104 с.
8 Мейнджер, Д. Java: основы программирования / Д. Мейнджер ; пер. с англ. ; под ред. Я. Шмидского. – Киев : BHV, 1997. – 320 с.
9 Джамса, К. Библиотека программиста Java / К. Джамса ; пер. с англ. – Минск : Попурри, 1997. – 640 с.
10 Хабибуллин, И. Ш. Создание распределенных приложений на Java-2 / И. Ш. Хабибуллин. – СПб. : БХВ – Петербург, 2002. – 704 с.
11 Ноутон, П. Java 2 / П. Ноутон, Г. Шилдт ; пер. с англ. – СПб. : БХВ – Петербург, 2001.
12 Дейтел, Х. М. Технологии программирования на Java 2 / Х. М. Дейтел, П. Дж. Дейтел, С. И. Сантри ; пер. с англ ; под ред. А. И. Тихонова. – М. : Бином-пресс, 2003.
13 Бишоп, Дж. Java 2 / Дж. Бишоп ; пер. с англ. – СПб. : Питер, 2002. – 592 с.
14 Хортон, А. Java 2. JDK 1.3: В 2 т. Т.1 / А. Хортон ; пер. с англ. – М. : Лори, 2002. – 486 с.
15 Морган, М. Java 2. Руководство разработчика / М. Морган ; пер. с англ. – Изд. дом «Вильямс», 2000. – 719 с.
16 Смирнов, Н. И. Java 2 : учеб. пособие / Н. И. Смирнов. – М. : Три Л, 2000. – 316 с.
17 Хабибуллин, И. Ш. Java: самоучитель / И. Ш. Хабибуллин. – СПб. : BHV-Санкт-Петербург, 2001. – 462 с.
18 Баженова, И. Ю. JBuilder. Программирование на Java / И. Ю. Баженова. – М. : КУДИЦ-ОБРАЗ, 2001. – 447 с.
19 Хабибуллин, И. Ш. Создание распределенных приложений на Java 2 / И. Ш. Хабибуллин. – СПб. : BHV-Санкт-Петербург, 2002. – 696 с.

Св. план 2014, поз. 17

Учебное издание

Быков Антон Алексеевич
Мельникова Елена Александровна
Яшин Константин Дмитриевич

ТЕХНОЛОГИИ ВИРТУАЛЬНОЙ РЕАЛЬНОСТИ. ПОСОБИЕ К ПРАКТИЧЕСКИМ ЗАНЯТИЯМ

ПОСОБИЕ

Редактор *Е. С. Чайковская*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *М. В. Гуртатовская*

Подписано в печать 09.01.2015. Формат 60x84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 3,84**.** Уч.-изд. л. 4,0. Тираж 100 экз. Заказ 116.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
ЛП №02330/264 от 14.04.2014.
220013, Минск, П. Бровки, 6