МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
Учреждение образования
«Гомельский государственный технический университет имени П.О. Сухого»

Кафедра «Информационные технологии»

Курс лекций
старшего преподавателя Стефановского И.Л. по
дисциплине

«ВИЗУАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНЫХ ПРИЛОЖЕНИЙ»

для специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)»

Гомель 2024

Содержание

Раздел 1. ОРГАНИЗАЦИЯ КЛИЕНТ-СЕРВЕРНОЙ ОБРАБОТКИ ДАННЫХ

Тема 1.1. Многопоточное программирование в Java.

Параллельное программирование, связанное с использованием легковесных процессов, или подпроцессов (multithreading, light-weight processes) — концептуальная парадигма, в которой вы разделяете свою программу на два или несколько процессов, которые могут исполняться одновременно.

В системах без параллельных подпроцессов используется подход, называемый циклом обработки событий. В этой модели единственный подпроцесс выполняет бесконечный цикл, проверяя и обрабатывая возникающие события. Синхронизация между различными частями программы происходит в единственном цикле обработки событий. Такие среды называют синхронными управляемыми событиями системами. Apple Macintosh, Microsoft Windows, X11/Motif — все эти среды построены на модели с циклом обработки событий.

Если вы можете разделить свою задачу на независимо выполняющиеся подпроцессы и можете автоматически переключаться с одного подпроцесса, который ждет наступления события, на другой, которому есть чем заняться, за тот же промежуток времени вы выполните больше работы. Вероятность того, что больше чем одному из подпроцессов одновременно надолго потребуется процессор, мала.

Модель легковесных процессов в Java

Исполняющая система Java в многом зависит от использования подпроцессов, и все ее классовые библиотеки написаны с учетом особенностей программирования в условиях параллельного выполнения подпроцессов. Java использует подпроцессы для того, чтобы сделать среду программирования асинхронной. После того, как подпроцесс запущен, его выполнение можно временно приостановить (suspend). Если подпроцесс остановлен (stop), возобновить его выполнение невозможно. Приоритеты подпроцессов

Приоритеты подпроцессов — это просто целые числа в диапазоне от 1 до 10 и имеет смысл только соотношения приоритетов различных подпроцессов. Приоритеты же используются для того, чтобы решить, когда нужно остановить один подпроцесс и начать выполнение другого. Это называется *переключением контекста.* Правила просты. Подпроцесс может добровольно отдать управление — с помощью явного системного вызова или при блокировании на операциях ввода-вывода, либо он может быть приостановлен принудительно. В первом случае проверяются все остальные подпроцессы, и управление передается тому из них, который готов к выполнению и имеет самый высокий приоритет. Во втором случае, низкоприоритетный подпроцесс, независимо от

того, чем он занят, приостанавливается принудительно для того, чтобы начал выполняться подпроцесс с более высоким приоритетом.

Синхронизация

Поскольку подпроцессы вносят в ваши программы асинхронное поведение, должен существовать способ их синхронизации. Для этой цели в Java реализовано элегантное развитие старой модели синхронизации процессов с помощью *монитора.*

Сообщения

Коль скоро вы разделили свою программу на логические части - подпроцессы, вам нужно аккуратно определить, как эти части будут общаться друг с другом. Java предоставляет для этого удобное средство — два подпроцесса могут “общаться” друг с другом, используя методы wait и notify. Работать с параллельными подпроцессами в Java несложно. Язык предоставляет явный, тонко настраиваемый механизм управления созданием подпроцессов, переключения контекстов, приоритетов, синхронизации и обмена сообщениями между подпроцессами.

Подпроцесс

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

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

В этом примере текущий подпроцесс хранится в локальной переменной t. Затем мы используем эту переменную для вызова метода setName, который

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

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

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

Не очень интересно работать только с одним подпроцессом, а как можно создать еще один? Для этого нам понадобится другой экземпляр класса Thread. При создании нового объекта Thread ему нужно указать, какой программный код он должен выполнять. Вы можете запустить подпроцесс с помощью любого объекта, реализующего интерфейс Runnable. Для того, чтобы реализовать этот интерфейс, класс должен предоставить определение метода run. Ниже приведен пример, в котором создается новый подпроцесс.

```
class ThreadDemo implements Runnable {
ThreadDemo() {
Thread ct = Thread.currentThread();
System.out.println("currentThread: " + ct);
Thread t = new Thread(this, "Demo Thread");
System.out.println("Thread created: " + t); t.start();
try {
Thread.sleep(3000);
}
catch (InterruptedException e) {
System.out.println("interrupted");
```

```
}
System.out.println("exiting main thread");
}
public void run() {
try {
for (int i = 5; i > 0; i--) {
System.out.println("" + i);
Thread.sleep(1000);
} }
catch (InterruptedException e) {
System.out.println("child interrupted");
}
System.out.println("exiting child thread");
}
public static void main(String args[]) {
new ThreadDemo();
} }
```

Обратите внимание на то, что цикл внутри метода run выглядит точно так же, как и в предыдущем примере, только на этот раз он выполняется в другом подпроцессе. Подпроцесс main с помощью оператора new Thread(this, "Demo Thread") создает новый объект класса Thread, причем первый параметр конструктора — this — указывает, что нам хочется вызвать метод run текущего объекта. Затем мы вызываем метод start, который запускает подпроцесс, выполняющий метод run. После этого основной подпроцесс (main) переводится в состояние ожидания на три секунды, затем выводит сообщение и завершает работу. Второй подпроцесс — “Demo Thread” — при этом по-прежнему выполняет итерации в цикле метода run до тех пор пока значение счетчика цикла не уменьшится до нуля. Ниже показано, как выглядит результат работы этой программы этой программы после того, как она отработает 5 секунд.

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

Приоритеты подпроцессов

Если вы хотите добиться от Java предсказуемого независимого от платформы поведения, вам следует проектировать свои подпроцессы таким образом, чтобы они по своей воле освобождали процессор. Ниже приведен

пример с двумя подпроцессами с различными приоритетами, которые не ведут себя одинаково на различных платформах. Приоритет одного из подпроцессов с помощью вызова setPriority устанавливается на два уровня выше Thread. NORM_PRIORITY, то есть, умалчиваемого приоритета. У другого подпроцесса приоритет, наоборот, на два уровня ниже. Оба этих подпроцесса запускаются и работают в течение 10 секунд. Каждый из них выполняет цикл, в котором увеличивается значение переменной-счетчика. Через десять секунд после их запуска основной подпроцесс останавливает их работу, присваивая условию завершения цикла while значение true и выводит значения счетчиков, показывающих, сколько итераций цикла успел выполнить каждый из подпроцессов.

```
class Clicker implements Runnable {
int click = 0; private Thread t;
private boolean running = true;
public clicker(int p) {
t = new Thread(this);
t.setPriority(p); }
public void run() { while
(running) {
click++;
} }
public void stop() {
running = false; } public
void start() { t.start();
} } class
HiLoPri {
public static void main(String args[]) {
Thread.currentThread().setPriority(Thread.MAX_PRIORITY); clicker
hi = new clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
lo.start(); hi.start(); try Thread.sleep(-10000) {
}
catch (Exception e) {
} lo.stop();
hi.stop();
System.out.println(lo.click + " vs. " + hi.click);
} }
```

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

C:\>java HiLoPri 304300
vs. 4066666

Синхронизация

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

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

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

new Caller(target, "World");
}
}

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

[Hello.
[Synchronized
]
[World
]]

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

Взаимодействие подпроцессов

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

- wait — приводит к тому, что текущий подпроцесс отдает управление и переходит в режим ожидания — до тех пор пока другой под-процесс не вызовет метод notify с тем же объектом.
- notify — выводит из состояния ожидания первый из подпроцессов, вызвавших wait с данным объектом.
- notifyAll — выводит из состояния ожидания все подпроцессы, вызвавшие wait с данным объектом.

Ниже приведен пример программы с наивной реализацией проблемы поставщик-потребитель. Эта программа состоит из четырех простых классов: класса Q, представляющего собой нашу реализацию очереди, доступ к которой мы пытаемся синхронизовать; поставщика (класс Producer), выполняющегося

в отдельном подпроцессе и помещающего данные в очередь; потребителя (класс Consumer), тоже представляющего собой подпроцесс и извлекающего данные из очереди; и, наконец, крохотного класса PC, который создает по одному объекту каждого из перечисленных классов. *class Q { int n;*

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

Хотя методы put и get класса Q синхронизованы, в нашем примере нет ничего, что бы могло помешать поставщику переписывать данные по того, как их получит потребитель, и наоборот, потребителю ничего не мешает многократно считывать одни и те же данные. Так что вывод программы

содержит вовсе не ту последовательность сообщений, которую нам бы хотелось иметь:

```
C:\> java PC
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7 Got:
7
```

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

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

Q.

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

```
notify(); return
n;
}
synchronized void put(int n) {
if (valueSet)
try wait(); catch(InterruptedException e);
this.n = n; valueSet = true;
System.out.println("Put: " + n);
notify(); }
}
```

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

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

Клинч (deadlock)

Клинч — редкая, но очень трудноуловимая ошибка, при которой между двумя легковесными процессами существует кольцевая зависимость от пары синхронизированных объектов. Например, если один подпроцесс получает управление объектом X, а другой — объектом Y, после чего X пытается вызвать любой синхронизированный метод Y, этот вызов, естественно блокируется. Если при этом и Y попытается вызвать синхронизированный метод X, то программа с такой структурой подпроцессов окажется заблокированной навсегда. В самом деле, ведь для того, чтобы один из подпроцессов захватил нужный ему объект, ему нужно снять свою блокировку, чтобы второй подпроцесс мог завершить работу.

Сводка функций программного интерфейса легковесных процессов

Ниже приведена сводка всех методов класса Thread, обсуждавшихся в этой лекции.

Методы класса

Методы класса — это статические методы, которые можно вызывать непосредственно с именем класса Thread. currentThread

Статический метод currentThread возвращает объект Thread, выполняющийся в данный момент. yield

Вызов метода yield приводит к тому, что исполняющая система переключает контекст с текущего на следующий доступный подпроцесс. Это один из способов гарантировать, что низкоприоритетные подпроцессы когда-нибудь получат управление. sleep(int n)

При вызове метода sleep исполняющая система блокирует текущий подпроцесс на n миллисекунд. После того, как этот интервал времени закончится, подпроцесс снова будет способен выполняться. В большинстве исполняющих систем Java системные часы не позволяют точно выдерживать паузы короче, чем 10 миллисекунд. Методы объекта

start

Метод start говорит исполняющей системе Java, что необходимо создать системный контекст подпроцесса и запустить этот подпроцесс. После вызова этого метода в новом контексте будет вызван метод run вновь созданного подпроцесса. Вам нужно помнить о том, что метод start с данным объектом можно вызвать только один раз. run

Метод run — это тело выполняющегося подпроцесса. Это — единственный метод интерфейса Runnable. Он вызывается из метода start после того, как исполняющая среда выполнит необходимые операции по инициализации нового подпроцесса. Если происходит возврат из метода run, текущий подпроцесс останавливается. stop

Вызов метода stop приводит к немедленной остановке подпроцесса. Это — способ мгновенно прекратить выполнение текущего подпроцесса, особенно если метод выполняется в текущем подпроцессе. В таком случае строка, следующая за вызовом метода stop, никогда не выполняется, поскольку контекст подпроцесса “умирает” до того, как метод stop возвратит управление. Более аккуратный способ остановить выполнение подпроцесса — установить значение какой-либо переменной-флага, предусмотрев в методе run код, который, проверив состояние флага, завершил бы выполнение подпроцесса. suspend

Метод suspend отличается от метода stop тем, что метод приостанавливает выполнение подпроцесса, не разрушая при этом его системный контекст. Если выполнение подпроцесса приостановлено вызовом suspend, вы можете снова активизировать этот подпроцесс, вызвав метод resume. resume

Метод resume используется для активизации подпроцесса, приостановленного вызовом suspend. При этом не гарантируется, что после вызова resume подпроцесс немедленно начнет выполняться, поскольку в этот момент может выполняться другой более высокоприоритетный процесс. Вызов resume лишь делает подпроцесс способным выполняться, а то, когда ему будет передано управление, решит планировщик. setPriority(int p)

Метод setPriority устанавливает приоритет подпроцесса, задаваемый целым значением передаваемого методу параметра. В классе Thread есть несколько предопределенных приоритетов-констант: MIN_PRIORITY,

NORM_PRIORITY и MAX_PRIORITY, соответствующих соответственно значениям 1, 5 и 10. Большинство пользовательских приложений должно выполняться на уровне NORM_PRIORITY плюс-минус 1. Приоритет фоновых заданий, например, сетевого ввода-вывода или перерисовки экрана, следует устанавливать в MIN_PRIORITY. Запуск подпроцессов на уровне MAX_PRIORITY требует осторожности. Если в подпроцессах с таким уровнем приоритета отсутствуют вызовы sleep или yield, может оказаться, что вся исполняющая система Java перестанет реагировать на внешние раздражители. SetPriority

Этот метод возвращает текущий приоритет подпроцесса — целое значение в диапазоне от 1 до 10.

setName(String name)

Метод setName присваивает подпроцессу указанное в параметре имя. Это помогает при отладке программ с параллельными подпроцессами. Присвоенное с помощью setName имя будет появляться во всех трассировках стека, которые выводятся при получении интерпретатором неперехваченного исключения. getName

Метод getName возвращает строку с именем подпроцесса, установленным с помощью вызова setName.

Есть еще множество функций и несколько классов, например, ThreadGroup и SecurityManager, которые имеют отношение к подпроцессам, но эти области в Java проработаны еще не до конца. Скажем лишь, что при необходимости можно получить информацию об этих интерфейсах из документации по JDK API.

А дорога дальше вьется

Простые в использовании встроенные в исполняющую среду и в синтаксис Java легковесные процессы — одна из наиболее веских причин, по которым стоит изучать этот язык. Освоив однажды параллельное программирование, вы уже никогда не захотите возвращаться назад к программированию с помощью модели, управляемой событиями. После того, как вы освоились с основами программирования на Java, включая создание классов, пакетов, и модель легковесных процессов, для вас не составит труда разобраться в той коллекции Java-классов, к обсуждению которой мы сейчас приступим.

Тема 1.2 Организация клиент-серверной обработки данных

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

InetAddress

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

Фабричные методы

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

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

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

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

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

Дейтаграммы

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

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

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

Сокеты “для клиентов”

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

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

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

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

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

После того, как объект Socket создан, им можно воспользоваться для того, чтобы получить доступ к связанным с ним входному и выходному потокам. Эти потоки используются для приема и передачи данных точно так же, как и обычные потоки ввода-вывода, которые мы видели в предыдущей лекции: • getInputStream() возвращает InputStream, связанный с данным объектом. • getOutputStream() возвращает OutputStream, связанный с данным объектом. • close() закрывает входной и выходной потоки объекта Socket.

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

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

Сокеты “для серверов”

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

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

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

URL

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

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

URL(String spec)

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

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

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

URL(URL context, String spec)

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

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

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

URLConnection

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

```
import java.net.*;
import java.io.*; class
localURL {
public static void main(String args[]) throws Exception { int
c;
URL hp = new URL("http", "127.0.0.1", 80, "/");
URLConnection hpCon = hp.openConnection();
System.out.println("Date: " + hpCon.getDate());
System.out.println("Type: " + hpCon.getContentType());
System.out.println("Exp: " + hpCon.getExpiration());
System.out.println( "Last M: " + hpCon.getLastModified());
System.out.println("Length: " + hpCon.getContentLength()); if
(hpCon.getContentLength() > 0) { System.out.println("===
Content ==="); InputStream input =
hpCon.getInputStream(); int i=hpCon.getContentLength();
while (((c = input. read()) != -1) && (--i > 0)) {
```

```
System.out.print((char) c);
}
input.close();
} else
{
System.out.println("No Content Available");
} }
}
```

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

Web-сервисы

Web-сервисы - новое слово в технологии распределенных систем. Спецификация Open Net Environment (ONE) корпорации Sun Microsystems и инициатива .Net корпорации Microsoft обеспечивают инфраструктуры для написания и развертывания Web -сервисов. В настоящий момент имеется несколько определений Web-сервиса. Web -сервисом может быть любое приложение, имеющее доступ к Web, например, Web -страница с динамическим содержимым. В более узком смысле Web -сервис - это приложение, которое предоставляет открытый интерфейс, пригодный для использования другими приложениями в Web.

Спецификация ONE Sun требует, чтобы Web -сервисы были доступны через HTTP и другие Web -протоколы, чтобы дать возможность обмениваться информацией посредством XML-сообщений и чтобы их можно было найти через специальные сервисы - сервисы поиска. Для доступа к Web сервисам разработан специальный протокол - Simple Object Access Protocol (SOAP),который представляет средства взаимодействия на базе XML для многих Web -сервисов. Web -сервисы особенно привлекательны тем, что могут обеспечить высокую степень совместимости между различными системами. Огромный потенциал Web -сервисов определяется не технологией, примененной для их создания. HTTP, XML и другие протоколы, используемые Web -сервисами, не новы. Функциональная совместимость и масштабируемость Web -сервисов подразумевает, что разработчики могут быстро создавать большие приложения и более крупные Web -сервисы из меньших Web -сервисов. Спецификация Sun Open Net Environment описывает архитектуру для созданияинтеллектуальных Web-сервисов.Интеллектуальные Web -сервисы задействуют общее операционное окружение. Совместно используя контекст, интеллектуальныеWeb -сервисы могут выполнять стандартную аутентификацию для финансовых транзакций, предоставлять рекомендации и указания в зависимости от географического

местоположения компаний, участвующих в электронном бизнесе. Согласно определению W3C, " WSDL - формат XML для описания сетевых сервисов как набора конечных операций, работающих при помощи сообщений, содержащих документно-ориентированную или процедурно-ориентированную информацию". Документ WSDL полностью описывает интерфейс Web сервиса с внешним миром. Он предоставляет информацию об услугах, которые можно получить, воспользовавшись методами сервиса, и способах обращения к этим методам. Таким образом, в случае если сигнатура метода Web сервиса точно не известна (например, она изменилась со временем), у целевого Web -сервиса может быть запрошено WSDL -описание - файл, в котором эта информация будет содержаться.

Следующим слоем технологии является сервис Universal Description, Discovery and Integration (UDDI).Эта технология предполагает ведение реестра Web -сервисов. Подключившись к этому реестру, потребитель сможет найти Web -сервисы, которые наилучшим образом подходят для решения его задач. Технология UDDI дает возможность поиска и публикации нужного сервиса, причем эти операции могут быть выполнены как человеком, так и другим Web -сервисом или специальной программой-клиентом. UDDI, в свою очередь, также представляет собой Web -сервис.

Таким образом, Web -сервисы являются еще одной реализацией системного программного обеспечения промежуточного слоя. Отличительной чертой этой технологии является ее независимость от используемого программного и аппаратного обеспечения, а также использование широко применяемых открытых стандартов (таких как XML) и стандартных коммуникационных протоколов.

В настоящее время Web -сервисы являются очень активно продвигаемой технологией и позиционируются как средство решения целого ряда задач.

Следует отметить, что с их применением могут строиться и так называемые "стандартные" приложения, где в качестве Web -сервиса оформляется серверная часть.

Простой протокол доступа к объектам (SOAP)

Базовым протоколом, обеспечивающим взаимодействие в среде Web сервисов, является протокол SOAP.

Протокол SOAP разработали корпорации IBM, Lotus Development Corporation, Microsoft, Develop-Mentor и Userland Software. Этот протокол основан на HTTP-XML. Он позволяет приложениям взаимодействовать между собой через Internet, используя для этого XML -документы, называемые сообщениями SOAP. Протокол SOAP совместим с любой объектной моделью, поскольку он включает только те функции и методы, которые абсолютно необходимы для формирования коммуникационной инфраструктуры. Таким образом, SOAP является независимым от платформы и конкретных приложений, а для его реализации может применяться любой язык

программирования. SOAP поддерживает практически любой транспортный протокол. SOAP также поддерживает любые методы кодирования данных, которые позволяют приложениям, основанным на SOAP, посылать в сообщениях SOAP информацию практически любого типа (например, изображения, объекты, документы и т.д.).

Сообщение SOAP содержит конверт, который описывает содержимое, предполагаемого получателя сообщения и требования к обработке сообщения. Необязательный элемент **header** (заголовок) сообщения SOAP содержит инструкции по обработке для приложений, которые принимают сообщение. Заголовок также может содержать информацию о маршрутизации. С помощью заголовка **header** поверх SOAP могут надстраиваться более сложные протоколы. Записи в заголовке могут модульно расширять сообщение для таких задач, как аутентификация, управление транзакциями и проведение платежей. Тело SOAP -сообщения содержит специфичные для приложения данные, предназначенные для предполагаемого получателя сообщения.

Исходный код Web-сервиса

После того, как указанные настройки будут закончены, можно приступить непосредственно к примеру.

Файл с исходным кодом Web -сервиса располагается в директории src, называется Hello.java и имеет следующий вид:

```
1  // Hello.java
2  package helloservice.endpoint;
3
4  import javax.jws.WebMethod;
5  import javax.jws.WebService;
6
7     @WebService()
8     public class Hello {
9     private String message = new String("Hello, ");
10
11        @WebMethod()
12        public String sayHello(String name) {
13        return message + name + ".";
14        }
15        }
```

Первое, что обращает на себя внимание, - удивительная краткость написанного кода. Но не стоит обольщаться: дело в том, что технология разработки Web -сервисов, которую мы будем использовать, просто скрывает от разработчика большую часть работы по реализации Web -сервиса. Фактически, все, что должен сделать разработчик, - реализовать код самих вызываемых методов; абсолютно всю работу по реализации механизмов,

позволяющих вызывать эти методы удаленно, берет на себя используемая нами технология.

Рассматриваемый класс **Hello** удовлетворяет всем указанным ограничениям. Кроме того, он объявляет единственный метод, аннотированный как **WebMethod** (строка 11), который принимает параметр типа **String** и возвращает его же с присоединенной в начале константной строкой.

Собственно, на этом разработка Web -сервиса заканчивается. Следующее, что необходимо сделать, - откомпилировать его, пропустить через утилиту **wsgen** для генерации вспомогательных классов, создать **warfile**,содержащий в себе откомпилированное приложение и необходимые ресурсы, и затем разместить и зарегистрировать его на сервере приложений. В комплекте с примерами, поставляемыми в пакете The Java Web Services Tutorial,поставляются также скрипты для их компиляции. Эти скрипты предназначены для специального инструментального средства компиляции, которое называется ant (исполняющая часть **ant** устанавливается вместе с Sun Java System Application Server).

Разработчики примеров для пакета The Java Web Services Tutorial постарались на славу, и теперь для компиляции и установки приложения необходимо выполнить лишь несколько простых команд. Мы воспользуемся этим обстоятельством, а затем подробно рассмотрим, что стоит за каждой из этих простых команд и какие действия при этом выполняются.

Компиляция и инсталляция на сервере приложений

Итак, первое, что предстоит сделать, - откомпилировать приложение. Для компиляции в настройках сборки определена специальная цель (target) build.

Набрав в командной строке команду **asant build (asant** - вызов командного файла, запускающего **ant, build** - имя цели, которую он должен выполнить), получим следующий вывод:

```
Buildfile: build.xml

javaee-home-test:
 init:

prepare:
[echo] Creating the required directories....
[mkdir] Created dir: H:\Java\jwstutorial20_new\examples\
jaxws\helloservice\build

compile-service:
[echo] Compiling the server-side source code ... [javac]
Compiling 1 source file to H:\Java\
jwstutorial20_new\examples\jaxws\helloservice\build
```

```
[wsgen] command line: wsimport -classpath
H:\Java\AppServer\lib\activation.jar;
H:\Java\AppServer\lib\admin-cli.jar;
H:\Java\AppServer\lib\appserv-admin.jar;
H:\Java\AppServer\lib\appserv-cmp.jar;
H:\Java\AppServer\lib\appserv-deployment-client.jar;
H:\Java\AppServer\lib\appserv-ext.jar; H:\Java\AppServer\lib\appserv-jstl.jar;
H:\Java\AppServer\lib\appserv-jwsacc.jar;
H:\Java\AppServer\lib\appserv-launch.jar;
H:\Java\AppServer\lib\appserv-rt.jar;
H:\Java\AppServer\lib\appserv-tags.jar;
H:\Java\AppServer\lib\appserv-upgrade.jar;
H:\Java\AppServer\lib\appserv-ws.jar;
H:\Java\AppServer\lib\com-sun-commons-launcher.jar;
H:\Java\AppServer\lib\com-sun-commons-logging.jar;
H:\Java\AppServer\lib\dbschema.jar;
H:\Java\AppServer\lib\j2ee-svc.jar;
H:\Java\AppServer\lib\j2ee.jar;
H:\Java\AppServer\lib\javaee.jar;
H:\Java\AppServer\lib\jhall.jar;
H:\Java\AppServer\lib\jmxremote_optional.jar;
H:\Java\AppServer\lib\jsf-impl.jar;
H:\Java\AppServer\lib\mail.jar;
H:\Java\AppServer\lib\sun-appserv-ant.jar;
H:\Java\AppServer\lib\toplink-essentials-agent.jar;
H:\Java\AppServer\lib\toplink-essentials.jar;
H:\Java\AppServer\jdk\lib\tools.jar;
H:\Java\jwstutorial20_new\examples\jaxws\helloservice\build -d
H:\Java\jwstutorial20_new\examples\jaxws\helloservice\ build -keep -s
H:\Java\jwstutorial20_new\examples\jaxws\helloservice\ build
-verbose helloservice.endpoint.Hello [wsgen] Note:      ap round: 1
[wsgen] [ProcessedMethods Class: helloservice.endpoint.Hello]
[wsgen] [should process method: sayHello hasWebMethods: true ]
[wsgen] [endpointReferencesInterface: false]
[wsgen] [declaring class has WebSevice: true]
[wsgen] [returning: true]
[wsgen] [WrapperGen - method: sayHello(java.lang.String)]  [wsgen]
[method.getDeclaringType():
helloservice.endpoint.Hello]
[wsgen] [requestWrapper: helloservice.endpoint.jaxws.SayHello]
[wsgen] [ProcessedMethods Class: java.lang.Object]
```

[wsgen] helloservice\endpoint\jaxws\SayHello.java
[wsgen] helloservice\endpoint\jaxws\SayHelloResponse.java
[wsgen] Note: ap round: 2
[wsgen] [completing model for endpoint:
helloservice.endpoint.Hello]
[wsgen] [ProcessedMethods Class: helloservice.endpoint.Hello]
[wsgen] [should process method: sayHello hasWebMethods: true]
[wsgen] [endpointReferencesInterface: false]
[wsgen] [declaring class has WebSevice: true]
[wsgen] [returning: true]
[wsgen] [WebServiceReferenceCollector - method:
sayHello(java.lang.String)]
[wsgen] [ProcessedMethods Class: java.lang.Object]
build-service: build:
BUILD SUCCESSFUL
Total time: 9 seconds

Поскольку мы намеренно включили опцию вывода отладочной информации для **ant**,вывод получился довольно обширный.

Первое, что делается для компиляции программы, - создается специальная директория build,в которую будут помещены откомпилированные модули. Она создается втекущей директории. Затем вызывается компилятор javac, который компилирует наш класс **Hello.java**,а результат компиляции кладет в директорию build.Поскольку класс **Hello** определен в паке-
те **helloservice.endpoint**,в директории build будет создана соответствующая система каталогов и файл **Hello.class** будет помещен в каталог ./build/ helloservice/endpoint.

Следующим шагом вызывается утилита **wsgen**,которая формирует вспомогательные классы. По умолчанию исходные коды этих классов после компиляции уничтожаются, однако, выставив опцию **keep=true** (эта и другие опции могут быть установлены в файле **build.properties**),исходные коды можно сохранить. Помещаются они в пакет **jaxws** того же пакета, которому принадлежит и класс. Соответственно, для нашего примера исходные файлы (а затем и откомпилированные классы) будут располагаться в директории ./build/ helloservice/endpoint/jaxws. После того как утилита wsgen отработала, мы имеем откомпилированный пакет **helloservice.endpoint.jaxws**,содержащий необходимые вспомогательные классы. На этом шаге компиляция нашего Web -сервиса закончена. Следующим этапом необходимо подготовить модуль развертывания. В нашем случае это делается с помощью команды:

asant create-war

Вывод получаем следующий:

Buildfile: build.xml

prepare-assemble:
[echo] Creating the assemble directory....
[mkdir] Created dir: H:\Java\jwstutorial20_new\examples\
jaxws\helloservice\assemble
[mkdir] Created dir: H:\Java\jwstutorial20_new\examples\
jaxws\helloservice\assemble\war

create-war:
[echo] Creating the WAR ...
[war] Building war: H:\Java\jwstutorial20_new\examples\
jaxws\helloservice\assemble\war\hello-jaxws.war
BUILD SUCCESSFUL
Total time: 3 seconds

Создается отдельный каталог assemble,в нем создается каталог war, в котором формируется файл **hello-jaxws.war**.Этот файл представляет собой архив, в который помещены откомпилированные файлы нашего приложения и некоторые вспомогательные файлы. Теперь у нас полностью готов модуль развертывания, который мы можем инсталлировать в сервере приложений. Инсталляция может быть выполнена командой: asant deploy

Результат выполнения команды следующий:
Buildfile: build.xml deploy: admin_command_common:

[echo] Doing admin task deploy assemble/war/hello-jaxws.war
[sun-appserv-admin] Executing: deploy --port 4848 --host
localhost --passwordfile "H:\Java\jwstutorial20_new\examples\
common\admin-password.txt" --user admin assemble/war/ hello-
jaxws.war
[sun-appserv-admin] Command deploy executed successfully.

BUILD SUCCESSFUL
Total time: 43 seconds

В процессе инсталляции, кроме прочего, был сгенерирован WSDL -файл, описывающий установленный Web -сервис. Как уже говорилось, этот файл содержит полное описание Web -сервиса, включая названия его методов, а также количество и типы передаваемых и возвращаемых параметров. Этот файл является важной составляющей частью технологии, поскольку он позволяет строить приложения, осуществляющие динамические вызовы методов Web -сервисов. Кроме того, этот файл может быть использован для автоматической генерации вспомогательных классов (классов- proxy) для обращения к Web -сервису.

Тестирование Web-сервиса

Итак, наш Web -сервис успешно инсталлирован. Осталось только убедиться в том, что он действительно работает. Чуть позже мы напишем специальное приложение-клиент, которое будет обращаться к нашему Web сервису, а пока воспользуемся средствами, предоставляемыми нам Sun Java System Application Server.Дело в том, что этот сервер приложений способен самостоятельно динамически выстроить среду для вызова методов инсталлированных в нем Web -сервисов. Всей необходимой информацией, а именно: имена публикуемых методов, количество и тип принимаемых и возвращаемых методами параметров - он обладает.

Для того чтобы воспользоваться указанной возможностью, нужно выбрать нужный нам сервис в списке сервисов (в правой части окна браузера) и нажать кнопку "Test" .

Откроется новое окно браузера, в котором отобразится динамически построенная сервером страница. На этой странице перечислены все опубликованные методы Web -сервиса (в нашем случае - один метод **sayHello)** и реализован интерфейс для их вызова. Если ввести в соответствующее поле строку и нажать кнопку - вызовется метод Web -сервиса и введенное значение будет передано ему в качестве параметра. Кроме всего прочего, на результирующей странице отобразятся SOAP -сообщения, соответственно, отправленные Web -сервису и пришедшие от него в качестве ответа.

Результирующая страница будет иметь следующий вид.

На странице видны значения и типы переданных параметров, ответ, который возвратил метод Web -сервиса, - как и ожидалось, ответ представляет собой строку "Hello, Web-service test", - а также отправленный и полученный пакеты.

Таким образом, разработанный нами Web -сервис успешно инсталлирован в сервере приложений и может обрабатывать запросы клиентов, в чем мы убедились, использовав тестовое окружение, предоставляемое сервером приложений.

Раздел 2 ПОТОКИ ВВОДА-ВЫВОДА. ВНУТРЕННИЕ КЛАССЫ

Тема 2.1 Потоки ввода-вывода

Обобщенное понятие источника ввода относится к различным способам получения информации: к чтению дискового файла, символов с клавиатуры, либо получению данных из сети. Аналогично, под обобщенным понятием вывода также могут пониматься дисковые файлы, сетевое соединение и т.п. Эти абстракции дают удобную возможность для работы с вводом-выводом (I/O), не требуя при этом, чтобы каждая часть вашего кода понимала разницу

между, скажем, клавиатурой и сетью. В Java эта абстракция называется потоком (stream) и реализована в нескольких классах пакета java.io. Ввод инкапсулирован в классе InputStream, вывод — в OutputStream. В Java есть несколько специализаций этих абстрактных классов, учитывающих различия при работе с дисковыми файлами, сетевыми соединениями и даже с буферами в памяти.

File

File — единственный объект в java.io, который работает непосредственно с дисковыми файлами. Хотя на использование файлов в апплетах наложены жесткие ограничения, файлы по прежнему остаются основными ресурсами для постоянного хранения и совместного использования информации. Каталог в Java трактуется как обычный файл, но с дополнительным свойством — списком имен файлов, который можно просмотреть с помощью метода list.

Для определения стандартных свойств объекта в классе File есть много разных методов. Однако, класс File несимметричен. Есть много методов, позволяющих узнать свойства объекта, но соответствующие функции для изменения этих свойств отсутствуют. В очередном примере используются различные методы, позволяющие получить характеристики файла: *import java.io.File; class FileTest { static void p(String s) {*

System.out.println(s);
}
public static void main(String args[]) { File f1 = new File("/java/COPYRIGHT"); p("File Name:" + f1 .getName()); p("Path:" + f1.getPath()); p("Abs Path:" + f1.getAbsolutePath()); p("Parent:" + f1.getParent()); p(f1.exists() ? "exists" : "does not exist"); p(f1.canWrite() ? "is writeable" : "is not writeable"); p(f1.canRead() ? "is readable" : "is not readable"); p("is " + (f1.isDirectory() ? " " : "not") + " a directory"); p(f1.isFile() ? "is normal file" : "might be a named pipe"); p(f1.isAbsolute() ? "is absolute" : "is not absolute"); p("File last modified:" + f1. lastModified()); p("File size:" + f1.length() + " Bytes");
} }

При запуске этой программы вы получите что-то наподобие вроде:

File Name:COPYRIGHT (имя файла)
Path:/java/COPYRIGHT (путь)
Abs Path:/Java/COPYRIGHT (путь от корневого каталога)
Parent:/java (родительский каталог) exists (файл существует) is writeable (разрешена запись) is readable (разрешено чтение) is not a directory (не каталог) is normal file (обычный файл)

is absolute

File last modified:812465204000 (последняя модификация файла)

File size:695 Bytes (размер файла)

Существует также несколько сервисных методов, использование которых ограничено обычными файлами (их нельзя применять к каталогам). Метод renameTo(File dest) переименовывает файл (нельзя переместить файл в другой каталог). Метод delete уничтожает дисковый файл. Этот метод может удалять только обычные файлы, каталог, даже пустой, с его помощью удалить не удастся.

Каталоги

Каталоги — это объекты класса File, в которых содержится список других файлов и каталогов. Если File ссылается на каталог, его метод isDirectory возвращает значение true. В этом случае вы можете вызвать метод list и извлечь содержащиеся в объекте имена файлов и каталогов. В очередном примере показано, как с помощью метода list можно просмотреть содержимое каталога.

```
import java.io.File; class
DirList {
public static void main(String args[]) {
String dirname = "/java"; // имя каталога  File
f1 = new File(dirname);
if (f1.isDirectory()) { // является ли f1 каталогом
System.out.println("Directory of ' + dirname);
String s[] = f1.list();
for ( int i=0; i < s.length; i++) { File f
= new File(dirname + "/" + s[i]);
if (f.isDirectory()) { // является ли f каталогом System.out.println(s[i] + " is a
directory"):
} else {
System.out.println(s[i] + " is a file"); }
} } else { System.out.println(dirname
+ " is not a directory");
} } }
```

В процессе работы эта программа вывела содержимое каталога /java моего персонального компьютера в следующем виде:

```
C:\> java DirList
Directory of /java bin
is a directory
COPYRIGHT is a file
README is a file FilenameFilter
```

Зачастую у вас будет возникать потребность ограничить количество имен файлов, возвращаемых методом list, чтобы получить от него только имена,

соответствующие определенному шаблону. Для этого в пакет java.io включен интерфейс FilenameFilter. Объекту, чтобы реализовать этот интерфейс, требуется определить только один метод — accept(), который будет вызываться один раз с каждым новым именем файла. Метод accept должен возвращать true для тех имен, которые надо включать в список, и false для имен, которые следует исключить.

У класса File есть еще два сервисных метода, ориентированных на работу с каталогами. Метод mkdir создает подкаталог. Для создания каталога, путь к которому еще не создан, надо использовать метод mkdirs — он создаст не только указанный каталог, но и все отсутствующие родительские каталоги.

InputStream

InputStream — абстрактный класс, задающий используемую в Java модель входных потоков. Все методы этого класса при возникновении ошибки возбуждают исключение IOException. Ниже приведен краткий обзор методов класса InputStream.

- read() возвращает представление очередного доступного символа во входном потоке в виде целого.
- read(byte b[]) пытается прочесть максимум b.length байтов из входного потока в массив b. Возвращает количество байтов, в действительности прочитанных из потока.
- read(byte b[], int off, int len) пытается прочесть максимум len байтов, расположив их в массиве b, начиная с элемента off. Возвращает количество реально прочитанных байтов.
- skip(long n) пытается пропустить во входном потоке n байтов. Возвращает количество пропущенных байтов.
- available() возвращает количество байтов, доступных для чтения в настоящий момент.
- close() закрывает источник ввода. Последующие попытки чтения из этого потока приводят к возбуждению IOException.
- mark(int readlimit) ставит метку в текущей позиции входного потока, которую можно будет использовать до тех пор, пока из потока не будет прочитано readlimit байтов.
- reset() возвращает указатель потока на установленную ранее метку.
- markSupported() возвращает true, если данный поток поддерживает операции mark/reset. OutputStream

Как и InputStream, OutputStream — абстрактный класс. Он задает модель выходных потоков Java. Все методы этого класса имеют тип void и возбуждают исключение IOException в случае ошибки. Ниже приведен список методов этого класса:

- write(int b) записывает один байт в выходной поток. Обратите внимание — аргумент этого метода имеет тип int, что позволяет вызывать write,

передавая ему выражение, при этом не нужно выполнять приведение его типа к byte.

- write(byte b[]) записывает в выходной поток весь указанный массив байтов.
- write(byte b[], int off, int len) записывает в поток часть массива — len байтов, начиная с элемента b[off].
- flush() очищает любые выходные буферы, завершая операцию вывода.
- close() закрывает выходной поток. Последующие попытки записи в этот поток будут возбуждать IOException.

Файловые потоки

FileInputStream

Класс FileInputStream используется для ввода данных из файлов. В приведенном ниже примере создается два объекта этого класса, использующие один и тот же дисковый файл.

```
InputStream f0 = new FileInputStream("/autoexec.bat");
File f = new File("/autoexec.bat"): InputStream f1 =
new FileInputStream(f);
```

Когда создается объект класса FileInputStream, он одновременно с этим открывается для чтения. FileInputStream замещает шесть методов абстрактного класса InputStream. Попытки применить к объекту этого класса методы mark и reset приводят к возбуждению исключения IOException. В приведенном ниже примере показано, как можно читать одиночные байты, массив байтов и поддиапазон массива байтов. В этом примере также показано, как методом available можно узнать, сколько еще осталось непрочитанных байтов, и как с помощью метода skip можно пропустить те байты, которые вы не хотите читать.

```
import java.io.*; import java.util.*; class FileInputStreamS
{ public static void main(String args[]) throws  Exception {
int size;
InputStream f1 = new FileInputStream("/wwwroot/default.htm"); size
= f1.available();
System.out.println("Total Available Bytes: " + size); System.out.println("First
1/4 of the file: read()");
for (int i=0; i < size/4; i++) {
System.out.print((char) f1.read());
}
System.out.println("Total Still Available: " + f1.available());
System.out.println("Reading the next 1/8: read(b[])"); byte
b[] = new byte[size/8]; if (f1.read(b) != b.length) {
System.err.println("Something bad happened");
}
```

```
String tmpstr = new String(b, 0, 0, b.length);
System.out.println(tmpstr);
System.out.println("Still Available: " + f1.available());
System.out.println("Skipping another 1/4: skip()"); f1.skip(size/4);
System.out.println( "Still Available: " + f1.available());
System.out.println("Reading 1/16 into the end of array"); if
(f1.read(b, b.length-size/16, size/16) != size/16) {
System.err.println("Something bad happened");
}
System.out.println("Still Available: " + f1.available()); f1.close();
}
}
```

FileOutputStream

У класса FileOutputStream — два таких же конструктора, что и у FileInputStream. Однако, создавать объекты этого класса можно независимо от того, существует файл или нет. При создании нового объекта класс FileOutputStream перед тем, как открыть файл для вывода, сначала создает его.

В очередном нашем примере символы, введенные с клавиатуры, считываются из потока System.in - по одному символу за вызов, до тех пор, пока не заполнится 12-байтовый буфер. После этого создаются три файла. В первый из них, file1.txt, записываются символы из буфера, но не все, а через один — нулевой, второй и так далее. Во второй, file2.txt, записывается весь ввод, попавший в буфер. И наконец в третий файл записывается половина буфера, расположенная в середине, а первая и последняя четверти буфера не выводятся.

```
import java.io.*; class
FileOutputStreamS {
public static byte getlnput()[] throws Exception {
byte buffer[] = new byte[12]; for
(int i=0; i<12; i++) {
buffer[i] = (byte) System.in.read();
}
return buffer;
}
public static void main(String args[]) throws Exception {  byte
buf[] = getlnput();
OutputStream f0 = new FileOutputStream("file1.txt");
OutputStream f1 = new FileOutputStream("file2.txt"); OutputStream
f2 = new FileOutputStream("file3.txt");
for (int i=0; i < 12; i += 2) {  f0.write(buf[i]);
```

```
} f0.close();
f1.write(buf); f1.close();
f2.write(buf, 12/4, 12/2);
f2.close();
} }
```

ByteArraylnputStream

ByteArrayInputStream - это реализация входного потока, в котором в качестве источника используется массив типа byte. У этого класса два конструктора, каждый из которых в качестве первого параметра требует байтовый массив. В приведенном ниже примере создаются два объекта этого типа. Эти объекты инициализируются символами латинского алфавита.

```
String tmp = "abcdefghijklmnopqrstuvwxyz";
byte b[] = new byte [tmp.length()]; tmp.
getBytes(0, tmp.length(), b, 0);
ByteArrayInputStream input1 = new ByteArrayInputStream(b);
ByteArrayInputStream input2 = new ByteArreyInputStream(b,0,3);
```

ByteArrayOutputStream

У класса ByteArrayOutputStream — два конструктора. Первая форма конструктора создает буфер размером 32 байта. При использовании второй формы создается буфер с размером, заданным параметром конструктора (в приведенном ниже примере — 1024 байта):

```
OutputStream out0 = new ByteArrayOutputStream(); OutputStream
out1 = new ByteArrayOutputStream(1024);
```

В очередном примере объект ByteArrayOutputStream заполняется символами, введенными с клавиатуры, после чего с ним выполняются различные манипуляции.

```
import java.io.*; import
java.util.*; class
ByteArrayOutputStreamS {
public static void main(String args[]) throws Exception {
int i;
ByteArrayOutputStream f0 = new ByteArrayOutputStream(12);
System.out.println("Enter 10 characters and a return"); while
(f0.size() != 10) { f0.write( System.in.read());
}
System.out.println("Buffer as a string");
System.out.println(f0.toString());
System.out.println ("Into array");
byte b[] = f0.toByteArray(); for
(i=0; i < b.length; i++) {
System.out.print((char) b[i]);
}
```

```
System.out.println();
System.out. println("To an OutputStream()"); OutputStream
f2 = new File0utput8tream("test.txt"); f0.writeTo(f2);
System.out.println("Doing a reset"); f0.
reset();
System.out.println("Enter 10 characters and a return"); while
(f0.size() != 10) {
f0.write (System.in.read());
}
System.out.println("Done.");
} }
```

Заглянув в созданный в этом примере файл test.txt, мы увидим там именно то, что ожидали:

```
C:\> type test.txt
0123456789
```

StringBufferlnputStream

StringBufferInputStream идентичен классу ByteArrayInputStream с тем исключением, что внутренним буфером объекта этого класса является экземпляр String, а не байтовый массив. Кроме того, в Java нет соответствующего ему класса StringBufferedOutputStream. У этого класса есть единственный конструктор:

StringBufferInputStream(String s)

Фильтруемые потоки

При работе системы вывода в среде с параллельными процессами при отсутствии синхронизации могут возникать неожиданные результаты. Причиной этого являются попытки различных подпроцессов одновременно обратиться к одному и тому же потоку. Все конструкторы и методы, имеющиеся в этом классе, идентичны тем, которые есть в классах InputStream и OutputStream, единственное отличие классов фильтруемых потоков в том, что их методы синхронизованы.

Буферизованные потоки

Буферизованные потоки являются расширением классов фильтруемых потоков, в них к потокам ввода-вывода присоединяется буфер в памяти. Этот буфер выполняет две основные функции:

- Он дает возможность исполняющей среде java проделывать за один раз операции ввода-вывода с более чем одним байтом, тем самым повышая производительность среды.
- Поскольку у потока есть буфер, становятся возможными такие операции, как пропуск данных в потоке, установка меток и очистка буфера.

BufferedInputStream

Буферизация ввода-вывода — общепринятый способ оптимизации таких операций. Класс BufferedlnputStream в Java дает возможность “окружить” любой объект InputStream буферизованным потоком, и, тем самым, получить выигрыш в производительности. У этого класса два конструктора. Первый из них

BufferedInputStream(InputStream in) создает буферизованный поток, используя для него буфер длиной 32 байта. Во втором

BufferedInputStream(InputStream in, int size)

размер буфера для создаваемого потока задается вторым параметром конструктора. В общем случае оптимальный размер буфера зависит от операционной системы, количества доступной оперативной памяти и конфигурации компьютера.

BufferedOutputStream

Вывод в объект BufferedOutputStream идентичен выводу в любой OutputStream с той разницей, что новый подкласс содержит дополнительный метод flush, применяемый для принудительной очистки буфера и физического вывода на внешнее устройство хранящейся в нем информации. Первая форма конструктора этого класса:

BufferedOutputStream(OutputStream out) создает поток с буфером размером 32 байта. Вторая форма:

BufferedOutputStream(OutputStream out, int size) позволяет задавать требуемый размер буфера.

PushbacklnputStream

Одно из необычных применений буферизации — реализация операции pushback (вернуть назад). Pushback применяется к InputStream для того, чтобы после прочтения символа вернуть его обратно во входной поток. Однако возможности класса PushbacklnputStream весьма ограничены - любая попытка вернуть в поток более одного символа приведет к немедленному возбуждению исключения IOException. У этого класса — единственный конструктор

PushbackInputStream(InputStream in)

Помимо уже хорошо нам знакомых методов класса InputStream, PushbacklnputStream содержит метод unread(int ch), который возвращает заданный аргументом символ ch во входной поток.

SequencelnputStream

Класс SequencelnputStream поддерживает новую возможность слияния нескольких входных потоков в один. В конструкторе класса SequenceInputStream в качестве параметра используется либо два объекта InputStream, либо перечисление, содержащее коллекцию объектов InputStream:

SequenceInputStream(Enumeration e) SequenceInputStream(InputStream s0, InputStream s1)

В процессе работы класс выполняет поступающие запросы, считывая информацию из первого входного потока до тех пор, пока он не закончится, после чего переходит ко второму и т.д.

PrintStream

Класс PrintStream предоставляет все те утилиты форматирования, которые мы использовали в примерах для вывода через файловые дескрипторы пакета System с самого начала книги. Вы уже привыкли писать “System.out.println”, не сильно задумываясь при этом о тех классах, которые занимаются форматированием выводимой информации. У класса PrintStream два конструктора:

PrintStream(OutputStream out) и PrintStream(OutputStream out, boolean autoflush). Параметр autoflush второго из них указывает, должна ли исполняющая среда Java автоматически выполнять операцию очистки буфера над выходным потоком.

В Java-объектах PrintStream есть методы print и println, “умеющие” работать с любыми типами данных, включая Object. Если в качестве аргумента этих методов используется не один из примитивных типов, то они вызывают метод toString класса Object, после чего выводят полученный результат.

ВНИМАНИЕ

В настоящее время в Java отсутствуют средства для форматирования выводимых данных простых типов, например, типов int и float. В C++ предусмотрены функции для форматирования чисел с плавающей точкой, позволяющие, например, задать вид вывода, при котором в напечатанном числе будет четыре цифры до десятичной точки и три - после. По течению грести легче

Потоки в Java предоставляют программисту ясную абстракцию для выполнения сложных и зачастую громоздких операций ввода-вывода данных. Javaпрограммы, опирающиеся на абстракции высокого уровня - классы InputStream и OutputStream, будут и в будущем функционировать правильно - даже тогда, когда будут изобретены новые улучшенные реализации классов ввода-вывода. Как вы увидите в следующей лекции, такая модель прекрасно работает и при переходе от набора потоков, ориентированных на файловую систему, к работе с сетевыми потоками и сокетами.

Тема 2.2 Внутренние классы

Классы, определенные внутри других классов, называются внутренними. Обычно внутренние классы не отличаются от любых других, за исключением того, что они определены в теле другого класса. Например:

```
public class Outer {
public class Inner {
      private int i;
```

```
        public void myMethod(){ … }
    }
}
```

Область существования

Внутренний класс, определенный вне методов класса, принадлежит классу и может применяться внутри класса (как поля класса). Объекты внутреннего класса могут быть созданы внутри любого метода класса. Внутренние классы могут быть определены внутри метода класса. Тогда они имеют область применения только внутри того метода, где они определены (как автоматические переменные). В этом методе можно создавать объекты данного класса, но не в других методах внешнего класса.

Внутренние классы, определенные внутри метода, имеют ряд ограничений:

- У них не может быть модификатора доступа
- Они не могут быть объявлены как static
- Внутренние классы могут использовать только те переменные и параметры метода, которые объявлены final

Внутренние классы полностью «видят» все поля и методы их внешних классов

```
public class Outer
{
   private int a = 5;
public class Inner {
       private int i=1;
       public void myMethod() {
          System.out.println( "a=" + a + ", i=" + i );
       }
   }

   public static void main( String[] args ) {
       Outer.Inner innerClass = new Outer().new Inner();
innerClass.myMethod();
   }
}
```

Внутренний класс Inner имеет доступ к собственной переменной i так же, как и к полю класса Outer переменной a, хотя она и имеет доступ private.

Модификаторы доступа

Из-за того, что внутренние классы, определенные вне методов, похожи на поля, можно применять к ним модификатор доступа. Модификатор доступа определяет, можно ли использовать объекты внутреннего класса вне внешнего класса.Если Вы объявляете внутренний класс public, то можно его

использовать, указав имя внешнего класса, а затем, через точку, имя внутреннего класса:

```
Outer outer = new Outer();  Outer.Inner
inner = o.new Inner()  Или:
Outer.Inner inner = new Outer().new Inner();
```

Статические внутренние классы

Внутренние классы могут иметь атрибут static. Так как они статические, то они не связаны с объектами внешнего класса. Это значит, что можно создвать объекты внутренних классов, не создавая объекта внешнего класса.

Внутренние статические классы имеют некоторые ограничения по доступа к членам внешнего класса:

Методы статического внутреннего класса не имеют доступа к полям и методам внешнего класса, если они не статические.

Методы статического внутреннего класса имеют доступ только к статическим полям и методам внешнего класса Листинг демонстрирует эти ограничения:

```
public class OuterTest
{
   public static int outerInt = 5;

   public static class StaticInner {
public static int doubleVal( int n ) {
        System.out.println( "outerInt=" + outerInt );
return 2*n;
      }
   }

   public void testInner() {
      int a = 5;
      System.out.println( "a=" + a + ", doubleVal=" +
                  StaticInner.doubleVal( a ) );
}

public static void main( String[] args ) {
int n = 7;
   System.out.println( "n=" + n + ", doubleVal=" +
                OuterTest.StaticInner.doubleVal( n ) );
   OuterTest out = new OuterTest();
   out.testInner();
}
```

}

Внутренний класс StaticInner использует статическое поле outerInt класса OuterTest в методе doubleVal. Если бы outerInt не былы бы static, класс StaticInner не мог бы использовать ее. Метод main показывает, как применять внутренний статический класс, не создавая объект внешнего класса:

OuterTest.StaticInner.doubleVal(n) ;

Анонимные внутренние классы

Они обладают следующими особенностями:

- Создаются без имени
- Определяются внутри метода
- Не имеют конструктора
- Объявляются и определяются в одном операторе
- Чаще всего используются для обработки событий

О ссылках на объекты

При создании переменной типа класса создается ссылка на объект этого класса. Ссылка нужна, чтобы обратиться к объекту класса, но она не жестко связана с конкретным объектом. Если Вы создаете два объекта, а потом выполняете присваивание, то обе ссылки указывают на один и тот же объект, а второй объект оказывается неиспользуемым и будет уничтожен при сборке мусора:

```
public class Number {
  private int number;

  public Number( int number ) {
    this.number = number;
  }

  public int getNumber() {
    return this.number;
  }

  public void setNumber( int number ) {
this.number = number;
  }

  public static void main( String[] args ) {
    Number one = new Number( 1 );
```

```
        Number two = new Number( 2 );
        System.out.println( "Beginning: " );
        System.out.println( "One = " + one.getNumber() );
        System.out.println( "Two = " + two.getNumber() );

        // Assign two to one
    two = one;
        System.out.println( "\nAfter assigning two to one: " );
        System.out.println( "One = " + one.getNumber() );
        System.out.println( "Two = " + two.getNumber() );

        // Change the value of two
    two.setNumber( 3 );
        System.out.println( "\nAfter modifying two: " );
        System.out.println( "One = " + one.getNumber() );
        System.out.println( "Two = " + two.getNumber() );
      }
    }
```

Будут выведены следующие данные:

```
Beginning:
One = 1
Two = 2

After assigning two to one:
One = 1
Two = 1

After modifying two:
One = 3
Two = 3
```

Отсюда ясно, что присваивание переменной two типа Number значения переменной one в действительности приводит к тому, что two указывает на ту же область памяти, что и one. Если модифицировать что-нибудь, используя two, то будет модифицировано и то, на что указывает one.

Раздел 3 РАЗРАБОТКА МОДЕЛЕЙ ПРОГРАММНЫХ СИСТЕМ

Тема 3.1 Разработка моделей программных систем

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

В начале 2000 года фирма Rational Software Corporation анонсировала появление собственного средства моделирования данных – Data Modeler, и в настоящее время оно доступно специалистам, например, использующим в своей работе Rational Rose 2000.

Целью данной лабораторной работы является знакомство с основными возможностями этого нового средства.

Авторы Data Modeler, прежде всего, ориентировались на создание инструмента проектирования физической модели данных. При этом не произошло отказа от UML как от средства моделирования данных, а некоторым образом были смещены акценты: теперь UML предполагается использовать для построения логической модели. По сути, логическая модель - это та же объектная модель, состоящая из объектов - сущностей. Переход от логической модели к физической и, наоборот, в части моделирования данных обеспечивается Rational Rose автоматически. Для этого введено соответствие элементов моделей (табл. 13.1).

Таблица 3.1. Соответствие элементов логической и физической модели

Логическая модель	Физическая модель
Class (Класс)	Table (Таблица)
Operation (Операция)	Constraint (Ограничение)
Attribute (Атрибут)	Column (Колонка)
Package (Пакет)	Scheme (Схема)

Component (Компонент)	Database (База данных)
Association (Ассоциация)	Relationship (Связь)
Нет	Trigger (Тригер)
Нет	Index (Индекс)

Rose Data Modeler

После установки Rational Rose в специальной редакции (Rational Rose Professional Data Modeler Edition) в разделе главного меню Tools появляется новый раздел Data Modeler (рис. 3.1).

В разделе Data Modeler имеются два пункта: “Add Schema” и “Reverse Engeneer…”. Пункт “Add Schema” используется для создания новых схем БД, а пункт “Reverse Engeneer” - для построения модели на основе существующей схемы БД.

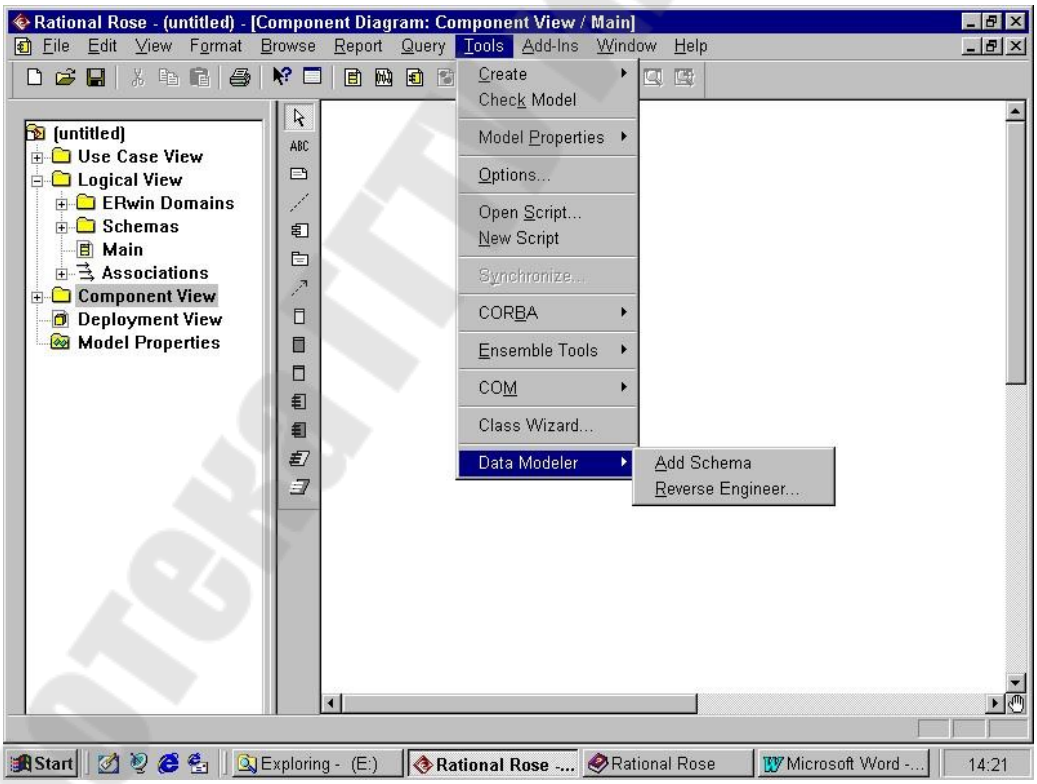


Рисунок 3.1- Отображение компоненты Data Modeler в меню Rational Rose Создание диаграммы модели данных

После создания схемы (рис. 3.1) в ней можно сформировать диаграмму модели данных. Эта диаграмма позволит добавить, изменить или просмотреть таблицы и другие элементы модели данных, поскольку играет ту же роль, что и диаграмма классов в объектной модели. Хотя можно добавлять элементы

моделирования данных непосредственно в браузере, диаграмма модели данных позволяет показать в графическом виде, как сами элементы, так и отношения между ними. Для любой схемы можно создать любое необходимое число Создание диаграммы модели данных осуществляется следующим образом:

1. В браузере щелкните правой кнопкой мыши на схеме.
2. Выберите Data Modeler > New > Data Model Diagram.
3. Введите имя новой диаграммы.
4. Дважды щелкните на диаграмме для ее открытия.

Как и другие диаграммы в среде Rose, диаграмма модели данных имеет специализированную панель инструментов для добавления таблиц, отношений и других элементов моделирования. Кнопки этой панели перечислены в таблице 3.2.

Таблица 3.2 – Значки панели инструментов для диаграммы модели данных

Значки	**Назначение**
	Курсор принимает форму стрелки для выделения элемента Добавляет в диаграмму текстовое поле Добавляет к элементу диаграммы примечание Соединяет примечание с элементом диаграммы Добавляет в диаграмму таблицу Рисует неидентифицируемое отношение между двумя таблицами Рисует идентифицируемое отношение между двумя таблицами Добавляет в диаграмму представление Рисует зависимость между двумя таблицами

Диаграмма Data Model предоставляет следующие возможности:

- создание и редактирование таблиц и их элементов (колонок, ограничений, индексов, триггеров и т. п.);
- создание и редактирование идентифицирующих связей между таблицами;
- создание и редактирование неидентифицирующих связей.

Основные возможности по работе с таблицей доступны, если войти в контекстном меню в пункт “Open Specification”. Появляющееся на экране окно включает следующую информацию (рис. 3.2).

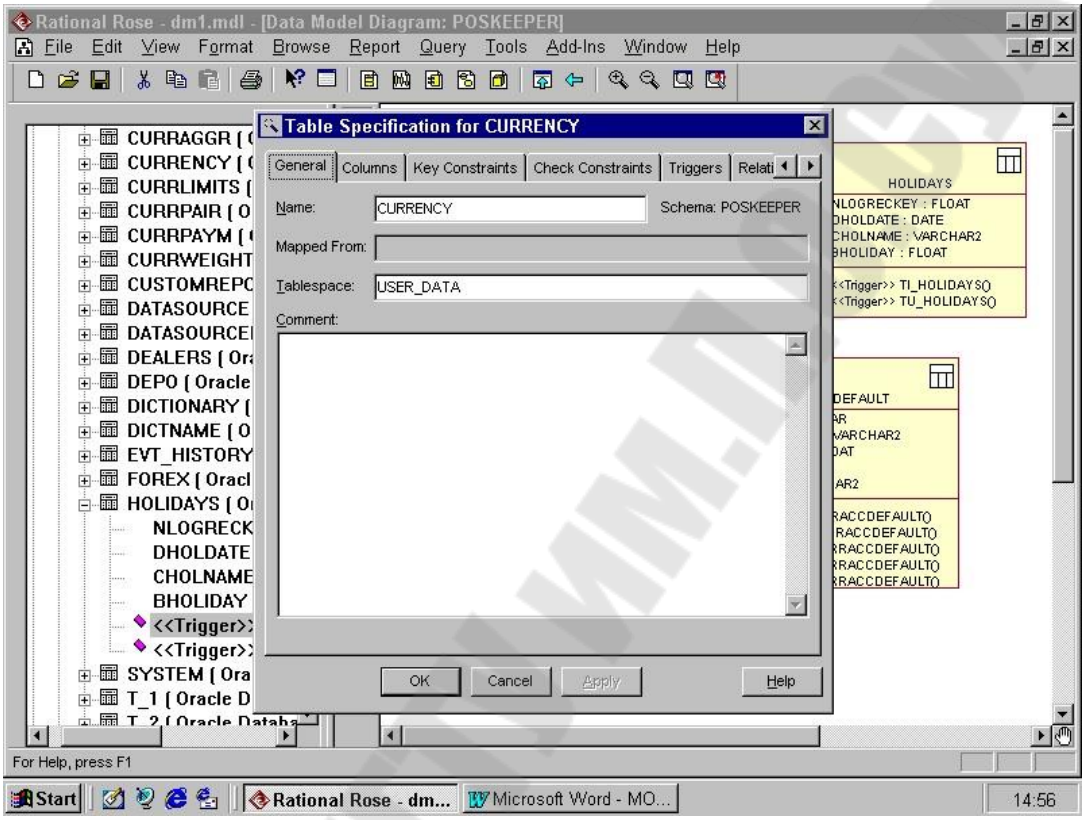


Рисунок 3.2 - Окно спецификации таблицы

При редактировании спецификации таблицы обеспечиваются следующие возможности (табл. 13.3).

Таблица 3.3 - Спецификация таблицы БД

Закладка	Описание
General	Вводится общая информация о таблице.
Columns	Задается описание колонок. Здесь можно добавить или отредактировать свойства колонок, задать тип, длину, обязательность (NULL, NOT NULL), а также пометить, что колонка входит в состав первичного ключа. Типы колонок соответствуют типам конкретной выбранной СУБД.
Key Constraints	Задаются ограничения на колонки таблицы. Здесь можно задать ограничение на уникальность первичного ключа, ограничение на уникальность альтернативных ключей, а также про-
	сто определить индекс.

Check Constraints	Задаются выражения – инварианты, которые должны выполнятся для всех строк таблицы.
Triggers	Содержит список триггеров, который можно отредактировать, в том числе добавив новый триггер.
Relationships	При наличии связей между таблицами, закладка содержит полный список связей.

Добавление отношений

Отношения в модели данных подобны отношениям в объектной модели. В объектной модели отношение связывает два класса, а в модели данных — две таблицы. В Rose поддерживаются два основных типа отношений: идентифицируемые отношения (identifying relationship) и неидентифицируемые отношения (non-identifying relationship).

В обоих случаях для поддержки отношений в дочернюю таблицу добавляется внешний ключ. При идентифицируемом отношении внешний ключ становится частью первичного ключа в дочерней таблице. В этом случае дочерняя таблица не может содержать запись, не связанную с записью в родительской таблице. Идентифицируемые отношения моделируются составными агрегациями.

Неидентифицируемые отношения тоже создают внешний ключ в дочерней таблице, но он не становится частью первичного ключа в дочерней таблице. При неидентифицируемом отношении мощность (множественность) определяет то, будет ли запись в дочерней таблице существовать без связи с записью в родительской таблице. Если мощность равна 1, должна присутствовать родительская запись. Если мощность равна 0..1, присутствие родительской записи необязательно. Неидентифицируемые отношения моделируются ассоциациями.

Для редактирования свойств связи требуется войти в пункт контекстного меню “Open specification”.

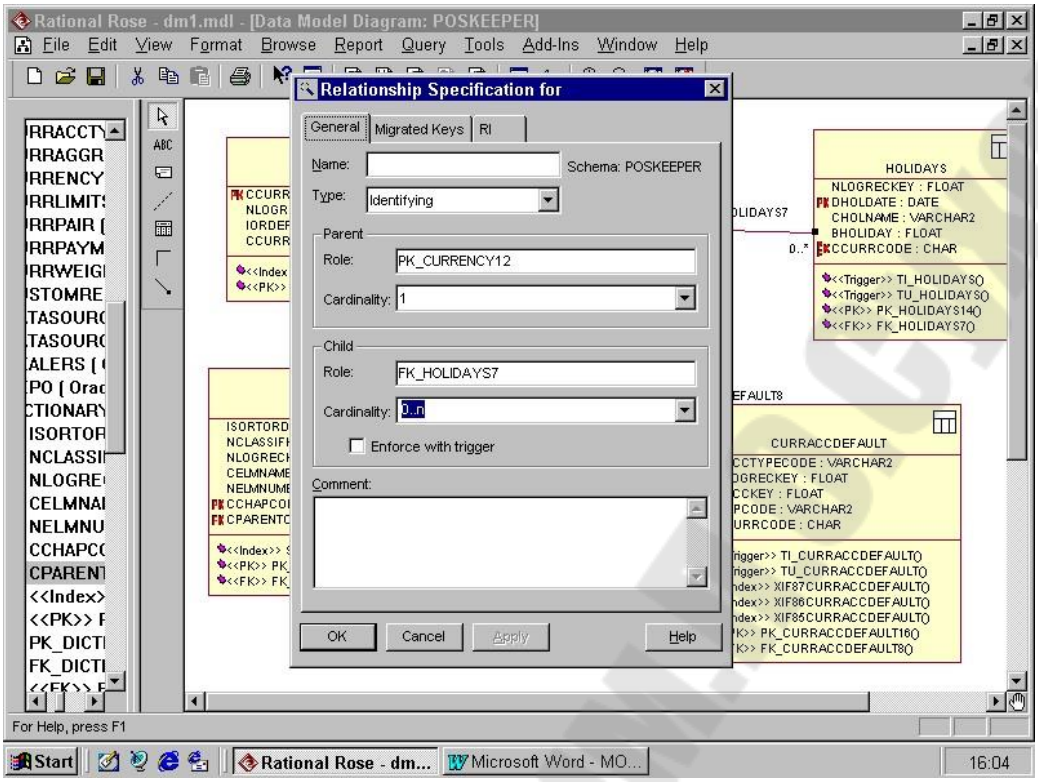


Рисунок 3.3 - Окно спецификации связи

При редактировании спецификации связи обеспечиваются следующие возможности (табл. 3.4).

Таблица 3.4 - Спецификация связи

Закладка	Описание
General	Основные свойства связи. Здесь задаются: имя связи; тип связи; наименования ролей (Parent, Child); кардинальность для каждой роли;
Migrated Key	Содержит список внешних ключей, образующихся в результате создания связи.

RI	Задание условий ссылочной целостности. Ссылочная целостность обеспечивается двумя способами: на основе триггеров; на основе декларативной ссылочной целостности (с использованием ограничений внешних ключей). Оба способа реализуют наиболее популярные алгоритмы, задаваемые для каждой роли (только для операций update и delete, для insert мы не нашли): Restrict; Cascade; Set Null; Set Default.

Пример концептуальной модели предметной области

На рисунке 3.4 приведен фрагмент модели данных заданной предметной области. Данная модель разработана в среде Rational Rose 2003 с использованием утилиты Rose Data Modeler.

Описание предметной области.

Организация предоставляет услуги по трудоустройству. Организацией ведется банк данных о существующих вакансиях. По каждой вакансии поддерживается следующая информация:

- предприятие, предоставляющее соответствующую вакансию;
- название вакансии (должность);
- требования к соискателю: пол, возраст, образование, знание определенных видов деятельности (выбор из перечня - знание электронного документооборота, определенных прикладных программ и т.п.), коммуникабельность;
- обязанности (выбор из перечня – заключение договоров, распространение агитационного материала, работа с клиентами и т.п.);
- предполагаемая оплата, единицы измерения оплаты - рубли;
- оформление трудовой книжки (да, нет);
- наличие социального пакета (да, нет);
- срок начала открытия вакансии;
- срок закрытия вакансии (вакансия занята).

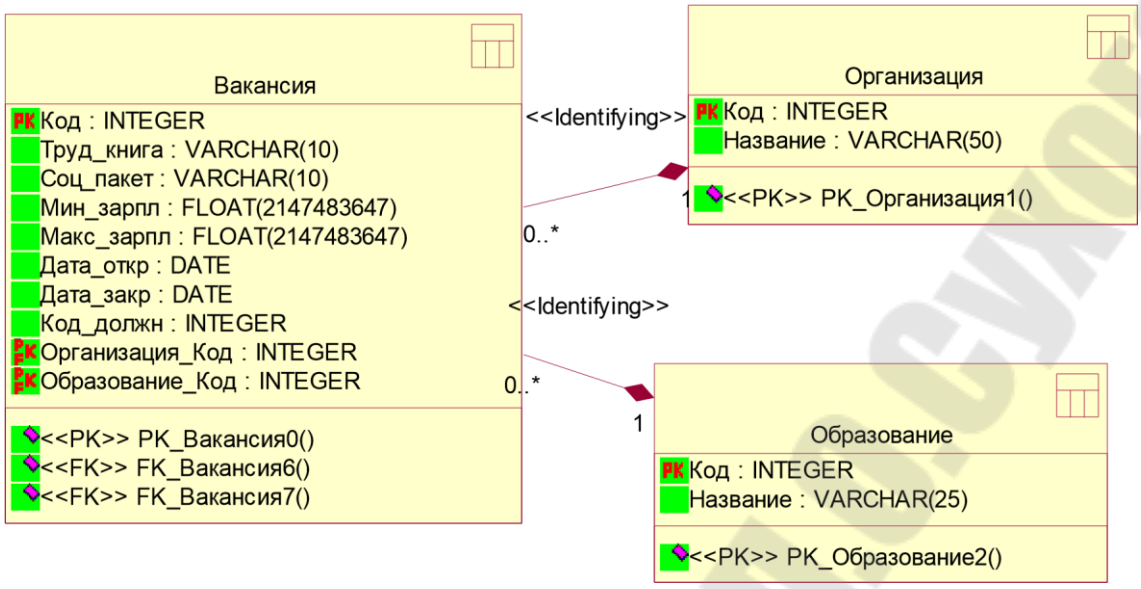


Рисунок 3.4 – Модель данных предметной области

Разработка диаграммы вариантов использования

Диаграмма вариантов использования отображает взаимодействие между вариантами использования, представляющими функции системы, и действующими лицами, представляющими людей или системы, получающие или передающие информацию в данную систему.

Данный тип диаграмм предназначен для создания списка операций, которые выполняет система, поэтому его иногда называют диаграммой функций. Любая система обладает своим множеством вариантов использования и множеством действующих лиц. Каждый вариант использования описывает элемент представляемой системой функциональности. Множество вариантов использования описывает всю функциональность системы на некотором уровне абстракции. Абстракция (abstraction) – сосредоточение на важнейших аспектах приложения и игнорирование всех остальных. Использование абстракции позволяет сохранить свободу принятия решений как можно дольше благодаря тому, что детали не фиксируются раньше времени. Каждое действующее лицо представляет собой один вид объектов, для которых система может выполнять некоторое поведение.

Язык UML предусматривает систему графических обозначений для вариантов использования (рис. 3.5).

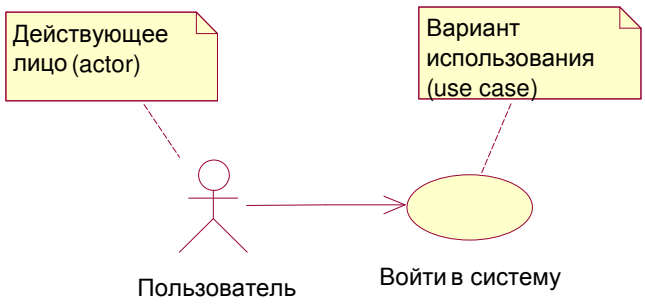


Рисунок 3.5 – Графические обозначения диаграммы вариантов использования

Действующее лицо (actor) – это непосредственный внешний пользователь системы. Это объект или множество объектов, непосредственно взаимодействующих с системой. Каждое действующее лицо является обобщением группы объектов, ведущих себя определенным образом по отношению к системе. Действующими лицами могут быть люди, устройства и другие системы – все, что взаимодействует с интересующей нас системой непосредственно.

Действующее лицо должно иметь одну четко определенную цель. Моделирование действующих лиц помогает определить границы системы, то есть идентифицировать объекты, находящиеся внутри системы, и объекты, лежащие на ее границе.

Различные взаимодействия действующих лиц с системой группируются в варианты использования. Вариант использования (use case) – это связный элемент функциональности, представляемый системой при взаимодействии с действующими лицами (рис. 3.6) .

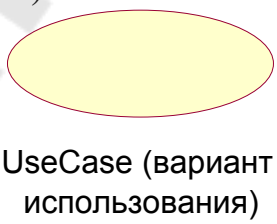


Рисунок 3.6 – Вариант использования

В каждом варианте использования участвуют одно или несколько действующих лиц и система.

Вариант использования объединяет все поведение, имеющее отношение к элементу функциональности системы: нормальное поведение, вариации нормального поведения, исключительные ситуации, сбойные ситуации и отмены запросов.

Любой вариант использования должен иметь краткое описание, объясняющее действия в этом варианте. Описание должно быть кратким, но в него необходимо включить сведения о разных типах пользователей,

выполняющих данный вариант использования, и ожидаемый результат. Во время работы (особенно если проект сложный) эти описания будут напоминать членам команды, почему тот или иной вариант использования был включен в проект и что он должен делать. Четко документируя, таким образом, цели каждого варианта использования, можно уменьшить неразбериху, возникающую среди разработчиков.

На рисунке 3.7 приведен пример документирования варианта использования.

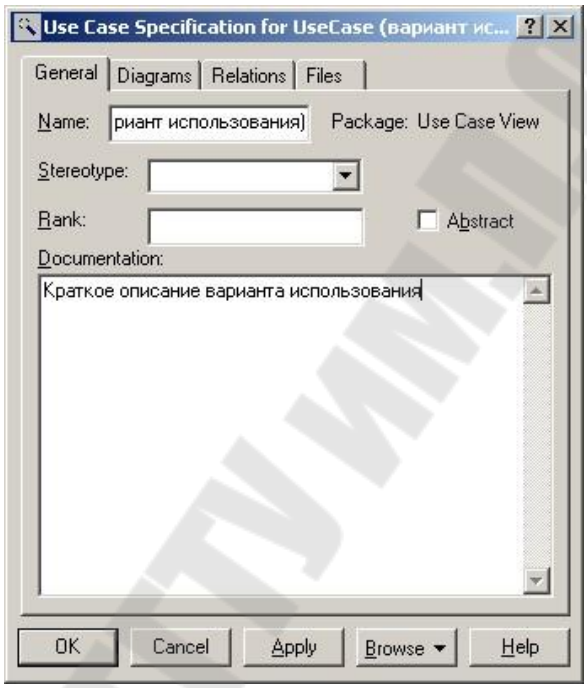
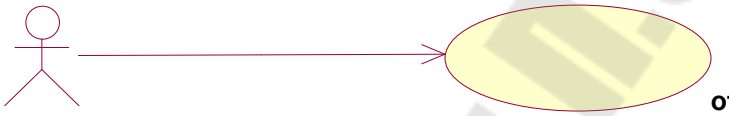


Рисунок 3.7 – Окно документирования варианта использования

Между компонентами диаграммы вариантов использования могут существовать различные отношения, которые описывают взаимодействие экземпляров одних актеров и вариантов использования с экземплярами других актеров и вариантов. Один актер может взаимодействовать с несколькими вариантами использования. В этом случае этот актер обращается к нескольким сервисам данной системы. В свою очередь один вариант использования может взаимодействовать с несколькими актерами, предоставляя для всех них свой сервис. Следует заметить, что два варианта использования, определенные для одной и той же сущности, не могут взаимодействовать друг с другом, поскольку каждый из них самостоятельно описывает законченный вариант использования этой сущности. Более того, варианты использования всегда предусматривают некоторые сигналы или сообщения, когда взаимодействуют с актерами за пределами системы. В то же время могут быть определены другие способы для взаимодействия с элементами внутри системы. В языке UML имеется несколько стандартных видов отношений между действующими лицами и вариантами использования:

- отношение ассоциации;
- отношение зависимости;
- отношение обобщения.

Ассоциация – структурное отношение, описывающее совокупность связей, представленных соединениями между объектами модели. Выделяют разновидность ассоциации, агрегирование, предусмотренное для выражения отношений между целым и его частями.



Действующее UseCase (вариант лицо использования)

Рисунок 3.8 – Пример реализации отношения ассоциации

Применительно к диаграммам вариантов использования оно служит для обозначения специфической роли актера в отдельном варианте использования. Другими словами, ассоциация специфицирует семантические особенности взаимодействия актеров и вариантов использования в графической модели системы. Таким образом, это отношение устанавливает, какую конкретную роль играет актер при взаимодействии с экземпляром варианта использования. На диаграмме вариантов использования, так же как и на других диаграммах, отношение ассоциации обозначается сплошной линией между актером и вариантом использования. Эта линия может иметь дополнительные условные обозначения, такие, например, как имя и кратность

Кратность (multiplicity) ассоциации указывается рядом с обозначением компонента диаграммы, который является участником данной ассоциации. Кратность характеризует общее количество конкретных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации. Применительно к диаграммам вариантов использования кратность имеет специальное обозначение в форме одной или нескольких цифр и, возможно, специального символа "*" (звездочка).

Зависимость – семантическое отношение между двумя сущностями, при которой изменение одной из сущностей, независимой, может повлиять на семантику второй сущности, зависимой.

Отношение зависимости определяет взаимосвязь экземпляров отдельного варианта использования с более общим вариантом, свойства

которого определяются на основе способа совместного объединения данных экземпляров. В метамодели отношение зависимости является направленным и указывает, что применительно к отдельным примерам некоторого варианта использования должны быть выполнены конкретные условия, определенные для расширения данного варианта использования.

Отношение зависимости между вариантами использования обозначается пунктирной линией со стрелкой, направленной от того варианта использования, который является расширением для исходного варианта использования.

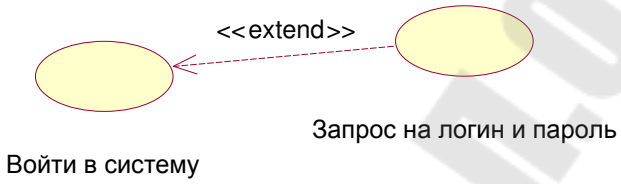


Рисунок 3.8 - Пример графического изображения отношения зависимости (расширения) между вариантами использования

Обобщение – отношение, при котором объект специализированного элемента может быть подставлен и использован вместо объекта обобщенного элемента. Следует подчеркнуть, что потомок наследует все свойства и поведение своего родителя, а также может быть дополнен новыми свойствами и особенностями поведения. Графически данное отношение обозначается сплошной линией со стрелкой в форме незакрашенного треугольника, которая указывает на родительский вариант использования (рис. 2.9).

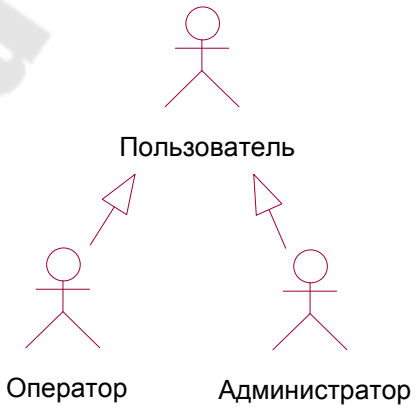


Рисунок 3.9 - Пример графического изображения отношения обобщения между действующими лицами

Особенности разработки диаграмм вариантов использования в среде IBM Rational Rose 2003

Запустите *Rational Rose*, и создайте новую пустую модель. Для этого в окне Create New Model мастера создания моделей, открывающегося после первого запуска системы, включите флажок Don't show this dialog in the future и нажмите кнопку *Cancel*. Эта команда закроет окно мастера, и не будет выводить его при следующих открытиях *Rational Rose*. В рабочем поле *Rational Rose* будет выведена пустое окно диаграммы классов. Это будет наша рабочая модель, в которой и должны быть отражены все нюансы будущей системы. Затем для того чтобы открыть окно диаграммы *Use Case* необходимо сделать двойной щелчок по значку *Main* в папке *User Case View* в окне Browser. Если открыто окно хотя бы одной диаграммы, то в главном меню активизируется пункт Browse и диаграмму *Use Case* можно открыть командой Browse,Use Case Diagram. Отметим, что в панели инструментов Standard нет кнопки просмотра *Browse Use Case Diagram.*Создание новых элементов в диаграмме *Use Case*

Rational *Rose* предоставляет несколько способов создания новых элементов в окне диаграмм *Use Case*:

1. Командой New,Use Case контекстного меню папки Use Case View в окне Browser .
2. Командой Tools,Create,Use Case главного меню.
3. Командами строки инструментов окна Use Case Diagram.

В первом случае элемент создается непосредственно в дереве модели (в папке Use Case View окна Browser), но его значок не включается ни в одну диаграмму. После создания элемента, таким способом, можно поместить его на выбранную диаграмму, например путём перетаскивания мышкой значка элемента из дерева модели окна Browser в окно диаграммы. Во втором и третьем случае вместе с созданием элемента его значок помещается на текущую диаграмму автоматически.

При создании элементов посредством меню Tools программа предоставляет возможность создавать все элементы, которые можно включить в текущую диаграмму, тогда как при создании средствами строки инструментов пользователь ограничен созданием элементов согласно включенным в данную строку значкам. По причине большей простоты и наглядности рекомендуем пользоваться третьим вариантом. Для этого необходимо ознакомиться с содержанием строки инструментов, установленной по умолчанию для данной диаграммы. Для моделирования бизнес-процессов *Rational Rose* предоставляет дополнительные элементы *Use Case*, которые можно активизировать при помощи режима настройки инструментов (командой Use Case Diagram… на вкладке *Toolbars* окна Optins, открываемого командой Tools,Options главного меню). Но для создания

системы учета товародвижения на складе достаточно значков, установленных в панелях инструментов по умолчанию.

Рассмотрим панель инструментов рабочего окна диаграммы Use Case.

Таблица 3.5 - Пиктограммы панели инструментов диаграммы Use Case

Пиктограмма	Кнопка	Описание
	Selects or deselects an item (Выделение или отмена выделения объекта)	Превращает курсор в стрелку указа теля, так что вы можете выделить объект
	Text Box (Текст)	Добавляет к диаграмме текст
	Note (Примечание)	Добавляет к диаграмме примечание
	Anchor Note to Item (Прикрепление примечания к объекту)	Связывает примечание с вариантом использования или объектом на диа-грамме
	Package (Пакет)	Помещает на диаграмму новый пакет
	Use Case (Вариант использования)	Помещает на диаграмму новый вариант использования
	Actor (Действующее лицо	Помещает на диаграмму новое действующее лицо
	Unidirectional Association (Однонаправленная ассоциация)	Рисует связь между действующим лицом и вариантом использования
	Dependency or Instantiates (Зависимость или наполнение)	Рисует зависимость между элементами диаграммы
	Generalization (Обобщение)	Рисует связь использования или расширения между вариантами использования либо рисует связь наследования между действующими лицами

Способы именования элементов и связей

Структурным элементам Actor, Use Case и Package, вводимым в диаграмму, система *Rational Rose* автоматически присваивает умалчиваемые имена (*NewClassN, NewUseCaseN* и *NewPackageN*). Изменит эти имена можно двумя способами:

–редактирования подписи под значком после двойного щелчка на ней левой клавишей мыши (эта команда переводит окно с надписью в режим редактирования текста);

–контекстной командой Rename на соответствующем значке в дереве модели окна Browser;

–активизируя панель спецификаций элемента (двойным левым щелчком мыши на самом значке элемента в окне диаграммы) и редактируя умалчиваемое имя в поле *Name:* этой панели.

Связям умалчиваемые имена не присваиваются. При необходимости их именования можно использовать только третий способ – вводом нужного имени в поле *Name:* окна спецификаций связей. Для открытия этого окна переведите указатель мыши точно на стрелку связи в окне диаграммы и выполните двойной левый щелчок. Должно открыться диалоговое окно Assotiasin Specification for… с полем *Name:*, в которое и вводится имя связи.

Пример разработанной диаграммы вариантов использования

На рисунке 3.7 приведена диаграмма вариантов использования для приложения АИС «Трудоустройство».

Описание предметной области. Организация предоставляет услуги по трудоустройству. Организацией ведется банк данных о существующих вакансиях. По каждой вакансии поддерживается определенная информация. Помимо проектирования реляционной базы данных АИС, необходимо еще и разработать приложение к ней.

Цель приложения: автоматизация информационного процесса определения подходящей вакансии по данным резюме (анкеты) соискателя.

Метод: дискриминантный анализ.

Пользователь: менеджер по работе с кадрами.

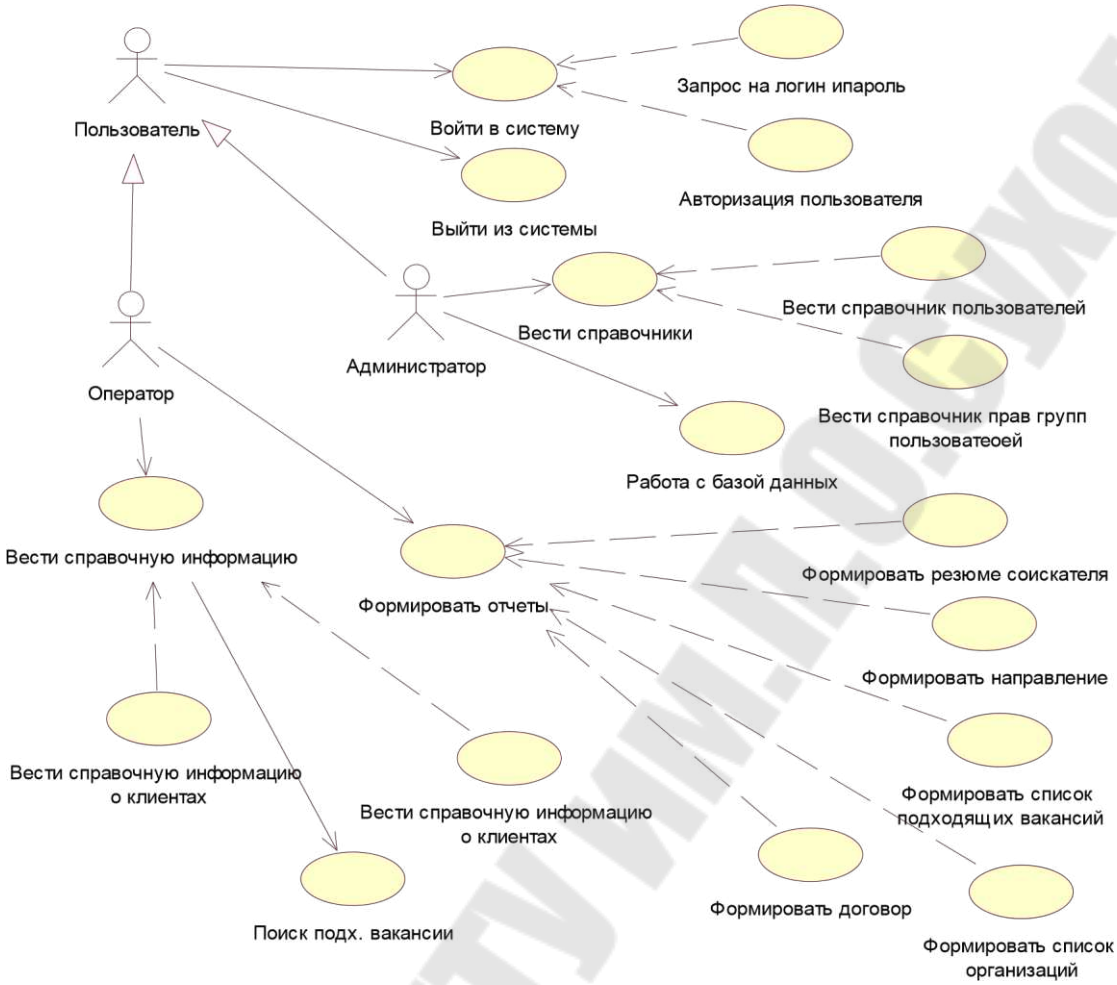


Рисунок 3.11 – Пример диаграммы вариантов использования

Тема 3.2 Параметризация классов

К наиболее важным новшествам версии языка J2SE 5 можно отнести появление параметризованных (generic) классов и методов, позволяющих использовать более гибкую и в то же время достаточно строгую типизацию, что особенно важно при работе с коллекциями. Применение generic-классов для создания типизированных коллекций будет рассмотрено в лекции «Коллекции». Параметризация позволяет создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр. Приведем пример generic-класса с двумя параметрами:

```
package chapt03;

public class Subject <T1, T2> {
private T1 name;
        private T2 id;

        public Subject() {
        }
        public Subject(T2 ids, T1 names) {
                id = ids;
                name = names;
        }
}
```

Здесь T1, T2 – фиктивные объектные типы, которые используются при объявлении членов класса и обрабатываемых данных. При создании объекта компилятор заменит все фиктивные типы на реальные и создаст соответствующий им объект. В качестве параметров классов запрещено применять базовые типы.

Объект класса Subject можно создать, например, следующим образом:

```
Subject<String,Integer> sub =  new
        Subject<String,Integer>();
                char ch[] = {'J','a','v','a'};
Subject<char[],Double> sub2 =
                        new Subject<char[],Double>(ch, 71D );
```

В объявлении sub2 имеет место автоупаковка значения 71D в Double. Параметризированные типы обеспечивают типовую безопасность.

Ниже приведен пример параметризованного класса Optional с конструкторами и методами, также инициализация и исследование поведения объектов при задании различных параметров:

```
package chapt03;

public class Optional <T> {
	private T value;

	public Optional() {
	}
	public Optional(T value) {
		this.value = value;
	}
	public T getValue() {
return value;
	}
	public void setValue(T val) {
		value = val;
	}
	public String toString() {
if (value == null) return null;
	   return value.getClass().getName() + " " + value;
	}
}
package chapt03;

public class Runner {
	public static void main(String[] args) {
	//параметризация типом Integer
		Optional<Integer> ob1 =
				new Optional<Integer>();
		ob1.setValue(1);
		//ob1.setValue("2");// ошибка компиляции: недопустимый тип
		int v1 = ob1.getValue();
		System.out.println(v1);
	//параметризация типом String
		Optional<String> ob2 =
				new Optional<String>("Java");
		String v2 = ob2.getValue();
		System.out.println(v2);
	//ob1 = ob2; //ошибка компиляции – параметризация не ковариантна

	//параметризация по умолчанию – Object
Optional ob3 = new Optional();
```

```
System.out.println(ob3.getValue());
ob3.setValue("Java SE 6");
            System.out.println(ob3.toString());/* выводится
                        тип объекта, а не тип параметризации */

            ob3.setValue(71);
            System.out.println(ob3.toString());

            ob3.setValue(null);
        }
}
```

В рассмотренном примере были созданы объекты типа Optional: ob1 на основе типа Integer и ob2 на основе типа String при помощи различных конструкторов. При компиляции вся информация о generic-типах стирается и заменяется для членов класса и методов заданными типами или типом Object, если параметр не задан, как для объекта ob3. Такая реализация необходима для обеспечения совместимости с кодом, созданным в предыдущих версиях языка.

Объявление generic-типа в виде <T>, несмотря на возможность использовать любой тип в качестве параметра, ограничивает область применения разрабатываемого класса. Переменные такого типа могут вызывать только методы класса Object. Доступ к другим методам ограничивает компилятор, предупреждая возможные варианты возникновения ошибок.

Чтобы расширить возможности параметризованных членов класса, можно ввести ограничения на используемые типы при помощи следующего объявления класса:

```
public class OptionalExt <T extends Тип> { private
    T value;
}
```

Такая запись говорит о том, что в качестве типа T разрешено применять только классы, являющиеся наследниками (суперклассами) класса Тип, и соответственно появляется возможность вызова методов ограничивающих (bound) типов.

Часто возникает необходимость в метод параметризованного класса одного допустимого типа передать объект этого же класса, но параметризованного другим типом. В этом случае при определении метода следует применить метасимвол ?. Метасимвол также может использоваться с ограничением extends для передаваемого типа:

```
/*пример  : использование метасимвола в параметризованном классе:
Mark.java, Runner.java */ package chapt03;

public class Mark<T extends Number> {
```

```
        public T mark;

 public Mark(T value) {   mark =
value;
        }
        public T getMark() {
return mark;
        }
        public int roundMark() {                return
Math.round(mark.floatValue());
        }
        /* вместо */ // public boolean sameAny(Mark<T> ob) {
        public boolean sameAny(Mark<?> ob) {
                return roundMark() == ob.roundMark();
        }
        public boolean same(Mark<T> ob) {
                return getMark() == ob.getMark();
        }
} package
chapt03;

public class Runner {
        public static void main(String[] args) {
        // Mark<String> ms = new Mark<String>("7"); //ошибка компиляции
                Mark<Double> md = new Mark<Double>(71.4D);//71.5d
                System.out.println(md.sameAny(md));
                Mark<Integer> mi = new Mark<Integer>(71);
                System.out.println(md.sameAny(mi));
                // md.same(mi); //ошибка компиляции
System.out.println(md.roundMark());
        }
}
```

В результате будет выведено:

```
true
true 71
```

Метод sameAny(Mark<?> ob) может принимать объекты типа Mark, инициализированные любым из допустимых для этого класса типов, в то время как метод с параметром Mark<T> мог бы принимать объекты с инициализацией того же типа, что и вызывающий метод объект.

Для generic-типов существует целый ряд ограничений. Например, невозможно выполнить явный вызов конструктора generic-типа:

```
class Optional <T> {
        T value = new T();
}
```
так как компилятор не знает, какой конструктор может быть вызван и какой объем памяти должен быть выделен при создании объекта.

По аналогичным причинам generic-поля не могут быть статическими, статические методы не могут иметь generic-параметры или обращаться к generic-полям, например:

*/*пример # : неправильное объявление полей параметризованного класса: Failed.java */*

```
package chapt03;

class Failed <T1, T2> {
        static T1 value;
        T2 id;

        static T1 getValue() {
    return value;
         }
        static void use() {
           System.out.print(id);
         }
    }
```

Параметризованные методы

Параметризованный (generic) метод определяет базовый набор операций, которые будут применяться к разным типам данных, получаемых методом в качестве параметра, и может быть записан, например, в виде:

```
<T extends Тип> returnType methodName(T arg) {}
<T> returnType methodName(T arg) {}
```

Описание типа должно находиться перед возвращаемым типом. Запись первого вида означает, что в метод можно передавать объекты, типы которых являются подклассами класса, указанного после extends. Второй способ объявления метода никаких ограничений на передаваемый тип не ставит.

Generic-методы могут находиться как в параметризованных классах, так и в обычных. Параметр метода может не иметь никакого отношения к параметру своего класса. Метасимволы применимы и к generic-методам.

/ пример : параметризованный метод: GenericMethod.java */*

```
public class GenericMethod { public static <T extends Number>
byte asByte(T num) { long n = num.longValue();
```

```
        if (n >= -128 && n <= 127) return (byte)n;
        else return 0;
      }
      public static void main(String [] args) {
            System.out.println(asByte(7));
            System.out.println(asByte(new Float("7.f")));
            // System.out.println(asByte(new Character('7'))); // ошибка компиляции
      }
}
```

Объекты типа Integer (int будет в него упакован) и Float являются подклассами абстрактного класса Number, поэтому компиляция проходит без затруднений. Класс Character не обладает вышеуказанным свойством, и его объект не может передаваться в метод asByte(T num).

Методы с переменным числом параметров

Возможность передачи в метод нефиксированного числа параметров позволяет отказаться от предварительного создания массива объектов для его последующей передачи в метод:

```
/* пример # : определение количества параметров метода: DemoVarargs.java
*/ package
chapt03;

public class DemoVarargs {     public static int
getArgCount(Integer... args) {          if (args.length
== 0)
  System.out.print("No arg=");   for (int i : args)
                        System.out.print("arg:" + i + "  ");
                return args.length;
        }
        public static void main(String args[]) {
          System.out.println("N=" + getArgCount(7, 71, 555));
          Integer[] i = { 1, 2, 3, 4, 5, 6, 7 };
          System.out.println("N=" + getArgCount(i));
          System.out.println(getArgCount());
        }
   }
```

В результате выполнения этой программы будет выведено:

```
arg:7  arg:71  arg:555  N=3 arg:1  arg:2  arg:3
arg:4  arg:5  arg:6  arg:7  N=7
No arg=0
```

В примере приведен простейший метод с переменным числом параметров. Метод getArgCount() выводит все переданные ему аргументы и возвращает их количество. При передаче параметров в метод из них автоматически создается массив. Второй вызов метода в примере позволяет передать в метод массив. Метод может быть вызван и без аргументов.

Чтобы передать несколько массивов в метод по ссылке, следует использовать следующее объявление:

```
void methodName(Тип[]... args){}
```

Методы с переменным числом аргументов могут быть перегружены:

```
void methodName(Integer...args) {}
void methodName(int x1, int x2) {} void
methodName(String...args) {}
```

В следующем примере приведены три перегруженных метода и несколько вариантов их вызова. Отличительной чертой является возможность метода с аргументом Object... args принимать не только объекты, но и массивы:

```
/* пример : передача массивов: DemoOverload.java */ package
chapt03;

public class DemoOverload {    public static void
printArgCount(Object... args) {//1
        System.out.println("Object args: " + args.length);
        }
        public static void printArgCount(Integer[]...args){//2
        System.out.println("Integer[] args: " + args.length);
        }
        public static void printArgCount(int... args) {//3
                System.out.print("int args: " + +args.length);
        }
        public static void main(String[] args) {
                Integer[] i = { 1, 2, 3, 4, 5 };

                printArgCount(7, "No", true, null);
        printArgCount(i, i, i);              printArgCount(i,
4, 71);              printArgCount(i);//будет вызван
метод 1              printArgCount(5, 7);
                // printArgCount();//неопределенность!
        }
}
```

В результате будет выведено:

```
Object args: 4
Integer[] args: 3
```

Object args: 3
Object args: 5 int
args: 2

При передаче в метод printArgCount() единичного массива i компилятор отдает предпочтение методу с параметром Object... args, так как имя массива является объектной ссылкой и потому указанный параметр будет ближайшим. Метод с параметром Integer[]...args не вызывается, так как ближайшей объектной ссылкой для него будет Object[]...args. Метод с параметром Integer[]...args будет вызван для единичного массива только в случае отсутствия метода с параметром Object...args.

При вызове метода без параметров возникает неопределенность из-за невозможности однозначного выбора.

Не существует также ограничений и на переопределение подобных методов.

Единственным ограничением является то, что параметр вида Тип...args должен быть последним в объявлении метода, например:

```
void methodName(Тип1 obj, Тип2... args) {}
```

Перечисления

Типобезопасные перечисления (typesafe enums) в Java представляют собой классы и являются подклассами абстрактного класса java.lang.Enum. При этом объекты перечисления инициализируются прямым объявлением без помощи оператора new. При инициализации хотя бы одного перечисления происходит инициализация всех без исключения оставшихся элементов данного перечисления.

В качестве простейшего применения перечисления можно рассмотреть следующий код:

```
/* пример # : применение перечисления: SimpleUseEnum.java */ package
chapt02;

enum Faculty {
        MMF, FPMI, GEO
}
public class SimpleUseEnum {
        public static void main(String args[]) {
                Faculty current;
current = Faculty.GEO;
switch (current) {
case GEO:
                        System.out.print(current);
                        break;
                case MMF:
```

```
                System.out.print(current);
                break;
            // case LAW : System.out.print(current);//ошибка компиляции!
        default:
            System.out.print("вне case: " + current);
            }
        }
}
```

В операторах case используются константы без уточнения типа перечисления, так как его тип определен в switch.

Перечисление как подкласс класса Enum может содержать поля, конструкторы и методы, реализовывать интерфейсы. Каждый тип enum может использовать методы:

static enumType[] values() – возвращает массив, содержащий все элементы перечисления в порядке их объявления; static T valueOf(Class<T> enumType, String arg) – возвращает элемент перечисления, соответствующий передаваемому типу и значению передаваемой строки; static enumType valueOf(String arg) – возвращает элемент перечисления, соответствующий значению передаваемой строки; int ordinal() – возвращает позицию элемента перечисления.

```
/* пример #  : объявление перечисления с методом : Shape.java */ package
chapt02;

enum Shape {
        RECTANGLE, TRIANGLE, CIRCLE;
        public double square(double x, double y) {
        switch (this) {             case
RECTANGLE:                  return x * y;
        case TRIANGLE:                  return x *
y / 2;                  case CIRCLE:
                return Math.pow(x, 2) * Math.PI;
            }
            throw new EnumConstantNotPresentException(
                this.getDeclaringClass(),this.name());
        }
}
/* пример : применение перечисления: Runner.java */ package
chapt02;
```

```
public class Runner {      public static void
main(String args[]) {              double x = 2,
y = 3;                 Shape[] arr =
Shape.values();              for (Shape sh : arr)
                       System.out.printf("%10s = %5.2f%n",
                                    sh, sh.square(x, y));
        }
}
```

В результате будет выведено:

```
RECTANGLE =  6,00
 TRIANGLE =  3,00
    CIRCLE = 12,57
```

Каждый из элементов перечисления в данном случае представляет собой в том числе и арифметическую операцию, ассоциированную с методом square(). Без throw данный код не будет компилироваться, так как компилятор не исключает появления неизвестного элемента. Данная инструкция позволяет указать на возможную ошибку при появлении необъявленной фигуры. Поэтому и при добавлении нового элемента необходимо добавлять соответствующий ему case.

```
/* пример: конструкторы и члены перечисления: DeanDemo.java */ package
chapt02;

enum Dean {
    MMF("Бендер"), FPMI("Балаганов"), GEO("Козлевич");
        String name;

        Dean(String arg) {
                name = arg;
        }
     String getName() {   return
    name;
        }
} package
chapt02;

public class DeanDemo {
  public static void main(String[] args) {
    Dean dn = Dean.valueOf("FPMI");
    System.out.print(dn.ordinal());
    System.out.println(" : " + dn + " : " + dn.getName()); }
}
```

В результате будет выведено:

1 : FPMI : Балаганов

Однако на перечисления накладывается целый ряд ограничений.

Им запрещено:

быть суперклассами; быть подклассами; быть абстрактными; создавать экземпляры, используя ключевое слово new.

Тема 3.3 Создание библиотек

jar — программа создания архивов Java

Доступность

JDK версии 1.1 и более поздних версий.

Синтаксис вызова **jar c|t|x[f][m][v] [jar-файл]** ***[файл описания} [файлы}***

Описание

Программа *jar* используется для создания архивных файлов Java (JAR) и работы с ними. JAR-файл представляет собой сжатый ZIP-файл с дополнительным файлом описания. Синтаксис команды *jar* напоминает синтаксис команды *tar* (tape archive — архив на магнитной ленте) ОС UNIX**.**

Параметры командной строки *jar* задаются в виде блока записанных слитно букв, которые передаются в виде одного аргумента, а не через отдельные аргументы командной строки. Первая буква такого аргумента задает необходимое действие, которое должна выполнить программа *jar.* Остальные буквы в этом аргументе являются необязательными. Различные аргументы файлов зависят от того, какие буквы параметров заданы.

Параметры

Первым аргументом командной строки *jar* является набор символов, задающих операцию, которая должна быть выполнена. Первый символ определяет основную операцию и является обязательным. Возможны следующие варианты: **c** Создать новый JAR-архив. В качестве последних аргументов командной

строки *jar* необходимо указать список файлов и/или каталогов.

t Вывести список файлов, содержащихся в JAR-архиве. Если задано имя JAR-файла с помощью параметра f, то список файлов выводится для него. В противном случае имя JAR-файла читается со стандартного устройства ввода.

x Извлечь содержимое JAR-архива. Если задано имя JAR-файла с помощью параметра f, то извлекается содержимое этого файла. В противном случае имя JAR-файла читается со стандартного устройства ввода. Когда командная строка завершается списком файлов и/или каталогов, из JAR-архива

извлекаются только файлы и каталоги, перечисленные в этом списке. В противном случае из архива извлекаются все файлы.

Вслед за идентификатором, определяющим выполняемое действие, могут следовать необязательные параметры:

f Указывает на то, что имя JAR-файла, который необходимо создать, из которого нужно извлечь файлы или получить список содержащихся файлов, задается в командной строке. Если f используется вместе с c, t или x, имя JARфайла должно задаваться в качестве второго аргумента командной строки вызовауаг (т.е. оно должно располагаться непосредственно за блоком параметров). Когда этот параметр не задан, *jar* записывает создаваемый JAR-файл в стандартное устройство вывода или читает его со стандартного устройства ввода.

m Используется только в сочетании с параметром c и указывает на то, что *jar* должна читать файл описания, указанный в командной строке и использовать его в качестве основы для создания описания, которое включается в JAR-файл. Когда этот параметр задается после параметра f, имя файла описания должно указываться после имени создаваемого архива. Если m стоит перед параметром f, то имя файла описания должно предшествовать имени файла создаваемого архива. **v** Описание. Если этот параметр задается вместе с параметром c, тоуаг вы-
водит имя каждого добавляемого в архив файла со статистикой его сжатия. Когда параметр используется в сочетании с t, *jar* выводит список файлов, в котором кроме имени файла содержится его объем и дата последнего изменения. Если v указывается одновременно с x, то *jar* выводит имя каждого извлекаемого из архива файла.

Примеры

Создание простого JAR-архива:

% jar cvf my.jar *.java images

Получение списка содержимого архива:

% jar tvf your.jar

Извлечение файла описания из JAR-файла:

% jar xf the.jar META-INF/MANIFEST.MF Создание

JAR-файла с заданным описанием:

% jar cfmv YesNoDialog.jar manifest.stub oreilly/beans/yesno

Смотри также *javakey*

Java — интерпретатор Java

Подключение внешних библиотек DLL. Native-методы

Для прикладного программирования средств Java в подавляющем большинстве случаев хватает. Однако иногда возникает необходимость подключить к программе ряд системных вызовов. Либо обеспечить доступ к библиотекам, написанным на других языках программирования. Для таких

целей в Java используются методы, объявленные с модификатором native – "родной". Этослово означает, что при выполнении метода производится вызов "родного" для конкретной платформы двоичного кода, а не платформонезависимого байт-кода как во всех других случаях. Заголовок "родного" метода описывается в классе Java, а его реализация осуществляется на какомлибо из языков программирования, позволяющих создавать динамически подключаемые библиотеки (DLL – Dynamic Link Library под Windows, Shared Objects под UNIX-образными операционными системами).

Правило для объявления и реализации таких методов носит название JNI – Java Native Interface.

Объявление "родного" метода в Java имеет вид

Модификаторы native ВозвращаемыйТип имяМетода(список параметров);

Тело "родного" метода не задается – оно является внешним и загружается в память компьютера с помощью загрузки той библиотеки, из которой этот метод должен вызываться:

System.loadLibrary("ИмяБиблиотеки");

При этом имя библиотеки задается без пути и без расширения. Например, если под Windows библиотека имеет имя myLib.dll, или под UNIX или Linux имеет имя myLib.so, надо указывать System.loadLibrary("myLib") ;

В случае, если файла не найдено, возбуждается непроверяемая исключительная ситуация UnsatisfiedLinkError.

Если требуется указать имя библиотеки с путем, применяется вызов System.load ("ИмяБиблиотекиСПутем");

Который во всем остальном абсолютно аналогичен вызову loadLibrary.

После того, как библиотека загружена, с точки зрения использования в программе вызов "родного" метода ничем не отличается от вызова любого другого метода.

Для создания библиотеки с методами, предназначенными для работы в качестве "родных", обычно используется язык C++.

В JDKсуществует утилита javah.exe, предназначенная для создания заголовков C++ из скомпилированных классов Java. Покажем, как ей пользоваться, на примере класса ClassWithNativeMethod. Зададим его в пакете нашего приложения:

```
package java_example_pkg;
public class ClassWithNativeMethod {

   /** Creates a new instance of ClassWithNativeMethod */
  public ClassWithNativeMethod() {
    }

   public native void myNativeMethod();
  }
```

Для того, чтобы воспользоваться утилитой javah, скомпилируем проект и перейдем в папку build\classes. В ней будут располагаться папка с пакетом нашего приложения java_example_pkg и папка META-INF. В режиме командной строки выполним команду javah.exe java_example_pkg.ClassWithNativeMethod

- задавать имя класса необходимо с полной квалификацией, то есть с указанием перед ним имени пакета. В результате в папке появится файл java_example_pkg_ClassWithNativeMethod.h со следующим содержимым:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class java_example_pkg_ClassWithNativeMethod */

#ifndef _Included_java_example_pkg_ClassWithNativeMethod
#define _Included_java_example_pkg_ClassWithNativeMethod
#ifdef __cplusplus extern
"C" {
#endif
/*
* Class: java_example_pkg_ClassWithNativeMethod
* Method: myNativeMethod
* Signature: ()V
 */
JNIEXPORT void JNICALL
Java_java_1example_1pkg_ClassWithNativeMethod_myNativeMethod
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Функция Java_java_1example_1pkg_ClassWithNativeMethod_myNativeMeth od(JNIEnv *, jobject), написанная на C++, должна обеспечивать реализацию метода myNativeMethod() в классе Java. Имя функции C++ состоит из: префикса Java, разделителя "_", модифицированного имени пакета (знаки подчеркивания "_" заменяются на "_1"), разделителя "_", имени класса, разделителя "_", имени "родного" метода. Первый параметр JNIEnv * в функции C++ обеспечивает доступ "родного" кода к параметрам и объектам, передающимся из функции C++ в Java. В частности, для доступа к стеку. Второй параметр,jobject, – ссылка на экземпляр класса, в котором задан "родной" метод, для методов объекта, и jclass – ссылка на сам класс– для методов класса. В языке C++ нет ссылок, но в Java все переменные объектного

типа являются ссылками. Соответственно, второй параметр отождествляется с этой переменной.

Если в "родном" методе имеются параметры, то список параметров функции C++ расширяется. Например, если мы зададим метод public native int myNativeMethod(int i); то список параметров функции C++ станет

(JNIEnv *, jobject, jint)

А тип функции станет jint вместо void.

Таблица 3.5. Соответствие типов Java и C++

Тип Java	Тип JNI (C++)	Характеристика типа JNI
boolean	jboolean	1 байт, беззнаковый
byte	jbyte	1 байт
char	jchar	2 байта, беззнаковый
short	jshort	2 байта
int	jint	4 байта
long	jlong	8 байт
float	jfloat	4 байта
double	jdouble	8 байт
void	void	-
Object	jobject	Базовый для остальных классов
Class	jclass	Ссылка на класс Java
String	jstring	Строки Java
массив	jarray	Базовый для классов массивов
Object[]	jobjectArray	Массив объектов
boolean[]	jbooleanArray	Массив булевских значений
byte[]	jbyteArray	Массив байт (знаковых значений длиной в байт)
char[]	jcharArray	Массив кодов символов
short[]	jshortArray	Массив коротких целых
int[]	jintArray	Массив целых
long[]	jlongArray	Массив длинных целых
float[]	jfloatArray	Массив значений float
double[]	jdoubleArray	Массив значений double
Throwable	jthrowable	Обработчик исключительных ситуаций

В реализации метода требуется объявить переменные. Например, если мы будем вычислять квадрат переданного в метод значения и возвращать в качестве результата значение параметра, возведенное в квадрат (пример чисто учебный), код реализации функции на C++ будет выглядеть так:

```
#include "java_example_pkg_ClassWithNativeMethod.h"
JNIEXPORT jint JNICALL
Java_java_1example_1pkg_ClassWithNativeMethod_myNativeMethod
  (JNIEnv *env, jobject obj, jint i ){
    return i*i };
```

Отметим, что при работе со строками и массивами для получения и передачи параметров требуется использовать переменнуюenv. Например, получение длины целого массива, переданного в переменную jintArray intArr, будет выглядеть так:

```
jsize length=(*env)->GetArrayLength(env, intArr);
```

Выделение памяти под переданный массив: jint

```
*intArrRef=(*env)->GetIntArrayElements(env, intArr,0);
```

Далее с массивом intArr можно работать как с обычным массивом C++. Высвобождение памяти из-под массива:

```
(*env)->ReleaseIntArrayElements(env, intArr, intArrRef ,0);
```

Имеются аналогичные функции для доступа к элементам массивов всех примитивных типов: GetBooleanArrayElements, GetByteArrayElements,…, GetDoubleArrayElements. Эти функции копируют содержимое массивов Java в новую область памяти, с которой и идет работа в C++. Для массивов объектов имеется не только функция GetObjectArrayElement, но иSetObjectArrayElement – для получения и изменения отдельных элементов таких массивов.

Строка Java jstring s преобразуется в массив символов C++ так:

```
const char *sRef=(*env)->GetStringUTFChars(env,s,0);
```

Ее длина находится как int s_len=strlen(sRef) ; Высвобождается из памяти как

```
(*env)->ReleaseStringUTFChars(env,s,sRef);
```

Краткие итоги

Программу, выполняющуюся под управлением операционной системы, называют процессом (process), или, что то же, приложением. У каждого процесса свое адресное пространство. Потоки выполнения (threads) отличаются от процессов тем, что выполняются в адресном пространстве своего родительского процесса. Потоки выполняются параллельно (псевдопараллельно), но, в отличие от процессов, легко могут обмениваться данными в пределах общего виртуального адресного пространства. То есть у них могут иметься общие переменные, в том числе – массивы и объекты.

В приложении всегда имеется главный (основной) поток. Если он закрывается – закрываются все остальные пользовательские потоки приложения. Кроме них возможно создание потоков-демонов, которые могут продолжать работу и после окончания работы главного потока.

Любая программа Java неявно использует потоки. В главном потоке виртуальная Java-машина (JVM) запускает метод main приложения, а также все методы, вызываемые из него. Главному потоку автоматически дается имя "main".

Если разные потоки получают доступ к одним и тем же данным, причем один из них или они оба меняют эти данные, для них требуется обеспечить установить разграничение доступа. Пока один поток меняет данные, второй не должен иметь права их читать или менять. Он должен дожидаться окончания доступа к данным первого потока. Говорят, что осуществляетсясинхронизация потоков. В Java для этих целей служит опера-
тор synchronize ("синхронизировать"). Иногда синхронизованную область кода (метод или оператор) называют критической секцией кода.

При запуске синхронизованного метода говорят, что объект входит в монитор, при завершении – что объект выходит из монитора. При этом поток, внутри которого вызван синхронизованный метод, считается владельцем данного монитора.

Имеется два способа синхронизации по ресурсам: синхронизация объекта и синхронизация метода.

Синхронизация объекта obj1 при вызове несинхронизованного метода:

synchronized(obj1) оператор;

Синхронизация метода с помощью модификатора synchronized при задании класса: public synchronized тип метод(...){...}

Кроме синхронизации по данным имеется синхронизация по событиям, когда параллельно выполняющиеся потоки приостанавливаются вплоть до наступления некоторого события, о котором им сигнализирует другой поток. Основными операциями при таком типе синхронизации являются wait ("ждать") и notify ("оповестить").

Имеется два способа создать класс, экземплярами которого будут потоки: унаследовать класс от java.lang.Thread либо реализовать интерфейс java.lang.Runnable.

Интерфейс java.lang.Runnable имеет декларацию единственного метода public void run(), который обеспечивает последовательность действий при работе потока. Класс Thread уже реализует интерфейс Runnable, но с пустой реализацией метода run(). Поэтому в потомке Thread надо переопределить метод run().

При работе с большим количеством потоков требуется их объединение в группы. Такая возможность инкапсулируется классом TreadGroup ("Группа потоков").

Для получения доступа к библиотекам, написанным на других языках программирования, в Java используются методы, объявленные с модификатором native – "родной". При выполнении такого метода производится вызов "родного" для конкретной платформы двоичного кода, а не платформонезависимого байт-кода как во всех других случаях. Заголовок "родного" метода описывается в классе Java, а его реализация осуществляется на какомлибо из языков программирования, позволяющих создавать динамически подключаемые библиотеки.

Тема 3.4 Java и XML

Так как программы прошли большой путь от выполняющихся на одной машине до распределенных систем, разработчики постоянно сталкиваются с новыми технологиями. Приложения выполняются не только на разных компьютерах, но и на различных операционных системах, на различных платформах. Разработчики вынуждены определять механизмы взаимодействия приложений, написанных на разных языках программирования и выполняемых на разных операционных системах.

Если все приложения написаны на Java, все легко. Java изначально предназначена для использования на разных операционных системах и платформах, лишь виртуальная машина Java должна быть установлена на компьютере. На самом деле могут возникнуть некоторые вопросы, связанные, например, с тем, что Java использует 4 байта для представления целых, а некоторые языки программирования используют два байта, а что делать с другими байтами. Порядок байтов различается в Windows и Unix: если целое представлено 4 байтами, то порядок байтов в разных операционных системах различен, будут ли значения отличаться.

Как уже было сказано, не будет проблем, если программы написаны на Java. На самом деле, далеко не всегда можно все написать на Java. Если приложения непосредственно работают с аппаратурой, то они должны учитывать особенности архитектуры и аппаратуры. Это только один пример, когда Java не может использоваться. Другой пример – операционная система, для которой нет Java виртуальной машины.

Теперь, когда Вы поняли, что необходимо связывать между собой разные приложения, зададим вопрос: как приложения будут обмениваться данными? Как переслать значение, например, 65? Как набор битов целого типа? Нет, целые значения представлены по-разному на разных платформах. Может быть, использовать пересылку значения как набора двух символов, символа 6, а за ним – символа 5? Символьная информация на разных языках программирования представлена одинаково.

Это работает, но как определить смысл значения, например, что 65 – скорость чего-то? Можно переслать строку в виде:
speed=65

Это можно использовать для примитивных типов, а если надо переслать объект? Надо переслать состояние объекта, чтобы на другой машине можно было бы восстановить объект по состоянию. Это сложнее. Давайте рассмотрим пересылку объекта Car из программы на Java, работающей под Windows
(операционная система не важна для Java), в программу на C++ , работающей под Sun Solaris.

Рассмотрим следующий класс Car:

```
public class Car {    private Color
color = Color.red;    private String
make = "Porsche";
  private String model = "Carerra 911";

  private int gas = 15; // in gallons
private int speed = 0; // in mph    private
int oil = 5; // in quarts
  private boolean running = false;

  ... methods ...
}
```

Мы можем пересылать строки между приложениями, чтобы разрешить проблемы разного представления данных. Как же представить в виде строк объект типа Car? Рассмотрим передачу следующей переменной в другую протграмму:

Car myCar = new Car();

Итак, мы хотим переслать переменную myCar со всеми данными. Было бы хорошо установить соотношения между параметрами автомобиля и их значениями, причем должна быть иерархическая структура строк: объект автомобиль содержит такие-то атрибуты. Мы не будем изобретать колесо, так как язык HTML обладает следующими свойствами:

- Теги HTML определяют иерархическую модель.
- HTML работает на любых операционных системах и для любых языков программирования.

Основа HTML документа - тег <html>, его вложенные теги - <head> и <body>. Тег <head> может содержать элемент <title> или элемент <script>. Элемент <body> может содержать атрибут bgcolor, который задает цвет фона страницы,

элемент <table> может включать элементы <tr>и т. д. Каждый элемент начинается с тега, который имеет имя <tag-name>. Каждый элемент заканчивается тегом с тем же именем, но с косой чертой: </tag-name>. Например:

```
<html>
  <head>
    <title>My Page</title>
  </head>
  <body bgcolor="black">
    ... body content ...
  </body>
</html>
```

Хотя HTML-документ имеет много тегов, они не могут описать автомобиль. Давайте составим свои теги для описания автомобиля, например, так:

```
<car name="myCar">
  <color>yellow</color>
  <make>Porsche</make>
  <model>Carerra 911</model>
  <gas>15</gas>
  <oil>5</oil>
  <speed>95</speed>
  <running>true</running>
</car>
```

Это описание автомобиля имеет следующие характеристики:

- Оно очень понятно. Вы видите, что автомобиль <car>имеет имя myCar, а скорость его - <speed> имеет значение 95.
- Возможна пересылка этого описания в любую операционную систему и язык программирования.
- Описание отражает иерархическую структуру объектноориентированного программирования.

Это и есть суть XML: Он основан на иерархической структуре, подобной HTML, но имена тегов и данные, которые они содержат, произвольны и содержательны. Они описывают структуру данных. Например, элемент <car> может содержать элемент <speed>, но элемент <speed> не может содержать элемент <car>, элемент <speed> может отсутствовать и считаться равным 0, если его нет, и т. д.

Документ XML

Основа XML документа – сам документ. Он включает следующие компоненты:

– Пролог – содержит информацию о версии, комментарии и ссылки на файлы Documenet Type Definition (DTD).
– Тело – содержит корень документа и его элементы.
– Эпилог:- содержит комментарии и инструкции по обработке.

Листинг содержит простой файл XML, который описывает книги, хранимые на полке или в библиотеке:

```
1:  <?xml version="1.0"?>
2:  <!DOCTYPE books SYSTEM "Books.dtd">
3:  <books>
4:    <book category="computer-programming">
5:      <author>Steven Haines</author>
6:      <title>Java 2 From Scratch</title>
7:      <price>39.95</price>
8:    </book>
9:    <book category="fiction">
10:     <author>Tim LaHaye</author>
11:     <title>Left Behind</title>
12:   </book>  13:
</books>
```

Строки 1 и 2 – это пролог. Строка 1 определяет версию XML документа - 1.0; а строка 2 ссылается на файл Document Type Definition (DTD), который определяет синтаксические правила для данного файла XML.

Строки с 3 по 13 – тело XML документа. Большинство документов XML, также как и этот, могут не иметь эпилога, но спецификация позволяет это.

Проверка документа XML

Когда документ разбирается или читается парсером (процесс, ответственный за разбор документа), тот может проверить синтаксис XML документа по файлу DTD и определить, правильно ли сформирован документ. Если документ верен, то из него можно извлекать данные по правилам, изложенным в DTD; в противном случае документ не рассматривается.

Sun обеспечивает с помощью Java API для XML Parsing (JAXP) обеспечивает два способа чтения XML документов:

− Модель событий, −

Модель дерева.

JDOM – открытый проект, который обеспечивает механизм работы с XML документами с помощью классов коллекций Java. Входящий в Simple API для XML (SAX) парсер – управляется событиями: программа регистрирует прослушиватель для парсера, а парсер получает данные из потока, определяя XML элементы. Модель Document Object Model (DOM) строит дерево представления XML документа и обеспечивает Application Programmers Interface (API) для доступа к данным в дереве.

SAX и DOM используются с разными целями и имеют свои достоинства и недостатки. Парсер SAX использует малый объем памяти, потому что он не сохраняет информацию, а просто просматривает документ и сообщает о тех элементах, которые обнаружены. Он достаточно быстр, но при его использовании надо программисту самому строить структуры в памяти для сохранения данных из документа. Модель DOM медленная и требует много памяти, но она обеспечивает сохранение данных в памяти, дает возможность использовать существующие классы и методы для доступа к данным.

Если Вы используете XML, чтобы загружать данные в Ваши структуры, то лучше применять SAX. Если же Вам нужно найти какое-то значение, содержащееся в документе XML или вычислить что-то на основе данных в XML (например, подсчитать количество книг), следует использовать SAX. С другой стороны, если Вам нужен доступ ко всему документу в памяти, следует применять DOM. Если Вы изменяете содержимое документа, а затем выводите модифицированные данные (например, сохраняете в файл), следует применять DOM.

Выбор обусловлен тем, как Вы собираетесь использовать данные, полученные из документа XML.

Все классы, необходимые для работы с XML, включены в JSDK, начиная с версии 1.5.

Разбор XML документов с помощью Simple API для XML (SAX)

Использование SAX-разбора для XML документа требует наличия SAX-парсера и обработчика той информации, которая будет получена от парсера.

JAXP обеспечивает класс SAX -парсер: javax.xml.parsers.SAXParser; причем SAX парсер доступен через класс javax.xml.parsers.SAXParserFactory. Класс SAXParserFactory – это набор методов для получения SAX-парсеров. Вызов метода newSAXParser() позволяет получить ссылку на сконфигурированный SAX-парсер. Оба эти класса – абстрактные, поэтому Вы должны обеспечить реализацию некоторых методов классов.

Ваша задача – создать обработчик документа. Он определяется классом org.xml.sax.helpers.DefaultHandler. Класс реализует несколько интерфейсов, но самый интересный - интерфейс org.xml.sax.ContentHandler.

Эти методы вызывает парсер документа. Когда начинается разбор документа, вызывается метод startDocument(), когда найден элемент <books>, вызывается метод startElement().

Класс DefaultHandler реализует интерфейсы org.xml.sax.DTDHandler, org.xml.sax.EntityHandler и org.xml.sax.ErrorHandler. этот класс предназначении для обработки ошибок при разборе. Для реализации обраблотчика лучше, чтобы он наследовал от класса DefaultHandler.

Обрабатываем сообщения SAX-парсера в методах нашего класса

DefaultHandler.

```
001: public class SAXBook {
002:    private String title;
003:    private String author;
004:    private String category;
005:    private float price;  006:
007:    public SAXBook() {
008:    }  009:
010:    public SAXBook( String title,
011:                    String author,
012:                    String category,
013:                    float price ) {
014:       this.title = title;
015:       this.author = author;
016:       this.category = category;
017:       this.price = price;
018:    }  019:
020:    public String getTitle() {
021:       return this.title;
022:    }  023:
024:    public void setTitle( String title ) {
025:       this.title = title;
026:    }  027:
028:    public String getAuthor() {
029:       return this.author;
030:    }  031:
032:    public void setAuthor( String author ) {
033:       this.author = author;
034:    }  035:
036:    public String getCategory() {
037:       return this.category;
038:    }  039:
040:    public void setCategory( String category ) {
041:       this.category = category;
042:    }  043:
044:    public float getPrice() {
045:       return this.price;
046:    }  047:
048:    public void setPrice( float price ) {
```

```
049:        this.price = price;
050:    }  051:
052:    public String toString() {
053:        return "Book: " + title + ", " + category + ", " +
author + ", " + price;
054:    }
055:}
```

Листинг – это простой JavaBean для полей класса: title, author, category и price. Определен также метод toString() для получения значений всех полей.

```
001:import java.util.ArrayList;   002:
003:public class SAXBooks {
004:    private ArrayList bookList = new ArrayList();  005:
006:    public SAXBooks() {
007:    }  008:
009:    public void addBook( SAXBook book ) {
010:        this.bookList.add( book );
011:    }  012:
013:    public SAXBook getBook( int index ) {
014:        if( index >= bookList.size() ) {
015:            return null;
016:        }
017:        return( SAXBook )bookList.get( index );
018:    }  019:
020:    public SAXBook getLastBook() {
021:        return this.getBook( this.getBookSize() - 1 );
022:    }  023:
024:    public int getBookSize() {
025:        return bookList.size();
026:    }  027:}
```

Листинг показывает класс SAXBooks, поддерживающий коллекцию объектов SAXBook в массиве java.util.ArrayList и обеспечивающий методы для вставки книги, получения данных о книге, получения общего числа книг. В этом классе есть еще один метод getLastBook(), который возвращает последнюю книгу в массиве ArrayList. Этот метод используется в файле в листинге:

```
001:import java.io.*;
002:import org.xml.sax.*;
003:import org.xml.sax.helpers.DefaultHandler;
004:import javax.xml.parsers.SAXParserFactory;
005:import javax.xml.parsers.ParserConfigurationException;
006:import javax.xml.parsers.SAXParser;  007:
```

```
008:public class SAXSample {
009:    public static void main( String[] args ) {
010:        try {
011:            File file = new File( "book.xml" );
012:            if( !file.exists() ) {
013:                System.out.println( "Couldn't find file..." );
014:                return;
015:            }
016:
017:            // Use the default (non-validating) parser
018:            SAXParserFactory factory = SAXParserFactory.newInstance();  019:
020:            // Create an instance of our handler  021:
MyHandler handler = new MyHandler();  022:
023:            // Parse the file
024:            SAXParser saxParser = factory.newSAXParser();
025:            saxParser.parse( file, handler );  026:
SAXBooks books = handler.getBooks();  027:
028:            for( int i=0; i<books.getBookSize(); i++ ) {
029:                SAXBook book = books.getBook( i );
030:                System.out.println( book );
031:            }  032:
033:        }
034:        catch( Throwable t ) {  035:
t.printStackTrace();
036:        }
037:    }
038:}
```

Список использованных источников

1. Дейтел Х. М., Дейтел П. Д., Сантри Технология программирования на Java2. Книга 1. Графика, JavaBeans, интерфейс пользователя. - М.: Бином, 2003. ISBN: 5-9518-0017-X.
2. Дейтел Х. М., Дейтел П. Д., Сантри Технология программирования на Java2. Книга 2. Распределенные приложения. - М.: Бином, 2003. ISBN: 5-9518-0051-X.
3. Дейтел Х. М., Дейтел П. Д., Сантри Технология программирования на Java2. Книга 3. Корпоративные системы, сервлеты, JSP, web-сервисы. - М.: Бином, 2003. ISBN: 5-9518-0034-X.
4. Дейтел Х. М., Дейтел П. Д. Как программировть на Java. Книга 1. Основы программирования. - М.: Бином, 2003. ISBN: 5-9518-0015-3.
5. Дейтел Х. М., Дейтел П. Д. Как программировть на Java. Книга 2. Файлы, сети, базы данных. - М.: Бином, 2006. ISBN: 5-9518-0127-3.
6. Ноутон П., Шилдт Г. Java2. Наиболее полное руководство. - СПб.: BHV, 2001. ISBN: 0-07-211976-4, 5-94157-012-0.
7. Биллиг В.А. Основы программирования на Java/ - М.: Изд-во «Интернетуниверситет информационных технологий – ИНТУИТ.ру», 2006. – 488с.
8. Шилдт Г.С. Java: Учебный курс. – СПб.:Питер, 2002. – 512с.
9. Шилдт Г.С. Полный справочник по Java. – М.: Издательский дом «Вильямс», 2004. – 752с.