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

Кафедра инженерной психологии и эргономики

И. В. Байдаков, К. Д. Яшин

ПСИХОЛОГИЯ ВЗАИМОДЕЙСТВИЯ ЧЕЛОВЕКА С ВИРТУАЛЬНОЙ РЕАЛЬНОСТЬЮ

Методическое пособие
к практическим занятиям по современным технологиям программирования
для студентов специальности 1-58 01 01
«Инженерно-психологическое обеспечение информационных технологий»
всех форм обучения

Минск БГУИР 2011

УДК 004.946:159.953(076.5)
ББК 32.973.26-018.2я73+88.5я73
Б18

Р е ц е н з е н т:
доцент кафедры информатики Белорусского государственного университета информатики и радиоэлектроники, кандидат технических наук
С. И. Сиротко

Байдаков, И. В.

Б18 Психология взаимодействия человека с виртуальной реальностью : метод. пособие к практич. занятиям по современным технологиям программирования для студ. спец. 1-58 01 01 «Инженерно-психологическое обеспечение информационных технологий» всех форм обуч. / И. В. Байдаков, К. Д. Яшин. – Минск : БГУИР, 2011. – 38 с.

ISBN 978-985-488-573-5.

В данном пособии, входящем в учебно-методический комплекс, изучаются основы психологии взаимодействия человека с виртуальной реальностью, освещаются вопросы объектно-ориентированного Java-программирования: работа со строками, работа с файлами, работа с сетевыми средствами. Особое внимание уделяется специфике объектно-ориентированного программирования для языка Java.

Предназначено для практических занятий студентов специальности «Инженерно-психологическое обеспечение информационных технологий» всех форм обучения. Полная версия пособия имеется на кафедре инженерной психологии и эргономики БГУИР.

УДК 004.946:159.953(076.5)
ББК 32.973.26-018.2я73+88.5я73

ISBN 978-985-488-573-5

Содержание

Занятие 1. Компиляция Java-файлов и освоение платформы Eclipse 4
Занятие 2. Инструменты для сборки Java-приложений Apache Ant 7
Занятие 3. Объектно-ориентированные возможности языка Java 13
Занятие 4. Коллекции Java 22
Занятие 5. Строки и система ввода-вывода Java 26
Занятие 6. Многопоточность в Java 31
Занятие 7. Сетевые возможности Java 35
Литература 37

Занятие 1. Компиляция Java-файлов и освоение платформы Eclipse

Цель: приобретение первичных навыков работы в среде разработки Eclipse и освоение процесса разработки программ на языке Java.

Теория и примеры решения задач

Запустите установочный файл, например *jdk-6u5-windows-i586-p.exe.*Установщик распаковывает файлы, необходимые для установки. По окончании распаковки появится экран приветствия и лицензионное соглашение. Нажмите на кнопку «Согласен» (Accept**)**, чтобы принять условия соглашения и продолжить процесс установки. На экране появится окно «Режим выборочной установки», где можно выбрать компоненты программного обеспечения, которые будут установлены. Удостоверившись, что нужные компоненты программы выбраны, щелкните на кнопке «Далее» (Next) для продолжения установки; одно за другим появятся несколько диалоговых окон, подтверждающих выполнение завершающих шагов процесса установки. Полное окончание установки подтверждается сообщением «Thank You!» (Благодарим).

Платформа Eclipse поставляется с Java Integrated Development Environment (IDE) для разработки Java-приложений. Она предоставляет некоторые обычные инструментальные средства и функциональные возможности для Java-разработчиков. Созданный на основе платформы Eclipse, Application Developer добавляет большое количество возможностей, включая J2EE-разработку, XML, разработки Web-служб и баз данных. При запуске Application Developer выберите место для расположения рабочего пространства (workspace). Рабочим пространством может быть любой каталог, в котором будут сохраняться ваши рабочие файлы. Если одновременно идет работа над несколькими проектами, используйте отдельные рабочие пространства для каждого из них, для того чтобы гарантировать разделение кода. Отобразится перспектива J2EE. Перспектива (perspective) представляет собой объединение инструментальных средств и видов (view), необходимых разработчику. Application Developer предлагает множество перспектив, специально настроенных для различных типов разработки, например: Web, Data, J2EE, Debug и Java-программирование. Application Developer поддерживает разработку различных типов приложений, в том числе: Java, Web, EJB, J2EE, Database, Web Service JMS, SQLJ, Портал.

Visual Modeling – визуальное моделирование, позволяющее визуализировать исходный код в виде диаграммы, последовательности, тематической диаграммы и т.д. Возможна разработка, тестирование и отлаживание любых из указанных приложений прямо в Application Developer.

Данное упражнение в пошаговом режиме демонстрирует процесс создания Java-приложения, выводящего строку " Hello, World! ": запустите Application Developer, если этого еще не сделали: в меню Windows Start выберите *Programs > IBM Rational > IBM Rational Application Developer v6.0 > Rational Ap-*

plication Developer; отобразится окно, которое запросит ввести каталог рабочего пространства. Нажмите кнопку «OK» для выбора каталога по умолчанию.

Создайте Java-проект под названием MyJavaProject, который имеет папки source и binary для хранения Java и class-файлов соответственно: в рабочей области (workbench) выберите *File* > *New* > *Project*; выберите *Java* > *Java Project* > *Next.* Если вы не видите Java, отметьте флажок *Show All Wizards*. Нажмите кнопку OK, если отобразится запрос о разрешении возможности Java-разработки; введите *MyJavaProject* в качестве названия проекта; выберите вариант "*Select the Create separate source and output folders*" и нажмите кнопку *Finish*; нажмите кнопку *Yes*, если отобразится запрос о переходе в перспективу *Java Perspective*.

Создайте Java-класс под названием: Test внутри пакета test, как показано на рис. 1: в виде Package Explorer нажмите правой кнопкой мыши на *MyJavaProject* и выберите пункт *New* > *Class*; введите *test* в качестве пакета и *Test* в качестве имени файла; должен быть отмечен флажок *public static void main(String[] args)*; нажмите кнопку *Finish* – откроется Java-редактор. В Java-редакторе Java-класс измените так, как показано в листинге 1.

Листинг 1. Исходный код класса test.Test

```
public class Test {
public static void main(String[] args) {
System.out.println ("Hello,World!!");
System.out.println ("Hello, World Again!!");
}
}
```

Во время ввода текста можно использовать функцию Code Assist (Ctrl-Space), помогающую завершать ключевые слова. Сохраните файл, нажав Ctrl-S.

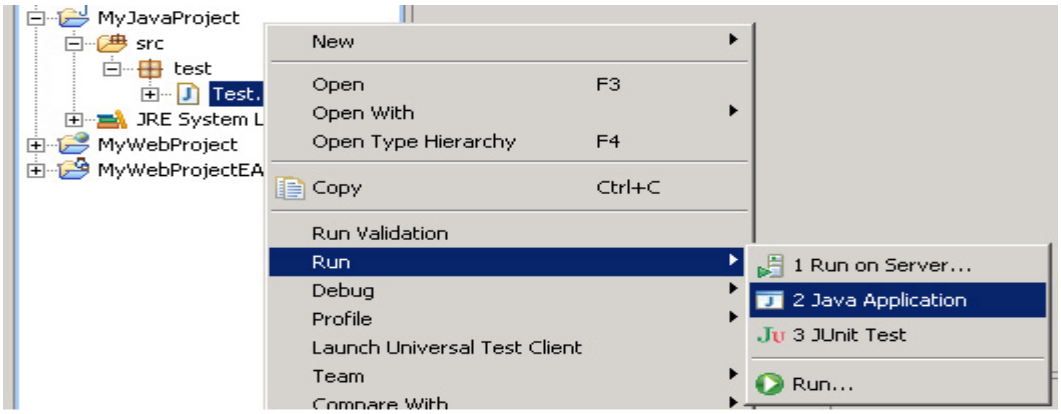

Рис. 1. Выполнение Java-приложения

Теперь можно запустить первое Java-приложение:

1) в виде Package Explorer нажмите правой кнопкой мыши на классе Test и выберите Run > Java Application, как показано на рис. 1.

2) перейдите в вид Console для просмотра результатов работы приложения, показанных на рис. 2. Если вид Console не отображен, можно перейти в него путем выбора меню *Windows menus* > *Show View* > *Console*.

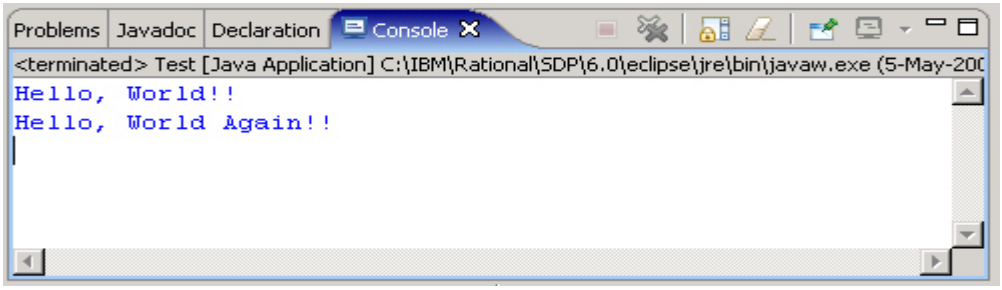

Рис. 2. Вид окна Console

В окне Console отображаются результаты работы вашего Java-приложения. Вы увидите две строки текста, распечатанные из Java-приложения. Для запуска приложения в Console требуется найти откомпилированный class-файл (обычно в папке bin проекта) и передать его в качестве аргумента интерпретатору Java: *java.exe D:\projects\my-project\bin\Test. Обратите внимание: что расширение (.class) не указывается!*

Задание на самостоятельную работу

Написать программу на Java, выводящую в консоль строчку «Hello, world!». Программу необходимо скомпилировать и запустить вручную из командной строки и из среды Eclipse.

Контрольные вопросы

1. Какие средства предоставляет среда Eclipse для разработчика java-приложений?
2. Какие дополнительные возможности, помимо написания Java-приложений имеются в среде Eclipse?
3. Что такое интерпретатор Java? Укажите его расположение на жестком диске.
4. Что такое компилятор Java? Укажите его расположение на жестком диске.

Занятие 2. Инструменты для сборки Java-приложений Apache Ant

Цель: приобретение навыков создания ant-скриптов для автоматизации процесса разработки Java-приложений.

Теория и примеры решения задач

Ant – это инструмент для работы с проектами. С его помощью можно выполнить компиляцию, отладку и тестирование проекта; создавать новые и удалять существующие файлы и папки; создавать архивы с исходными кодами и др. Естественно, все эти операции Ant выполняет с помощью дополнительных средств. Например, для компиляции проекта должен быть установлен Java SDK (software development kit). Ant выполняет чтение и анализ специального файла (обычно он называется build.xml), который содержит команды для работы с проектом. Этот файл можно создать самостоятельно, либо с помощью IDE в ходе разработки проекта.

Рассмотрим конкретный пример. Допустим, имеется проект, состоящий из одного файла (hello.java) с программой "Hello, World!". Необходимо: скомпилировать программу; запустить её (для проверки работоспособности); создать jar-файл (файл manifest.mf имеется); создать архив с исходниками (например для размещения в Internet). Нужно создать три переменных: *ANT_HOME = C:\ant* (укажите здесь путь к папке, в которую установлен Ant); *JAVA_HOME = C:\Program Files\Java\jdk1.5.0_06* (укажите здесь путь к папке, в которую установлен Java SDK); *PATH = C:\ant\bin* (укажите тут путь к папке, в которой находится файл ant.bat).

В первую очередь нужно продумать структуру проекта: проект находится в папке myProject; исходники – в папке src; тесты – в папке test; скомпилированные файлы исходников должны размещаться в папке dist\classes; скомпилированные тесты – в dist\tests; jar-файл будет размещён в папке dist.

В данный проект входят три файла программы (Main.java, Class1.java, Class2.java), манифест (manifest.mf) и два файла с тестами (Class1Test.java, Class2Test.java). Причём файлы Class1.java и Class2.java входят в состав пакета tools.utils. Таким образом, проект имеет следующую структуру (рис. 3).

Теперь нужно создать файл build.xml (название файла может быть любым, но тогда при вызове Ant его придётся указывать явно). Этот файл имеет xml формат, информации о котором более чем достаточно в Internet. Но глубокие знания этого формата в данном случае не нужны. В документации для каждой задачи приводится необходимый xml-фрагмент с описанием параметров. Любой xml-документ начинается со строки

<?xml version="1.0" encoding="windows-1251"?>

в которой указываются номер версии xml и кодировка, которая используется в файле. Все задачи, описанные в build- файле, должны находится внутри тега

<project name="myProject" basedir="." default="run"> . . ,

который, в нашем случае, имеет три параметра: `name` – имя проекта; `basedir` – имя папки, относительно которой рассчитываются все пути в проекте (точка означает текущую папку); `default` – имя задачи, которая будет выполнена по умолчанию.

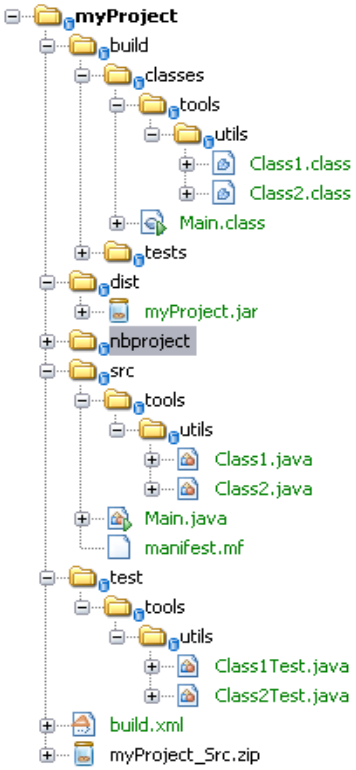

Рис. 3. Структура проекта

Теперь перейдём непосредственно к написанию задач. Предварительно создадим несколько свойств.

1. *<property name="src" location="src"/>*
2. *<property name="buildSrc" location="build/classes"/>*
3. *<property name="buildTest" location="build/tests"/>*
4. *<property name="dist" location="dist"/>*
5. *<property name="test" location="test"/>*

Например, выражение `myProject/${buildTest}` будет преобразовано в myProject/build/tests.

Теперь перейдём непосредственно к созданию задач и в первую очередь той задачи, которая создаёт папки build/classes, build/tests и dist. Назовём её `init` (инициализация).

1. *<target name="init">*
2. *<mkdir dir="${buildSrc}"/>*
3. *<mkdir dir="${buildTest}"/>*
4. *<mkdir dir="${dist}"/>*
5. *</target>*

Описание каждой задачи должно находиться внутри тега target. Этот тег имеет несколько параметров, но в данном случае используется только один – name, который задаёт имя задачи**.**

Тег *mkdir* – создаёт папку, а его параметр dir – указывает, какую именно.

Следующая задача *compile***.**

```
1. <target name="compile" depends="init">
2. <javac srcdir="${src}" destdir="${buildSrc}">
3. <javac srcdir="${test}" destdir="${buildTest}">
4. <classpath>
5. <pathelement path="C:/junit3.8.1/junit.jar"></pathelement>
6. <pathelement path="${buildSrc}"></pathelement>
7. </classpath>
8. </javac>
9. </javac>
10.</target>
```

Здесь появляется ещё один параметр тега `target`, `depends`, в котором указываются имена задач, от которых зависит данная задача, то есть в данном случае при вызове задачи `compile` сначала будет выполнена задача `init`. Можно указать любое количество задач, от которых зависит данная, например: `depends="init, task1, task2"`.

Поскольку файлы скомпилированы, нужно упаковать их в jar-архив. Следующая задача как раз для этого и предназначена.

```
1. <target name="dist" depends="compile" description="Create jar-file">
2. <jar jarfile="${dist}/myProject.jar" basedir="${buildSrc}"
3. manifest="${src}/manifest.mf"/>
4. </target>
```

Теперь напишем задачу для запуска только что созданного jar-файла.

```
1. <target name="run" depends="dist" description="Run program">
2. <java jar="${dist}/myProject.jar" fork="true"/>
3. </target>
```

Теперь запустим наши тесты.

```
1. <pre lang="xml"><target name="test" depends="compile">
2. <junit fork="yes" haltonfailure="yes">
3. <classpath>
4. <pathelement location="${buildTest}"/>
5. <pathelement location="${buildSrc}"/>
6. </classpath>
7. <formatter type="plain" usefile="false" />
8. <test name="tools.utils.Class1Test"/>
9. <test name="tools.utils.Class2Test"/>
10.</junit>
11.</target>
```

Для того чтобы работала эта задача, нужно «рассказать» Ant, где находится библиотека junit.jar. В документации к Ant описывается три различных способа, которые позволяют это сделать. В данном случае скопируем файл junit.jar в папку ANT_HOME/lib. Тестирование выполняется в теге junit, который имеет два параметра: `fork` – запускает тесты в отдельной виртуальной машине; `haltonfailure` – останавливает выполнение в случае, если тест не проходит. Теги `classpath` задают размещение скомпилированных тестов и классов, которые они трестируют.Тег *formatter* задает параметры отображения результатов тестирования. Параметр `type="plain"` указывает, что результаты тестирования должны отображаться в виде обычного текста, а параметр `usefile="false"` обеспечивает вывод результатов на экран, а не в файл. С помощью тегов `test` запускаем тесты. С помощью параметра `name` указываем название теста. Итак, программа протестирована, теперь подготовим её исходный код к размещению в Internet. Для этого упакуем его в обычный zip-архив.

```
<target name="packSrc">
<zip destfile="myProject_Src.zip">
<fileset dir="." includes="**/*.java, **/*.mf,
**/*.xml"/>
</zip>
</target>
```

Создание архива выполняется с помощью тега `zip`. С помощью параметра `destfile` задаем имя архива. Вложенный тег `fileset` позволяет указать перечень файлов, которые войдут в архив. Поясним сказанное. Во-первых, в архив нужно включить только те файлы, которые созданы, а именно: файлы с расширениями java, mf, xml. Во-вторых, поиск файлов, которые нужно включить в проект, начинаем с папки myProject. Для этого используем параметры `dir` и `includes; dir` – задает стартовую папку, `includes` – указывает шаблоны выбора файлов в архив, где ("**" – любая папка, "*" – любое количество любых символов в имени файла).

И наконец с помощью задачи `clean` можем удалить результаты работы всех предыдущих задач (останутся только исходники).

```
<target name="clean">
<delete dir="build"/>
<delete dir="${dist}"/>
<delete file="myProject_Src.zip"/>
</target>
```

Если имя задачи не задано, будет выполнена задача по умолчанию (та, что задана в параметре `default` тега `project`), в нашем случае `run`.Теперь посмотрим весь файл целиком.

```
<?xml version="1.0" encoding="windows-1251"?>
<project name="myProject" basedir="." default="run">
<!–Устанавливаем глобальные свойства для данного проекта–>
<property name="src" location="src"/>
```

```
5.  <property name="buildSrc" location="build/classes"/>
6.  <property name="buildTest" location="build/tests"/>
7.  <property name="dist"  location="dist"/>
8.  <property name="test" location="test"/>
9.  <!–Эта задача создает папки для размещения скомпилированных исходников и дистрибутива –>
10. <target name="init">
11. <mkdir dir="${buildSrc}"/>
12. <mkdir dir="${buildTest}"/>
13. <mkdir dir="${dist}"/>
14. </target>
15. <!–Эта задача выполняет компиляцию проекта–>
16. <target name="compile" depends="init">
17. <javac srcdir="${src}" destdir="${buildSrc}"/>
18. <javac srcdir="${test}" destdir="${buildTest}">
19. <classpath>
20. <pathelement path="C:/junit3.8.1/junit.jar"/>
21. <pathelement path="${buildSrc}"/>
22. </classpath>
23. </javac>
24. </target>
25. <!– Эта задача упаковывает программу в jar-архив –>
26.  <target name="dist" depends="compile" description="Create jar-file">
27.  <jar jarfile="${dist}/myProject.jar" basedir="${buildSrc}"
28.  manifest="${src}/manifest.mf"/>
29.  </target>
30.  <!– Эта задача запускает программу –>
31.   <target name="run" depends="dist" description="Run program">
32.  <java jar="${dist}/myProject.jar" fork="true"/>
33.  </target>
34.  <!– Эта задача упаковывает файлы с исходными кодами и ресурсами в zip-архив –>
35.   <target name="packSrc">
36.  <zip destfile="myProject_Src.zip">
37.  <fileset dir="." includes="**/*.java, **/*.mf, **/*.xml"/>
38.  </zip>
39.  </target>
40.  <!– Эта задача выполняет тестирование проекта –>
41.  <target name="test" depends="compile">
42.  <junit fork="yes" haltonfailure="yes">
43.  <classpath>
44.  <pathelement location="${buildTest}"/>
45.  <pathelement location="${buildSrc}"/>
46.  </classpath>
```

```
47. <formatter type="plain" usefile="false" />
48. <test name="tools.utils.Class1Test"/>
49. <test name="tools.utils.Class2Test"/>
50. </junit>
51. </target>
52. <!– Эта задача удаляет все, кроме исходников –>
53. <target name="clean">
54. <delete dir="build"/>
55. <delete dir="${dist}"/>
56. <delete file="myProject_Src.zip"/>
57. </target>
58. </project>
```

Задание на самостоятельную работу

Написать ant-скрипт сборки программы «hello, world». Должна осуществляться компиляция, формирование jar-файла, исполнение программы из-под скрипта и очистка временных файлов.

Контрольные вопросы

1. В чём состоит необходимость автоматизации процесса разработки?
2. Какие возможности по автоматизации процесса разработки предоставляет инструмент Ant.
3. Опишите элементы (задачи) ant-скрипта по сборке обычного java-приложения.
4. Для чего необходимо подключать дополнительные модули для Ant? Как это сделать?

Занятие 3. Объектно-ориентированные возможности языка Java

Цель: приобретение навыков создания пользовательских Java-классов, интерфейсов, ознакомление с механизмом наследования в Java.

Теория и примеры решения задач

Рассмотрим подробнее принципы объектно-ориентированного программирования. Мы должны абстрагироваться от некоторых конкретных деталей объекта. Очень важно выбрать правильную степень абстракции. Слишком высокая степень даст только приблизительное описание объекта, не позволит правильно моделировать его поведение. Слишком низкая степень абстракции сделает модель очень сложной, перегруженной деталями и потому непригодной. Описание каждой модели производится в виде одного или нескольких классов (classes). После того как описание класса закончено, можно создавать конкретные объекты, экземпляры (instances) описанного класса. Создание экземпляров производится в три этапа, подобно описанию массивов. На первом этапе объявляются ссылки на объекты: записывается имя класса и через пробел перечисляются экземпляры класса, точнее, ссылки на них. На втором этапе операцией new определяются сами объекты: под них выделяется оперативная память, ссылка получает адрес этого участка в качестве своего значения.

lada2110 = new Automobile();
fordScorpio = new Automobile();
oka = new Automobile();

На третьем этапе происходит инициализация объектов, задаются начальные значения. Этот этап, как правило, совмещается со вторым, именно для этого в операции new повторяется имя класса со скобками Automobile(). Поскольку имена полей, методов и вложенных классов у всех объектов одинаковы, они заданы в описании класса и уточняются именем ссылки на объект:

lada2110.maxVelocity = 150;
fordScorpio.maxVelocity = 180;
oka.maxVelocity = 350;
oka.moveTo(35, 120);

Напомним, что текстовая строка в кавычках понимается в Java как объект класса String . Поэтому можно написать

int strlen = "Это объект класса String".length();

Объект "строка" выполняет метод length() – один из методов своего класса String, подсчитывающий число символов в строке. В результате получаем значение strlen, равное 24. Сообщение идет к конкретному объекту, знающему метод решения задачи, в примере этот объект — текущее значение переменной person. У каждого объекта свое текущее состояние, свои значения полей класса, что может повлиять на выполнение метода.

Способ выполнения поручения, содержащегося в сообщении, зависит от объекта, которому оно послано. Один хозяин поставит миску с "Chappi", другой бросит кость, третий выгонит собаку на улицу. Это интересное свойство назы-

вается полиморфизмом (polymorphism) и будет обсуждаться ниже. Обращение к методу произойдет только на этапе выполнения программы, компилятор ничего не знает про метод. Это называется «поздним связыванием» в противовес «раннему связыванию», при котором процедура присоединяется к программе на этапе компоновки.

Принцип модульности утверждает: каждый класс должен составлять отдельный модуль. Члены класса, к которым не планируется обращение извне, должны быть инкапсулированы. В противоположность закрытым, некоторые члены класса объявляем *открытыми,* записав вместо слова private модификатор public , например: public void getFood(int food, int drink). Принцип модульности предписывает открывать члены класса только в случае необходимости. Вспомните надпись: «Нормальное положение шлагбаума – закрытое».

Если же необходимо обратиться к полю класса, то рекомендуется включить в класс специальные *методы доступа* (access methods) – отдельно для чтения этого поля (get method) и для записи в это поле (set method). Имена методов доступа рекомендуется начинать со слов get и set , добавляя к этим словам имя поля. В классе Master методы доступа к полю Name в самом простом виде могут выглядеть так:

```
public String getName(){
return name;
}
public void setName(String newName) {
name = newName;
}
```

В реальных ситуациях доступ ограничивается разными проверками, особенно в *set*-методах, меняющих значения полей. Можно проверять тип вводимого значения, задавать диапазон значений, сравнивать со списком допустимых значений.

Кроме методов доступа рекомендуется создавать проверочные *is-методы,* возвращающие логическое значение true или false . Например, в класс Master можно включить метод, проверяющий, задано ли имя хозяина:

```
public boolean isEmpty(){
return name == null ? true : false;
}
```

и использовать этот метод для проверки при доступе к полю Name , например:

```
if (master01.isEmpty()) master01.setName("Иванов");
```

Итак, оставляем открытыми только методы, необходимые для взаимодействия объектов. При этом удобно спланировать классы так, чтобы зависимость между ними была наименьшей, как принято говорить в теории ООП, чтобы было наименьшее *зацепление* (low coupling) между классами. Тогда структура программы сильно упрощается. Кроме того, такие классы удобно использовать как строительные блоки для построения других программ. Из этого общего схематического описания принципов объектно-ориентированного программи-

рования видно, что язык Java позволяет легко воплощать все эти принципы. Разберем теперь подробнее правила записи классов и рассмотрим дополнительные их возможности.

Описание класса начинается со слова class, после которого записывается имя класса. Соглашения "Code Conventions" рекомендуют начинать имя класса с заглавной буквы. В листинге 2 показано, как можно оформить метод деления пополам для нахождения корня нелинейного уравнения.

Листинг 2 Нахождение корня нелинейного уравнения методом бисекций

```
class Bisection2 {
   private static final double EPS = 1e - 8; // Константа
   private double a = 0.0, b = 1.5, root;  // Закрытые поля
   public double getRoot() {
      return root;
   }  // Метод доступа
   private double f(double x) {
      return x * x * x - 3 * x * x + 3;   // Или другая функция
   }
   private void bisect() {      // Параметров нет —
      // метод работает с полями экземпляра
      double y = 0.0;            // Локальная переменная — не поле
      do {
         root = 0.5 * (a + b);
         y = f(root);
         if (Math.abs(y) < EPS) break;
// Корень найден. Выходим из цикла
// Если на концах отрезка [a; root]
// функция имеет разные знаки:
         if (f(a) * y < 0.0)
            b = root;
// значит, корень здесь
// Переносим точку b в точку root
//В противном случае:
         else
            a = root;
// переносим точку a в точку root
// Продолжаем, пока [a; Ь] не станет мал
      } while (Math.abs(b - a) >= EPS);
   }
   public static void main(String[] args) {
      Bisection2 b2 = new Bisection2();
      b2.bisect();
      System.out.println("x = " +
            b2.getRoot() + // Обращаемся к корню через метод доступа
            ", f() = " + b2.f(b2.getRoot()));
```

```
    }
}
```

В описании метода f() сохранен старый, процедурный стиль: метод получает аргумент, обрабатывает его и возвращает результат. Описание метода bisect 0 выполнено в духе ООП: метод активен, он сам обращается к полям экземпляра b2 и сам заносит результат в нужное поле. Метод bisect () — это внутренний механизм класса Bisection2, поэтому он закрыт (private).

Имя метода, число и типы параметров образуют *сигнатуру* (signature) метода. Компилятор различает методы не по их именам, а по сигнатурам. Это позволяет записывать разные методы с одинаковыми именами, различающиеся числом и/или типами параметров. Тип возвращаемого значения не входит в сигнатуру метода, значит, методы не могут различаться только типом результата их работы. Например, в классе, обозначенном Automobile мы записали метод moveTo(int x, int y) , причем пункт его назначения отмечен географическими координатами. Можно определить еще метод moveTo (String destination) для указания географического названия пункта назначения и обращаться к нему так:

```
oka.moveTo("Москва") ;
```

Такое дублирование методов называется *перегрузкой* (overloading). Перегрузка методов очень удобна в использовании. В частности, в главе 1 на экран методом println() вывели данные любого типа, не заботясь о том, данные какого именно типа выводятся. На самом же деле под одним и тем же именем println были использованы разные методы t. Конечно, все эти методы надо тщательно спланировать и заранее описать в классе, что и было сделано в классе PrintStream, где представлено около двадцати методов print() и println() . Если же записать метод с тем же именем в подклассе, например:

```
class Truck extends Automobile{
void moveTo(int x, int y){
// Какие-то действия
}
// Что-то еще
}
```

то он перекроет метод суперкласса. Определив экземпляр класса Truck, например:

```
Truck gazel = new Truck();
```

и записав *gazel.moveTo(25, 150)*, обратимся к методу класса *Truck.* Произойдет *переопределение* (overriding) метода.

Переопределение методов приводит к интересным результатам. В классе Pet описан метод voice() . Переопределим его в подклассах и используем в классе chorus , как показано в листинге 3.

Листинг 3. Пример полиморфного метода

```
abstract class Pet {
   abstract void voice();
}
```

```
class Dog extends Pet {
   int k = 10;
   void voice() {
      System.out.println("Gav-gav!");
   }
}
class Cat extends Pet {
   void voice() {
      System.out.println("Miaou!");
   }
}
class Cow extends Pet {
   void voice() {
      System.out.println("Mu-u-u!");
   }
}
public class Chorus {
   public static void main(String[] args) {
      Pet[] singer = new Pet[3];
      singer[0] = new Dog();
      singer[1] = new Cat();
      singer[2] = new Cow();
      for (int i = 0; i < singer.length; i++)
         singer[i].voice();
   }
}
```

Все дело здесь в определении поля singer[]. Хотя массив ссылок singer [] имеет тип Pet , каждый его элемент ссылается на объект своего типа – Dog, Cat, Cow. При выполнении программы вызывается метод конкретного объекта, а не метод класса, которым определялось имя ссылки. Так в Java реализуется полиморфизм. При описании класса Pet в методе voice() не задается никакой полезный алгоритм, поскольку у всех животных совершенно разные голоса. В таких случаях записываем только заголовок метода и ставим после закрывающей список параметров скобки точку с запятой. Этот метод будет *абстрактным* (abstract), что необходимо указать компилятору модификатором abstract. Если класс содержит хоть один абстрактный метод, то создать его экземпляры, а тем более использовать их, не удастся. Такой класс становится *абстрактным,* что обязательно надо указать модификатором abstract. Если член класса не отмечен ни одним из модификаторов private, protected, public, то к нему по умолчанию осуществляется *пакетный доступ* (default access), а именно: к такому члену может обратиться любой метод любого класса из того же пакета. Пакеты ограничивают и доступ к классу целиком – если класс не помечен модификатором public , то все его члены, даже открытые (public), не будут видны из других пакетов.

Чтобы создать пакет в первой строке Java-файла с исходным кодом, надо просто записать в строку package имя, например package mypack. Тем самым создается пакет с указанным именем mypack, и все классы, записанные в этом файле, попадут в пакет mypack. Повторяя эту строку в начале каждого исходного файла, включаем в пакет новые классы. Имя подпакета уточняется именем пакета. Чтобы создать подпакет с именем, например subpack, следует в первой строке исходного файла написать: package mypack.subpack; и все классы этого файла и всех файлов с такой же первой строкой попадут в подпакет subpack пакета mypack. Можно создать и подпакет подпакета, написав, к примеру: package mypack.subpack.sub и т. д. сколько угодно раз.

Рассмотрим пример. Пусть имеется пять классов, размещенных в двух пакетах. В файле Base.java описаны три класса: inpi, Base и класс Derivedp1, расширяющий класс Base. Эти классы размещены в пакете p1. В классе Base определены переменные всех четырех типов доступа, а в методах f() классов inp1 и Derivedp1 сделана попытка доступа ко всем полям класса Base. Неудачные попытки отмечены комментариями. В комментариях помещены сообщения компилятора. Листинг 4 показывает содержимое этого файла.

*Листинг 4***.** Файл Base.java с описанием пакета p1

```
class Inp1 {
   public void f() {
      Base b = new Base();
//    b.priv = 1;   // "priv has private access in p1.Base"
      b.pack = 1;
      b.prot = 1;
      b.publ = 1;
   }
}
public class Base {
   private int priv = 0;
   int pack = 0;
   protected int prot = 0;
   public int publ = 0;
}
class Derivedp1 extends Base {
   public void f(Base a) {
// a.priv = 1; // "priv has private access in pi.Base"
      a.pack = 1;
      a.prot = 1;
      a.publ = 1;
//   priv = 1; // "priv has private access in pi.Base"
      pack = 1;
      prot = 1;
      publ = 1;
   }
}
```

Как видно из листинга 4, в пакете недоступны только закрытые, private, поля другого класса. В файле Inp2.java описаны два класса: inp2 и Derivedp2, расширяющий класс Base. Эти классы находятся в другом пакете p2. В этих классах тоже сделана попытка обращения к полям класса Base. Неудачные попытки прокомментированы сообщениями компилятора. Листинг 5 показывает содержимое этого файла. Напомним, что класс Base должен быть помечен при своем описании в пакете p1 модификатором public , иначе из пакета p2 не будет видно ни одного его члена.

Листинг 5. Файл Inp2.java с описанием пакета p2

```
package p2;
import p1.Base;
class Inp2{
public static void main(String[] args)
Base b = new Base();
// b.priv = 1;// "priv has private access in p1.Base"
// b.pack = 1;// "pack is not public in p1.Base"
// cannot be accessed from outside package"
// b.prot = 1;// "prot has protected access in p1.Base"
b.publ = 1;
}
}
class Derivedp2 extends Base{
public void, f (Base a){
// a.priv = 1;// "priv has private access in. p1.Base"
// a.pack = 1;// "pack is not public in p1.Base; cannot
//be accessed from outside package"
// a.prot = 1; // "prot has protected access in p1.Base"
a.publ = 1;
// priv = 1;// "priv has private access in p1.Base"
// pack = 1;// "pack is not public in p1.Base; cannot
// be accessed from outside package"
prot = 1;
publ = 1;
super.prot = 1;
 }
}
```

Из независимого класса можно обратиться только к открытым, public, полям класса другого пакета. Из подкласса можно обратиться еще и к защищенным, protected, полям, но только унаследованным непосредственно, а не через экземпляр суперкласса.

Импорт классов и пакетов. Правила использования оператора import очень просты: пишется слово import и через пробел – полное имя класса, завершенное точкой с запятой. Сколько классов надо указать, столько операторов import и пишется.

В нашем примере можно было написать import p1.*; Напомним, что импортировать можно только открытые классы, помеченные модификатором public. Пакет java.lang просматривается всегда, его необязательно импортировать. Остальные пакеты стандартной библиотеки надо указывать в операторах import, либо записывать полные имена классов. Подчеркнем, что оператор import вводится только для удобства программистов и слово "импортировать" не означает никаких перемещений классов.

Интерфейс – это конструкция языка Java. Интерфейс (interface) в отличие от класса содержит только константы и заголовки методов без их реализации. Интерфейсы размещаются в тех же пакетах и подпакетах, что и классы, и компилируются также в class-файлы.

Все константы и методы в интерфейсах всегда открыты, не надо даже указывать модификатор public.

Вот какую схему можно предложить для иерархии автомобилей:

interface Automobile{ . . . }
interface Car extends Automobile{ . . . }
interface Truck extends Automobile{ . . . }
interface Pickup extends Car, Truck{ . . . }

Таким образом, интерфейс — это только набросок, эскиз. В нем указано, что делать, но не указано, как это делать.

Вот как можно реализовать иерархию автомобилей:

interface Automobile{ . . . }
interface Car extends Automobile! . . . }
class Truck implements Automobile! . . . }
class Pickup extends Truck implements Car{ . . . }
или так:
interface Automobile{ . . . }
interface Car extends Automobile{ . . . }
interface Truck extends Automobile{ . . . }
class Pickup implements Car, Truck{ . . . }

Реализация интерфейса может быть неполной, некоторые методы интерфейса расписаны, а другие – нет . Такая реализация – абстрактный класс, его обязательно надо пометить модификатором abstract. Как реализовать в классе pickup метод f() , описанный и в интерфейсе car и в интерфейсе Truck с одинаковой сигнатурой? Ответ простой — никак. Такую ситуацию нельзя реализовать в классе Pickup . Программу надо спроектировать по-другому.

Итак, интерфейсы позволяют реализовать средствами Java чистое объектно-ориентированное проектирование, не отвлекаясь на вопросы реализации проекта. Приступая к разработке проекта, запишем его в виде иерархии интерфейсов, не думая о реализации, а затем построим по этому проекту иерархию классов, учитывая ограничения одиночного наследования и видимости членов классов при создании ссылок на интерфейсы. Такая ссылка может указывать только на какую-нибудь реализацию интерфейса. Тем самым получаем еще один способ организации полиморфизма.

Листинг 6. Использование интерфейса для организации полиморфизма

```
package p1;
interface Voice {
   void voice();
}
class Dog implements Voice {
   public void voice() {
      System.out.println("Gav-gav!");
   }
}
class Cat implements Voice {
   public void voice() {
      System.out.println("Miaou!");
   }
}
class Cow implements Voice {
   public void voice() {
      System.out.println("Mu-u-u!");
   }
}
public class Chorus {
   public static void main(String[] args) {
      Voice[] singer = new Voice[3];
      singer[0] = new Dog();
      singer[1] = new Cat();
      singer[2] = new Cow();
      for (int i = 0; i < singer.length; i++)
         singer[i].voice();
   }
}
```

Здесь используется интерфейс voice вместо абстрактного класса Pet.

Задание на самостоятельную работу.

Создать класс, описывающий структуру автомобиля. Добавить в класс методы, позволяющие получать данные полей класса. Создать интерфейс, содержащий логику хранения объектов в коллекции типа Stack. Создать основной класс, реализующий данный интерфейс для хранения объектов типа Автомобиль.

Контрольные вопросы

1. Для чего предназначена инкапсуляция?
2. Что такое суперкласс и подкласс?
3. Для чего предназначен модификатор final?
4. Для чего предназначены пакеты в Java? Перечислите стандартные пакеты и их предназначение.

Занятие 4. Коллекции Java

Цель: обретение навыков работы с коллекциями Java.

Теория и примеры решения задач

В классе Vector из пакета Java.util хранятся элементы типа object, а значит, любого типа. Количество элементов может быть любым и заранее не определенным. Элементы получают индексы 0, 1, 2, ... К каждому элементу вектора можно обратиться как по индексу, так и как к элементу массива.

В классе четыре конструктора: Vector() – создает пустой объект нулевой длины; Vector(int capacity) – создает пустой объект указанной емкости capacity; Vector (int capacity, int increment) – создает пустой объект указанной емкости capacity и задает число increment , на которое увеличивается емкость при необходимости; Vector (Collection c) – вектор создается по указанной коллекции. Если capacity отрицательно, создается исключительная ситуация. После создания вектора его можно заполнять элементами.

Метод add(Object element) позволяет добавить элемент в конец вектора (то же делает старый метод addElement(Object element). Методом add(int index, Object element) или старым методом insertElementAt(Object element, int index) можно вставить элемент в указанное место index. Элемент, находившийся на этом месте, и все последующие элементы сдвигаются, их индексы увеличиваются на единицу. Метод addAll(Collection coll) позволяет добавить в конец вектора все элементы коллекции coll. Метод set(int index, object element) заменяет элемент, стоявший в векторе в позиции index , на элемент element (то же позволяет выполнить старый метод setElementAt(Object element, int index). Количество элементов в векторе всегда можно узнать методом size(). Метод capacity возвращает емкость вектора. Логический метод isEmpty() возвращает true , если в векторе нет ни одного элемента. Обратиться к первому элементу вектора можно методом firstElement (), к последнему – методом lastElement (), к любому элементу – методом get(int index) или старым методом elementAt (int index). Эти методы возвращают объект класса object. Перед использованием его следует привести к нужному типу. Получить все элементы вектора в виде массива типа object [] можно методами toArray() и toArray(Object [] a). Второй метод заносит все элементы вектора в массив a, если в нем достаточно места. Логический метод contains(object element) возвращает true , если элемент element находится в векторе. Логический метод containsAll(Collection c) возвращает true , если вектор содержит все элементы указанной коллекции.

Четыре метода позволяют отыскать позицию указанного элемента element: indexOf(Object element) – возвращает индекс первого появления элемента в векторе. Логический метод remove(Object element) удаляет из вектора первое вхождение указанного элемента element. Метод возвращает true, если элемент найден и удаление произведено. Метод remove(int index) удаляет элемент из позиции index и возвращает его в качестве своего результата типа object. Аналогичные действия позволяют выполнить старые методы типа void

removeElement(Object element) и removeElementAt(int index) , не возвращающие результата.

Листинг 7. Работа с вектором

```
Vector v = new Vector();
String s = "Строка, которую мы хотим разобрать на слова.";
StringTokenizer st = new StringTokenizer(s, " \t\n\r,.");
while (st.hasMoreTokens()){
// Получаем слово и заносим в вектор
v.add(st.nextToken());   // Добавляем в конец вектора
}
System.out.println(v.firstElement());  // Первый элемент
System.out.println(v.lastElement());    // Последний элемент
v.setSize(4);                  // Уменьшаем число элементов
v.add("собрать.");                  // Добавляем в конец
// укороченного вектора
v.set(3, "опять");                  // Ставим в позицию 3
for (int i = 0; i < v.size(); i++)   // Перебираем весь вектор
System.out.print(v.get(i) + " ");
System.out.println();
```

Дополнительно к методам класса Vector класс Stack содержит пять методов, позволяющих работать с коллекцией, как со стеком:

push(Object item) – помещает элемент item в стек;

pop() – извлекает верхний элемент из стека;

peek() – читает верхний элемент, не извлекая его из стека;

empty() – проверяет, не пуст ли стек;

search(object item) – находит позицию элемента item в стеке. Верхний элемент занимает позицию 1, под ним элемент 2 и т. д. Если элемент не найден, возвращается элемент с позицией 1. Листинг 8 показывает, как можно использовать стек для проверки парности символов.

Листинг 8. Проверка парности скобок

```
import java.util.*;
class StackTest {
   static boolean checkParity(String expression,
                        String open, String close) {
      Stack Stack = new Stack();
      StringTokenizer st = new StringTokenizer(expression,
            " \t\n\r+*/-(){}", true);
      while (st.hasMoreTokens()) {
         String tmp = st.nextToken();
         if (tmp.equals(open)) Stack.push(open);
         if (tmp.equals(close)) Stack.pop();
      }
      if (Stack.isEmpty()) return true;
      return false;
```

```
    }
    public static void main(String[] args) {
        System.out.println(checkParity("a - (b - (c - a) / (b + c) - 2)", "(", ")"));
    }
}
```

Еще один пример коллекции совсем другого рода – таблицы – предоставляет класс Hashtable. Класс Hashtable расширяет абстрактный класс Dictionary . В объектах этого класса хранятся пары "ключ-значение". Каждый объект класса Hashtable кроме размера (size) и количества пар имеет еще две характеристики: емкость (capacity) – размер буфера и показатель загруженности (load factor) – процент заполненности буфера, по достижении которого увеличивается его размер. Для создания объектов класс Hashtable предоставляет четыре конструктора: Hashtable () – создает пустой объект с начальной емкостью в 101 элемент и показателем загруженности 0,75; Hashtable (int capacity) – создает пустой объект с начальной емкостью capacity и показателем загруженности 0,75; Hashtable(int capacity, float loadFactor) – создает пустой Объект с начальной емкостью capacity и показателем загруженности loadFactor; Hashtable (Map f) – создает объект класса Hashtable , содержащий все элементы отображения f, с емкостью, равной удвоенному числу элементов отображения f, но не менее 11, и показателем загруженности 0,75.

Для заполнения объекта класса Hashtable используется два метода: Object put(Object key, Object value) – добавляет пару " key-value ", если ключа key не было в таблице, и меняет значение value ключа key , если он уже есть в таблице. Возвращает старое значение ключа или null , если его не было. Если хотя бы один параметр равен null, возникает исключительная ситуация; void putAll(Map f) – добавляет все элементы отображения f . В объектах-ключах key должны быть реализованы методы hashCode() и equals ().

Метод get (Object key) возвращает значение элемента с ключом key в виде объекта класса object. Для дальнейшей работы его следует преобразовать в конкретный тип. Логический метод containsKey(object key) возвращает true , если в таблице есть ключ key. Логический метод containsvalue (Object value) или старый метод contains (object value) возвращают true, если в таблице есть ключи со значением value. Логический метод isEmpty () возвращает true , если в таблице нет элементов. Метод values () представляет все значения value таблицы в виде интерфейса Collection. Все модификации в объекте collection изменяют таблицу, и наоборот. Метод keyset () предоставляет все ключи key таблицы в виде интерфейса set. Все изменения в объекте set корректируют таблицу, и наоборот. Метод entrySet() представляет все пары " key-value " таблицы в виде интерфейса Set. Все модификации в объекте set изменяют таблицу, и наоборот. Метод toString () возвращает строку, содержащую все пары. Старые методы elements () и keys () возвращают значения и ключи в виде интерфейса Enumeration. Метод remove (Object key) удаляет пару с ключом key , возвращая значение этого ключа, если оно есть, и null , если пара с ключом key не найдена. Метод clear 0 удаляет все элементы, очищая таблицу.

В листинге 9 показано, как можно использовать класс Hashtabie для создания телефонного справочника.

Листинг 9. Телефонный справочник

```
import java.util.*;
class PhoneBook {
   public static void main(String[] args) {
      Hashtable yp = new Hashtable();
      String name = null;
      yp.put("John", "123-45-67");
      yp.put("Lemon", "567-34-12");
      yp.put("Bill", "342-65-87");
      yp.put("Gates", "423-83-49");
      yp.put("Batman", "532-25-08");
      try {
         name = args[0];
      } catch (Exception e) {
         System.out.println("Usage: Java PhoneBook Name");
         return;
      }
      if (yp.containsKey(name))
         System.out.println(name + "'s phone = " + yp.get(name));
      else
         System.out.println("Sorry, no such name");
   }
}
```

Задание на самостоятельную работу

Есть массив, в котором хранится информация об итогах сессии. Сведения о каждом студенте – это фамилия, номер группы и результаты экзаменов по трем дисциплинам. Вывести в алфавитном порядке по группам информацию о студентах в порядке убывания их средней успеваемости (Группа – Успеваемость – Фамилия). Задача подразумевает использование некоторой коллекции.

Контрольные вопросы

1. В чём преимущества коллекции по сравнению с массивом?
2. В чём недостатки коллекции по сравнению с массивом?
3. Перечислите основные методы класса Vector.
4. Перечислите основные методы класса Hashtable.

Занятие 5. Строки и система ввода-вывода Java

Цель: приобретение навыков работы со стоками и потоками ввода-вывода Java.

Теория и примеры решения задач

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 s1 = "Это строка."* Если константа длинная, можно записать ее в нескольких строках текстового редактора, связывая их операцией сцепления: *String s2 = "Это длинная строка, "* + "записанная в двух строках исходного текста".

Со строками можно производить операцию *сцепления строк* (concatenation), обозначаемую знаком плюс +. Эта операция создает новую строку, просто составленную из состыкованных первой и второй строк, как показано в начале данной главы. Ее можно применять и к константам, и к переменным. Например:

String attention = "Внимание: ";

String s = attention + "неизвестный символ";

Вторая операция – присваивание += – применяется к переменным в левой части:

attention += s;

Поскольку операция + перезагружена со сложения чисел на сцепление строк, встает вопрос о приоритете этих операций. У сцепления строк приоритет выше, чем у сложения, поэтому, записав "2" + 2 + 2 , получим строку " 222 ". Но записав 2 + 2 + "2" , получим строку "42", поскольку действия выполняются слева направо. Если же запишем "2" + (2 + 2) , то получим "24" .

В классе String есть множество методов для работы со строками. Для того чтобы узнать длину строки, т. е. количество символов в ней, надо обратиться к методу length() :

String s = "Write once, run anywhere.";

int len = s.length{);

или еще проще

int len = "Write once, run anywhere.".length(),

поскольку строка-константа – полноценный объект класса *String* . Заметьте, что строка – это не массив, у нее нет поля *length.* В массив будет записано end-begin символов, которые займут элементы массива, начиная с индекса ind до индекса ind + (end-begin) - 1. Этот метод при переводе символов из Unicode в ASCII использует локальную кодовую таблицу.

Если же надо получить массив байтов не в локальной кодировке, а в какой-то другой, используем метод getBytes(String encoding). Метод subString(int begin, int end) отделяет подстроку от символа с индексом begin включительно до символа с индексом end исключительно. Длина подстроки будет равна *end-begin.* Метод subString (int begin) выделяет подстроку от индекса begin включи-

тельно до конца строки. Если индексы отрицательны, индекс end больше длины строки или begin больше чем end, то возникает исключительная ситуация. Логический метод equals (object obj) , переопределенный из класса object , возвращает true , если аргумент obj не равен null; он является объектом класса String , и строка, содержащаяся в нем, полностью идентична данной строке вплоть до совпадения регистра букв. В остальных случаях возвращается значение false.

Логический метод equalsIgnoreCase(object obj) работает так же, но одинаковые буквы, записанные в разных регистрах, считаются совпадающими. Например, s2.equals ("другая строка") даст в результате false, а s2.equalsIgnore Case ("другая строка") возвратит true. Метод compareTo(String str) возвращает целое число типа int , вычисленное по следующим правилам. Сравниваются символы данной строки this и строки str с одинаковым индексом, пока не встретятся различные символы с индексом, допустим k , или пока одна из строк не закончится. В первом случае возвращается значение this.charAt(k)-str.charAt(k), т. е. разность кодировок Unicode первых несовпадающих символов. Во втором случае возвращается значение this.length()-str.length() , т. е. разность длин строк. Если строки совпадают, возвращается 0. Если значение str равно null , возникает исключительная ситуация. Нуль возвращается в той же ситуации, в которой метод equals() возвращает true. Метод compareToignoreCase(String str) производит сравнение без учета регистра букв, точнее говоря, выполняется метод

this.toUpperCase().toLowerCase().compareTo(
str.toUpperCase().toLowerCase());

Еще один метод – compareTo (Object obj) создает исключительную ситуацию, если obj не является строкой. В остальном он работает как метод compareTo(String str). Здесь ind1 – индекс начала подстроки данной строки this, ind2 – индекс начала подстроки другой строки str . Результат false получается в следующих случаях: хотя бы один из индексов ind1 или ind2 отрицателен; хотя бы одно из ind1 + len или ind2 + len больше длины соответствующей строки; хотя бы одна пара символов не совпадает.

Этот метод различает символы, записанные в разных регистрах. Если надо сравнивать подстроки без учета регистров букв, то используйте логический метод: *regionMatches(boolean flag, int ind1, String str, int ind2, int len).* Если первый параметр flag равен true , то регистр букв при сравнении подстрок не учитывается, если false — учитывается. Поиск всегда ведется с учетом регистра букв. Первое появление символа ch в данной строке this можно отследить методом indexOf(int ch) , возвращающим индекс этого символа в строке или -1 , если символа ch в строке this нет. Например, *"Молоко", indexOf('o')* выдаст в результате 1. Второе и следующие появления символа ch в данной строке this можно отследить методом *indexOf(int ch, int ind)* . Этот метод начинает поиск символа ch с индекса ind . Если ind < o, то поиск идет с начала строки, если ind больше длины строки, то символ не ищется, т. е. возвращается -1. Поиск всегда ведется с учетом регистра букв.

Первое вхождение подстроки sub в данную строку this отыскивает метод *indexof (String sub).* Он возвращает индекс первого символа первого вхождения

подстроки sub в строку или -1, если подстрока sub не входит в строку this . Например, *" Раскраска ".indexof ("рас")* даст в результате 4.

Если поиск начинается не с начала строки, а с какого-то индекса ind, используйте метод *indexOf (String sub, int ind).* Если *ind* < 0, то поиск идет с начала строки; если ind больше длины строки, то символ не ищется, т. е. возвращается -1. Последнее вхождение подстроки sub в данную строку this можно отыскать методом *lastIndexOf(String sub),* возвращающим индекс первого символа последнего вхождения подстроки sub в строку this или -1, если подстрока sub не входит в строку this. Перечисленные выше методы создают исключительную ситуацию, если *sub == null.* Если необходимо осуществить поиск, не учитывающий регистр букв, измените предварительно регистр всех символов строки. Метод *toLowerCase ()* возвращает новую строку, в которой все буквы переведены в нижний регистр, т. е. сделаны строчными. Метод *toUpperCase ()* возвращает новую строку, в которой все буквы переведены в верхний регистр, т. е. сделаны прописными. При этом используется локальная кодовая таблица по умолчанию. Если нужна другая локаль, то применяются методы *toLowerCase(Locale loc)* и *toUpperCase(Locale loc).* Метод *replace (int old, int new)* возвращает новую строку, в которой все вхождения символа old заменены символом new . Если символа old в строке нет, то возвращается ссылка на исходную строку. Например, после выполнения " *Рука в руку сует хлеб" , replace ('у', 'е')* получим строку *"Река в реке сеет хлеб".* Регистр букв при замене учитывается.

В языке Java принято соглашение – каждый класс отвечает за преобразование других типов в тип этого класса и должен содержать нужные для этого методы. Класс String содержит восемь статических методов valueOf (type elem) преобразования в строку примитивных типов boolean, char, int, long, float, double, массива char[] и просто объекта типа object. Девятый метод *valueOf(char[] ch, int offset, int len)* преобразует в строку подмассив массива ch , начинающийся с индекса offset и имеющий len элементов. Кроме того, в каждом классе есть метод toString () , переопределенный или просто унаследованный от класса Object . Он преобразует объекты класса в строку. Фактически, метод valueOf вызывает метод toString() соответствующего класса. Поэтому результат преобразования зависит от того, как реализован метод toString (). Еще один простой способ – сцепить значение elem какого-либо типа с пустой строкой: "" + elem. При этом неявно вызывается метод *elem. toString ().*

Создать объект класса StringBuf fer можно только конструкторами. В классе *StringBuffer* три конструктора:

StringBuffer () – создает пустой объект с емкостью 16 символов;

StringBuffer.(int capacity) – создает пустой объект заданной емкости capacity ;

StringBuffer (String str) – создает объект емкостью str . length () + 16, содержащий строку str **.**

В классе StringBuffer есть десять методов append (), добавляющих подстроку в конец строки. Они не создают новый экземпляр строки, а возвращают ссылку на ту же самую, но измененную строку.

Основной метод *append (String str)* присоединяет строку str в конец данной строки. Если ссылка *str == null*, то добавляется строка *"null"*.

Шесть методов *append (type elem)* добавляют примитивные типы *boolean, char, int, long, float, double,* преобразованные в строку.

Два метода присоединяют к строке массив str и подмассив sub символов, преобразованные в строку: *append (char [] str) И append (char [.] , sub, int offset, int len).*

Десятый метод добавляет просто объект *append (Object obj).* Перед этим объект obj преобразуется в строку своим методом *toString ().* Десять методов insert () предназначены для вставки строки, указанной параметром метода, в данную строку. Место вставки задается первым параметром метода ind . Это индекс элемента строки, перед которым будет сделана вставка. Он должен быть неотрицательным и меньше длины строки, иначе возникнет исключительная ситуация. Строка раздвигается, емкость буфера при необходимости увеличивается. Методы возвращают ссылку на ту же преобразованную строку. Основной метод *insert (int ind, String str)* вставляет строку str в данную строку перед ее символом с индексом and . Если ссылка *str == null,* вставляется строка *"null".*

Метод *delete (int begin, int end)* удаляет из строки символы начиная с индекса begin включительно, до индекса end исключительно; если end больше длины строки, то до конца строки. Например, после выполнения *String s = new StringBuffer("Это небольшая строка").delete(4, 6).toString();* получим *s == "Это большая строка".* Если begin отрицательно, больше длины строки или больше end, возникает исключительная ситуация. Если begin == end, удаление не происходит. Метод *replace (int begin, int end. String str)* удаляет символы из строки начиная с индекса begin включительно, до индекса end исключительно, если end больше длины строки, то до конца строки, и вставляет вместо них строку str.Если begin отрицательно, больше длины строки или больше end , возникает исключительная ситуация. Разумеется, метод *replace ()* – это последовательное выполнение методов *delete () и insert ().*

Задача разбора введенного текста – *парсинг* (parsing) – вечная задача программирования наряду с сортировкой и поиском. Написана масса программ-парсеров (parser), разбирающих текст по различным признакам. Есть даже программы, генерирующие парсеры по заданным правилам разбора: YACC, LEX и др. Но задача остается. И вот очередной программист, отчаявшись найти что-нибудь подходящее, берется за разработку собственной программы разбора. В пакет *java.util* входит простой класс *StringTokenizer* , облегчающий разбор строк.

Класс *StringTokenizer* из пакета *java.util* небольшой, в нем три конструктора и шесть методов. Первый конструктор *StringTokenizer (String str)* создает объект, готовый разбить строку str на слова, разделенные пробелами, символами табуляций '\t', перевода строки '\n' и возврата каретки '\r'. Разделители не включаются в число слов. Второй конструктор *StringTokenizer (String str. String delimeters)* задает разделители вторым параметром *delimeters*, например: *StringTokenizer("Казнить,нельзя:пробелов-нет", " \t\n\r,:-").*Здесь первый разде-

литель – пробел. Потом идут символ табуляции, символ перевода строки, символ возврата каретки, запятая, двоеточие, дефис. Порядок расположения разделителей в строке delimeters не имеет значения. Разделители не включаются в число слов. Третий конструктор позволяет включить разделители в число слов: *StringTokenizer(String str, String delimeters, boolean flag);* Если параметр flag равен true , то разделители включаются в число слов, если false — нет. Например: *StringTokenizer("a - (Ь + c) / Ь * c", " \t\n\r+*-/(), true;*

В разборе строки на слова активно участвуют два метода: *метод nextToken()* возвращает в виде строки следующее слово; логический метод *hasMoreTokens()* возвращает true , если в строке еще есть слова, и false , если слов больше нет. Третий метод *countTokens()* возвращает число оставшихся слов. Четвертый метод *nextToken(String newDelimeters)* позволяет "на ходу" менять разделители. Следующее слово будет выделено по новым разделителям newDelimeters; новые разделители действуют далее вместо старых разделителей, определенных в конструкторе или предыдущем методе *nextToken().* Оставшиеся два метода *nextElement ()* и *hasMoreElements ()* реализуют интерфейс *Enumeration.* Они просто обращаются к методам *nextToken ()* и *hasMoreTokens().*

Прежде чем перейти к классам, которые действительно читают и записывают данные в поток, мы посмотрим на утилиты, обеспечивающиеся библиотекой при обработке директории файлов.

Предположим, необходимо получить список директории. Объект File может выдать его двумя способами. Если вызовите list() без аргументов, то получите полный список, содержащийся в объекте File. Однако если нужно ограничить список, например, чтобы получить все файлы с расширением java, используется “фильтр директории”, который является классом, который определяет, как использовать объект File для отображения.

DirFilter показывает, что из-за того что interface содержит только набор методов, нет ограничения в написании только этих методов. (Однако необходимо, как минимум, обеспечить определение для всех методов интерфейса.) В этом случае также создается конструктор DirFilter.

Метод accept() должен принимать объект File, представляющий директорий, в котором находится определенный файл, а String содержит имя этого файла. Можно выбрать: использовать или игнорировать любой из этих аргументов, но вам, вероятно, как минимум, нужно использовать имя файла. Помните, что метод list() вызывает метод accept() для каждого имени файла в директории, чтобы проверить, какой из них должен быть включен, – на это указывает тип boolean результата, возвращаемого accept().

Чтобы убедиться в том, что рабочий элемент является всего лишь именем файла и не содержит информации о пути, нужно получить объект String и создать из него объект File, затем вызвать getName(), который отсекает всю информацию о пути (платформонезависимым способом). Затем accept() использует метод indexOf() класса String, чтобы убедится, что искомая строка afn присутствует в любом месте имени файла. Если afn найдено в строке, возвращаемым значением является начальный индекс afn, а если не найдено, возвращаемым

значением является -1. Напомним, что это простой поиск строк, который не имеет «глобальных» выражений подстановочных символов, таких как fo?.b?r*”, являющихся более сложными в реализации.

Метод list() возвращает массив. Можно опросить этот массив о его длине, а затем пройтись по нему, выбирая элементы массива. Эта способность легкого прохода по массиву вне методов и в методах является значительным улучшением по сравнению с поведением C и C++.

Задание на самостоятельную работу

Написать программу, которая считывает текстовый файл большого объема и выводит статистику по буквам – сколько раз каждая буква встречается в файле.

Контрольные вопросы

1. Перечислите способы манипуляции со строками?
2. Является ли строка изменяемым объектом? Почему?
3. Для чего предназначены классы StringBuffer и StringTokenizer?
4. Перечислите основные классы для работы с вводом-выводом в java и их предназначение.

Занятие 6. Многопоточность в Java

Цель: приобретение навыков создания многопоточных приложений в Java.

Теория и примеры решения задач

Работу многозадачной системы можно упростить и ускорить, если разрешить взаимодействующим процессам работать в одном адресном пространстве. Такие процессы называются threads. В русской литературе предлагаются различные переводы этого слова. Буквальный перевод – «нить». Часто переводят thread как «поток», но здесь разговор идет о потоке ввода/вывода. Иногда просто говорят «тред», но в русском языке уже есть «тред-юнион». Встречается перевод «легковесный процесс», но в некоторых операционных системах, например Solaris, есть и thread, и lightweight process. Остановимся на слове «подпроцесс».

Главным подпроцессом апплета является один из подпроцессов браузера, в котором апплет выполняется. Главный подпроцесс не играет никакой особой роли, просто он создается первым. Подпроцесс в Java создается и управляется методами класса Thread. После создания объекта этого класса одним из его конструкторов новый подпроцесс запускается методом start ().

В классе Thread семь конструкторов: Thread (ThreadGroup group, Runnable target, String name) – создает подпроцесс с именем name, принадлежащий группе group и выполняющий метод run() объекта target. Это основной конструктор, все остальные обращаются к нему с тем или иным параметром, равным null; Thread() – создаваемый подпроцесс будет выполнять свой метод run (); Thread(Runnable target); Thread (Runnable target, String name); Thread(String name); Thread (ThreadGroup group, Runnabie target); Thread (ThreadGroup group, String name).

Имя подпроцесса name не имеет никакого значения, оно не используется виртуальной машиной Java и применяется только для различения подпроцессов в программе. После создания подпроцесса его надо запустить методом start(). Виртуальная машина Java начнет выполнять метод run() этого объекта-подпроцесса. Подпроцесс завершит работу после выполнения метода run (). Для уничтожения объекта-подпроцесса вслед за этим он должен присвоить значение null. Выполняющийся подпроцесс можно приостановить статическим методом sleep(long ms) на ms миллисекунд. Если вычислительная система может отсчитывать наносекунды, то можно приостановить подпроцесс с точностью до наносекунд методом sleep(long ms, int nanosec). Основная сложность при написании программ, в которых работают несколько подпроцессов, – это согласовать совместную работу подпроцессов с общими ячейками памяти.

Классический пример – банковская транзакция, в которой изменяется остаток на счету клиента с номером numDep. Предположим, что для ее выполнения запрограммированы такие действия:

Deposit myDep = getDeposit(numDep);// Получаем счет с номером numDep
int rest = myDep.getRest();// Получаем остаток на счету myDep

Deposit newDep = myDep.operate(rest, sum); // Изменяем остаток
// на величину sum
myDep.setDeposit(newDep); // Заносим новый остаток на счет myDep

В этом примере первый подпроцесс должен вначале заблокировать счет myDep, затем полностью выполнить всю транзакцию и снять блокировку. Второй подпроцесс приостановится и станет ждать, пока блокировка не будет снята, после чего начнет работать с объектом myDep.

Все это делается одним оператором synchronized () {}, как показано ниже:

```
Deposit myDep = getDeposit(numDep);
synchronized(myDep){
int rest = myDep.getRest();
Deposit newDep = myDep.operate(rest, sum);
myDep.setDeposit(newDep);
}
```

В заголовке оператора synchronized в скобках указывается ссылка на объект, который будет заблокирован перед выполнением блока. Объект будет недоступен для других подпроцессов, пока выполняется блок. После выполнения блока блокировка снимается.

Если при написании какого-нибудь метода оказалось, что в блок synchronized входят все операторы этого метода, то можно просто пометить метод квантификатором synchronized, сделав его *синхронизированным* (synchronized):

```
synchronized int getRest()(
// Тело метода
}
synchronized Deposit operate(int rest, int sum) {
// Тело метода
}
synchronized void setDeposit(Deposit dep){
// Тело метода
}
```

Многие методы Java 2 SDK синхронизированы. Обратите внимание, что часто слова выводятся вперемешку, но каждое слово выводится полностью. Это происходит потому, что метод print() класса PrintStream синхронизирован, при его выполнении выходной поток system out блокируется до тех пор, пока метод print() не закончит свою работу.

Итак, легко организовать последовательный доступ нескольких подпроцессов к полям одного объекта с помощью оператора synchronized () {}. Синхронизация обеспечивает *взаимно исключающее* (mutually exclusive) выполнение подпроцессов. Но что делать, если нужен совместный доступ нескольких подпроцессов к общим объектам? Для этого в Java существует механизм ожидания и уведомления (wait-notify).

Возможность создания многопоточных программ заложена в язык Java ещё в процессе самого его создания. В каждом объекте есть три метода wait() и

один метод notify(), позволяющие приостановить работу подпроцесса с этим объектом, позволить другому подпроцессу поработать с объектом, а затем уведомить (notify) первый подпроцесс о возможности продолжения работы. Эти методы определены прямо в классе object и наследуются всеми классами. Отличие данного метода от метода sleep() в том, что метод wait() снимает блокировку с объекта. С объектом может работать один из подпроцессов из «зала ожидания», обычно тот, который ждал дольше всех, хотя это не гарантируется спецификацией JLS. Второй метод wait() эквивалентен wait(0). Третий метод wait(long millisec, int nanosec) уточняет задержку на nanosec наносекунды, если их сумеет отсчитать операционная система. Метод notify () выводит из «зала ожидания» только один произвольно выбранный подпроцесс. Метод notifyAll() выводит из состояния ожидания все подпроцессы. Эти методы тоже должны выполняться в синхронизированном блоке или методе.

Задание на самостоятельную работу

Написать приложение, осуществляющее перемножение двух матриц. Организовать несколько потоков по количеству строк первой матрицы ($cij = \sum_k aik * bkj$). Отображать на консоли номера работающих потоков.

Контрольные вопросы

1. Что такое контекст выполнения?
2. Перечислите конструкторы класса Thread и их предназначение.
3. Что такое критический участок?
4. При каких условиях корректно вызывать метод wait()?

Занятие 7. Сетевые возможности Java

Цель: приобретение навыков работы с сокетами в Java.

Теория и примеры решения задач. Программы-серверы, прослушивающие свои порты, работают под управлением операционной системы. У машин-серверов могут быть самые разные операционные системы, особенности которых передаются программам-серверам. Чтобы сгладить различия в реализациях разных серверов, между сервером и портом введен промежуточный программный слой, названный сокетом (socket). К сокету может присоединиться любой клиент, лишь бы он работал по тому же протоколу, что и сервер. Каждый сокет связан (bind) с одним портом, говорят, что сокет прослушивает (listen) порт. Соединение с помощью сокетов устанавливается так: 1) сервер создает сокет, прослушивающий порт сервера; 2) клиент также 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. Чаще всего применяется конструктор Socket(String host, int port). Многочисленные методы доступа устанавливают и получают параметры сокета. В том числе методы, создающие потоки ввода/вывода: getInputStream() – возвращает входной поток типа InputStream; getOutputStream() – возвращает выходной поток типа OutputStream.

Приведем 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: 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. И наобарот, закрытие сокета закрывает и потоки. Для создания сервера в пакете java.net есть класс Serversocket. В конструкторе этого класса указывается номер порта ServerSocket(int port). Основной метод этого класса accept () ожидает поступления запроса. Когда запрос получен, метод устанавливает соединение с клиен-

том и возвращает объект класса socket, через который сервер будет обмениваться информацией с клиентом. Программы, реализующие стек протоколов TCP/IP, всегда создают так называемую «петлю» с адресом 127.0.0.1 и доменным именем localhost. Это адрес самого компьютера. Он используется для отладки приложений «клиент-сервер» пользуясь этим адресом можно запускать клиент и сервер на одной машине.

Класс содержит массу методов доступа к параметрам сокета и, кроме того, методы отправки и приема дейтаграмм: send(DatagramPacket pack) – отправляет дейтаграмму, упакованную в пакет pack; receive (DatagramPacket pack) – дожидается получения дейтаграммы и заносит ее в пакет pack. При обмене дейтаграммами соединение обычно не устанавливается, дейтаграммы посылаются наудачу, в расчете на то, что получатель ожидает их. Но можно установить соединение методом connect(InetAddress addr, int port). При этом устанавливается только одностороннее соединение с хостом по адресу addr и номером порта port – или на отправку, или на прием дейтаграмм. Потом соединение можно разорвать методом disconnect(). При посылке дейтаграммы по протоколу JJDP сначала создается сообщение в виде массива байтов, например:

String mes = "This is the sending message.";

byte[] data = mes.getBytes();

Потом записывается адрес – объект класса inetAddress, например: InetAddress addr = InetAddress.getByName (host); затем сообщение упаковывается в пакет – объект класса DatagramPacket. При этом указывается массив данных, его длина, адрес и номер порта: DatagramPacket pack = new DatagramPacket(data, data.length, addr, port). Далее создается дейтаграммный сокет DatagramSocket ds = new DatagramSocket() и дейтаграмма отправляется ds.send(pack). После посылки всех дейтаграмм сокет закрывается, не дожидаясь какой-либо реакции со стороны получателя: ds.close ().

Прием и распаковка дейтаграмм производится в обратном порядке, вместо метода send () применяется метод receive (DatagramPacket pack).

Задание на самостоятельную работу

Написать чат «сервер — много клиентов». Можно в консольном варианте, также можно использовать дейтаграммные сокеты. Не забывайте обрабатывать исключительные ситуации.

Контрольные вопросы

1. Опишите архитектуру «клиент-сервер».
2. Как осуществляется ввод-вывод в Socket()?
3. Что такое серверный сокет?
4. Чем отличается работа по протоколу TCP от работы по протоколу UDP?

Литература

1. Эккель, Б. Философия Java: Библиотека программиста / Б. Эккель. – СПб. : Питер, 2001. – 880 с.
2. Ноутон, П. Java 2 / П. Ноутон; пер. с англ. – СПб. : BHV-Санкт-Петербург, 2001. – 1072 с.
3. Симкин, С. Программирование на JAVA: путеводитель / С. Симкин; пер. с англ. – Киев : DiaSoft, 1996. – 736 с.
4. Вебер, Д. Технология Java в подлиннике / Д. Вебер; пер. с англ. – СПб. : BHV-Санкт-Петербург, 1997. – 1104 с.

Св. план 2010, поз. 20

Учебное издание

Байдаков Иван Владимирович
Яшин Константин Дмитриевич

ПСИХОЛОГИЯ ВЗАИМОДЕЙСТВИЯ ЧЕЛОВЕКА С ВИРТУАЛЬНОЙ РЕАЛЬНОСТЬЮ

Методическое пособие
к практическим занятиям по современным технологиям программирования
для студентов специальности 1-58 01 01
«Инженерно-психологическое обеспечение информационных технологий»
всех форм обучения

Редактор Т. П. Андрейченко
Корректор И. П. Острикова
Компьютерная верстка М. В. Гуртатовская

Подписано в печать. Формат 60x84 1/16. Бумага офсетная.
Гарнитура «Таймс». Отпечатано на ризографе. Усл. печ. л.
Уч.-изд. л. 2,0. Тираж 70 экз. Заказ 321.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 03.04.2009.
220013, Минск, П. Бровки, 6