# Элементы J2EE

Александр Владимирович Коновалов
avkon@imm.uran.ru

6 ноября 2002 г.

## Содержание



## 1 Чем мы займёмся и кому это надо

Если отвечать на первый вопрос коротко и содержательно, то “J2EE”. Более развёрнутый ответ: “Программными средствами создания больших систем на примере архитектуры Java2 Enterprise Edition”. Со 2-м вопросом ситуация следующая: разные языковые тонкости, конечно же, нас не интересуют: что нам за дело до `return`

в `try` и `finally`, мы ведь транслятор писать не собираемся. Ситуация с, к примеру, Message Driven Bean-ами, по-моему, совершенно другая: можно утверждать, что в любой достаточно сложной системе нечто подобное возникнет. Лучше, поэтому, познакомиться с промышленным стандартом, обсудить тонкости реализации, изучить типичные решения, включая типичные неправильные решения и пр., дабы не изобретать велосипеда.

Почему именно J2EE?

- Она существует в реальности. Каждый в меру образованный программист запросто назовёт полдюжины великолепных программных систем, которые решили все проблемы нашего нынешнего окружения, отличаются необыкновенной концептуальной целостностью и пр. В отличие от Intel 432 и Taligent, J2EE реально используется для зарабатывания денег.

- Существуют несколько независимых реализаций каждой спецификации. Это гарантирует, что спецификация нечто специфицирует, что по ней можно что-то написать, позволяет оценить на практике иногда совершенно разные подходы, выбранные реализаторами.

- Можно (тем или иным способом) посмотреть в исходные тексты. Если вы не Ричард Столлмен (или, совсем наоборот, не *журналист*), тексты вам нужны совсем не для того, чтобы "исправлять ошибки". Главное в текстах — возможность их **читать**. Формирующемуся профессионалу полезно посмотреть в тексты, чтобы извести взгдяд на большие программные системы как на магический продукт психотворчества сверхлюдей. Наконец, слабым местом современного Университетского образования является отсутствие навыков чтения чужих (иногда совершенно безобразных) текстов[1].

- Всё это довольно большое. Спецификация EJB 2.0 занимает 573 страницы, "базовый" сервер приложений JBoss 3.0 состоит из 490 тыс. строк, спецификация сервлетов 2.2 — 79 страниц, Tomcat, её реализующий — 93 тыс. строк. Для сравнения: ядро Линукса 2.4.7 — 3 млн. строк, а Перла 5.6.1 — 153 тысячи. Каждому ясно, что программистские проблемы, достойные так называться, начинаются лишь с какого-то размера. Так вот: здесь размера, видимо, хватает.

Естественно, такой подход стимулирует настроение поскорее "проскочить" унылые подробности языка и сосредоточиться на вещах интересных.

По ходу изложения легко будет заметить, что некоторые особенности Явы вызывают у автора неприятие либо, временами, недоумение. Может возникнуть вопрос — если всё ужасно, зачем её вообще изучать? Причины, в принципе, приведены чуть выше, можно добавить лишь мысль Стауструпа (по поводу выбора C как основы языка): "Во вновь спроектированном языке были бы свои недостатки, а недостатки C нам хорошо известны".

Изложение строится так: сначала мы разбираем содержательный пример, знакомясь по ходу дела с полезными возможностями Явы, а затем смотрим вокруг реализации примера и пытаемся осознать, какие реализационные вещи из примера можно сделать по-другому и зачем. Платой за такой стиль изложения будет некая хаотичность, но уж очень тоскливо в двадцать десятый раз приводить "общую структуру цикла while". Поэтому не переживайте, если тонкости синтаксиса вам непонятны: наша цель — осознать идеи, а синтаксис может быть любым.

Покончим с публицистикой и попытаемся начать.

[1] Автор с восторгом наблюдал популярного победителя разных (программистских) олимпиад, сообщяюшего, что ему очень трудно разобраться в 200-строчечной программе на C! Восторг вызвало, естественно, не стремление ничего не делать в магистратуре, а то, что человек с программистской специальности и работающий программистом таких аргументов не стесняется.

## 2 Содержательный пример

В качестве введения в язык обсудим решение задачи про обработку лога на Яве.

```
package ru.test;

import java.io.*;
import java.util.*;
import java.util.regex.*;

public class Test {
    public static void main(String[] args) throws Exception {
        long beg_time = System.currentTimeMillis();
          // map from url to hashes with hostnames
        HashMap url2hosts = new HashMap();
          // map from url to num of uniq hosts at this url
        final HashMap url2hostnum = new HashMap();
        String s;
        ArrayList urls = new ArrayList();
        Object ok = new Object();
        ArrayList ignorePatt = new ArrayList();
        String []ignoreURLs = {
                       // \Q quotes . et al so that it just . not any symbol
            "\\Q/al.css\\E",
            "\\Q/scripts/root.exe?/c+dir\\E",
            "/[cd]\\Q/winnt/system32/cmd.exe?/c+dir\\E",
            "\\Q/scripts/..%255c../winnt/system32/cmd.exe?/c+dir\\E",
            "/(_vti_bin|_mem_bin)\\Q/..%255c../..%255c../..%255c../" +
                "winnt/system32/cmd.exe?/c+dir\\E",
            "\\Q/scripts/..\\E(%C1%1C|%c0%2f|%C0%AF|%C1%9C)\\Q../" +
                "winnt/system32/cmd.exe?/c+dir\\E",
            "\\Q/msadc/..%255c../..%255c../..%255c/..%C1%1C../..%C1%1C.." +
                "/..%C1%1C../winnt/system32/cmd.exe?/c+dir\\E",
            "\\Q/MSADC/root.exe?/c+dir\\E"        };

        for (int i=0; i<ignoreURLs.length; i++) {
            ignorePatt.add(Pattern.compile(ignoreURLs[i]));
        }
        Pattern
            quoteURLs = Pattern.compile("^(\\S+) .+\\\"GET (\\S+) HTTP");

        BufferedReader in =
            new BufferedReader(new FileReader("d:/access_log"));
        Pattern logLine = Pattern.compile("^(\\S+) .+\\\"GET (\\S+) HTTP");
LINE:   while ( null != (s = in.readLine()) ) {
            Matcher m = logLine.matcher(s);
            if (! m.find())
                continue;
            String host = m.group(1); # 1-st braces contents
            String url = m.group(2);  # 2-nd ...

            for (Iterator e = ignorePatt.iterator(); e.hasNext(); ) {
                Pattern p = (Pattern)e.next();
                Matcher m1 = p.matcher(url);
                if (m1.matches())
                    continue LINE;
            }
            if (null != url2hosts.get(url))
                if (((HashMap)url2hosts.get(url)).containsKey(host))
                    continue;          // this user already be there
                else {                 // first uniq user
```

```
                ((HashMap)url2hosts.get(url)).put(host, ok);
                Integer oldNum = (Integer)url2hostnum.get(url); // *
                Integer newNum = new Integer(oldNum.intValue()+1);
                url2hostnum.put(url, newNum);
            }
        else {                      // increase num of uniq users
            url2hosts.put(url, new HashMap());
            ((HashMap)url2hosts.get(url)).put(host, ok);
            urls.add(url);
            url2hostnum.put(url, new Integer(1));
        }
    }
    in.close();
    Collections.sort(urls, new Comparator() {
        public int compare(Object o1, Object o2) {
            String u1 = (String)o1, u2 = (String)o2;
            int n1 = ((Integer)url2hostnum.get(u1)).intValue(); // **
            int n2 = ((Integer)url2hostnum.get(u2)).intValue();
            if (n1 > n2)
                return -1;
            else
                if (n1 == n2)
                    return 0;
                else
                    return 1;
        }
    });
    long end_time = System.currentTimeMillis();
    int i=0;
    for (Iterator e = urls.iterator(); e.hasNext() && i<20; i++) {
        String url = (String)e.next();
        System.out.println(url+"\t"+url2hostnum.get(url));
    }
    System.out.println((end_time-beg_time)/1000. + " sec.");
  }
}
```

Всё, в общем, понятно, исключая некоторые тонкости.

Все типы данных языка — это примитивные типы (т.е. `int` и пр.) либо указатели на объекты. Именно из-за того, что кругом указатели, мы и пишем `new`. Об указательной сути не нужно забывать, вызывая процедуры: если вы поменяете объект в вызвавшей, то в вызванной он тоже, естественно, поменяется.

Начиная с версии 1.4, Ява снабжена “стандартными” регулярными выражениями, и именно они используются в примере. Конечно же, существует множество других реализаций регулярных выражений для всевозможных JDK (а вот сколько-нибудь объективного их сравнения я не встречал). Что здесь нужно понимать — что если в задачке просятся регулярные выражения, то лучше взять готовую библиотеку, а не громоздить очередной кривой ad hoc парсер[2].

В стандартные библиотеки входят основные структуры данных, требуемые в реальной жизни: динамические массивы, хеши, (красно-чёрные) деревья и пр. Разглядывая документацию, легко обнаружить в этой области некое дублирование: к примеру, есть Vector, и есть ArrayList, есть Hashtable, и есть HashMap. Ситуация здесь простая: часть классов унаследованные, а часть — новые, от того и параллелизм. Совет: используйте новые контейнеры, если нет причин делать наоборот (их можно отличить по надписи Since: JDK 1.2 в документации).

У Яовской иерархии классов есть один корень — класс Object. Это используется при организации контейнеров: в контейнер помещаются и из контейнера достают-

[2]Это, понятно, не только к Яве относится. Для каждого более-менее популярного языка существуют реализации регулярных выражений.

ся (ссылки на) Object-ы, то есть контейнер может одновременно хранить объекты разных типов, а извлечённый Object перед использованием нужно привести к *ожидаемому* типу. Никакого нарушения защиты тут не происходит, ведь если реальный тип отличается от ожидаемого, вместо приведения произойдет `ClassCastException`.

Ещё одна забавная особенность разбираемой программки — манипуляции с Integer (см. (*), (**)). Идея здесь такая: как уже говорилось, есть полноценные объекты и есть примитивные типы. Так вот, примитивные типы не наследуют от Object, и потому в контейнер помещены быть не могут! Но рядом с каждым примитивным типом живёт тип-оболочка, хранящий значения соответствующего примитивного типа, а вот его помещать в контейнер уже можно. Интересной особенностью объектов-оболочек является то, что они неизменяемы. Как таковой, концепции константности в языке нет, неизменяемость значит просто, что у объекта типа Integer нет методов, меняющих тот int, который он хранит. По-видимому, причина появления таких чудо-классов — безопасность. Понятно, что легко написать изменяемый аналог Integer, но этим мы займёмся чуть позже.

Измерения показывают, что производительность получившейся программки — примерно та же, что у Перловского аналога. Сравнение более чем честное, ведь регулярные выражения Перла написаны на C, а Явы — на Яве[3].

Ява отличается от, скажем, Перла изобилием инструментальных средств. Профилировщики позволяют "в одно касание" выяснить, кто же держит. Профилировщик покажет часть очевидную и не совсем очевидную. Во-первых, наиболее длительная часть основного цикла ($\approx 50\%$) — разбор текущей строки лога на `host` и `url`. Во-вторых, профилировщик показывает, что до трети общего времени тратится сборщиком мусора[4]. Размышляя о производительности современных языков, необходимо осознавать не только, сколько занимает исполнение данной строчки, но и сколько будет стоить сбор мусора, который здесь, возможно, сгенерируется.

Вообще же, современная Ява-машина — это ещё и хитроумный оптимизирующий компилятор. Не стоит недооценивать её хитроумие: автор с удовольствием прочитал на с. 572 книжки М.Моргана **Java 2. Руководство разработчика**[5] полезный и небанальный совет "выносить повторяющиеся участки вычислений из циклов для повышения производительности", украшенный примером, где выносят `myString.length()`. Реальность же такова, что Ява-машина прекрасно знает о неизменяемости строк, и потому никакого ускорения в случае `String.length()` не будет[6].

Наконец, многообещающе выглядит окрестность `Collections.sort(urls, new Comparator()`, но это мы рассмотрим чуть ниже.

# 3 Литературный обзор

Методы, которым Ява проталкивалась в сознание программистских масс, не могли не сказаться на среднем уровне текстов о ней. Впрочем, некая позитивная динамика есть и тут. Если раньше критерием минимальной вменяемости было отсутствие фраз "В Яве нет указателей", то сейчас совсем странных книжек поубавилось.

# Список литературы

[1] Bruce Eckel **Thinking in Java** (2nd Edition) 1128 p., Prentice Hall PTR; (June 5, 2000) Оригинал есть в электронном виде. Существует русская книжка изда-

[3] Напомним по ходу дела, что единственное место, где Перл гордится своей скорость — как раз регулярные выражения.

[4] Понятно, что пример — совершенно модельный, и ничего не стоит *в данном случае* дать Яве-машине столько памяти, что сборка мусора вообще станет незначимой. Однако в реальной жизни сборка мусора — не артефакт.

[5] Да, издательства SAMS!

[6] Будет небольшое и загадочное замедление, но 5% можно и проигнорировать.

тельства Питер под названием **Философия Java**, попадался также перевод в электронном виде совершенно безумного качества (не тот, что в книжке). Мне нравится подход Эккеля: вместо рассуждений для общего случая и рисования (условно говоря) форм Бэкуса-Наура нам приводят набор содержательных примеров, которые затем поясняют. Платой за такой подход будет, естественно, не-глубина, но, в конце концов, необходимость читать документацию никто не отменял.

[2] Ed Roman, Scott W. Ambler, Tyler Jewell, Floyd Marinescu **Mastering Enterprise JavaBeans** (2nd Edition) 672 p., John Wiley & Sons, (December 2001) Довольно водянистый текст, автор кое-чего не понимает вообще, но вместе со следующей книжкой позволяет "приобщиться" к J2EE. Доступна в pdf.

[3] Stephanie Bodoff (Editor), Dale Green, Kim Haase, Eric Jendrock, Monica Pawlan, Beth Stearns **The J2EE(TM) Tutorial** 528 p., Addison-Wesley Pub Co (March 2002) Сановский учебник. Доступна в pdf.

[4] Floyd Marinescu, Ed Roman **EJB Design Patterns: Advanced Patterns, Processes, and Idioms** 288 p., John Wiley & Sons, (February 19, 2002). Начавшая самостоятельную жизнь глава из книги Романа [2]. Доступна в pdf.

[5] Bruce A. Tate **Bitter Java** 350 p., Manning Publications Company (April 2002) Интересна рассказами о типичных ошибках Ява-программистов, конструктивная часть скорее хромает, но местами любопытно. Позволяет узнать много новых (английских) слов. Доступна в pdf.

[6] Marc Fleury, Scott Stark, The JBoss Group **JBoss Administration and Development** 528 p., Sams, (March 20, 2002). Лучше было бы назвать JBoss Iternals. Доступна в pdf.

# 4 Языковые возможности

## 4.1 Наследование, интерфейсы

Наследование есть, наследование одиночное, если объявить функцию без реализации, которую обязаны реализовать в порождённом классе, используется ключевое слово `abstract`, и т.д. Это всё неинтересно...

Вернёмся к сортировке. Каждому ясно, что любой современный язык программирования содержит базовые алгоритмы (наподобие сортировки и бинарного поиска) в готовом виде, в Яве эти методы хранятся в `java.util. Collections`. Интересные проблемы возникают, когда нужно передать этим алгоритмам параметры, обеспечить, чтобы для классов существовали требуемые операции[7] и пр.

Стандартный способ достижения этой цели в Яве — *интерфейсы*. Интерфейс представляет собой набор объявлений функций[8], с ним можно делать 2 вещи: расширять (`extends`), унаследовав от него другой интерфейс, и реализовывать (`implements`) в некотором классе. К примеру, один из видов контейнеров — (красно-чёрное) дерево. Добавляемое в него, ясное дело, должно иметь операцию сравнения. Это устроено так: задан интерфейс,

```
package java.lang;
public interface Comparable {
   public int compareTo(Object o);
}
```

и всё, претендующее на умение добавляться, должно этот интерфейс реализовывать:

[7] Например, разумная работа с деревом предполагает возможность сравнивать ключи. Хорошо функции сортировки — ей ключи как параметр передаются, в случае с добавлением в дерево это было бы грусно...

[8] Про константы я, в общем-то, знаю.

```
class MapData implements Comparable {
    private String s;
    private int i;
    MapData(String s, int ii) {
        this.s = s;
        i = ii;
    }
       # not reqs by interface but sort implementation
    public boolean eqvals(Object o) {
        if (s.equals(o) && i == ((MapData)o).i)
            return true;
        else
            return false;
    }
    public int compareTo(Object o) {
        int strRes = s.compareTo(((MapData)o).s);
        if (strRes > 0)
            return 1;
        else if (strRes < 0)
            return -1;
        else
            if (i < ((MapData)o).i)
                return -1;
            else if (i > ((MapData)o).i)
                return 1;
            else
                return 0;
    }
}
```

Возможно реализовывать несколько интерфейсов. Довольно часто интерфейс используют как маркер, т.е. никаких методов он не реализует, значимым является само наличие интерфейса. Возможна ситуация, когда класс реализует интрефейс, но не все методы этого интерфейса. В этом случае создавать объекты такого класса нельзя, но можно у него наследовать.

Простейший способ проверить, можно ли привести объект к данному типу[9] — оператор `instanceof`. Например, в случае MapData после

```
MapData md = new MapData("foo", 13);
boolean der = md instanceof Comparable;
```

der окажется истиной. Более подробно о метаинформации — в специальном разделе 4.4.2.

В общепринятой методике проектирования, интерфейс представляет собой контракт, который класс, интерфейс реализующий, обязуется выполнять. В такой интерпретации не очень даже важно, достигаются ли цели через поведение функций, которые реализатора интерфейса заставил написать транслятор, или иными способами (см., например, разговоры про `java.lang.Cloneable` и `java.io.Serializable` ниже). Как механизм проектирования идея интерфейсов-контрактов оказывается очень к месту даже на plain C, но, конечно, для всех лучше, если она поддержана языком.

## 4.2 clone и Cloneable

Практика показывает, что представления о соотношении объектов и указателей подчас неглубоко проникают в сознание начинающих и случайных программистов. Поэтому остановимся на этом более подробно.

[9] Т.е. принадлежит ли предъявленный объект данному классу, производным от данного класса, реализует данный интерфейс, или производный от данного интерфейса.

Как мы уже договорились, присваивание объектов — это на самом деле присваивание указателей на них. Но что делать, если действительно нужно создать копию объекта? Идея здесь такая: если объект хочет, чтобы его клонировали, он предоставляет метод `Object clone()`. Здесь возникает дилемма: с одной стороны, объект не должен клонироваться без его согласия, с другой, клонирование нужно довольно часто, и совсем вручную им заниматься было бы грустно. Кстати говоря, здесь проявляется характерная двойственность: Ява одновременно и язык, и ОС, потому, помимо удобства разработки, проектировщики стремились к непробиваемой защите[10].

Решение такое: как известно, все классы Явы наследуют от класса `Object`, так вот в классе `Object` метод, реализующий клонирование, есть, но он — защищённый (`protected`). Это позволяет без особых усилий обеспечить клонирование: нужно просто сделать `Object.clone()` публично-доступным. Ещё один искусственный приём, возникающий здесь — требование к классу реализовать интерфейс `Cloneable`. В отличии от public метода clone, это требование проверяется в run-time, и если оно не выполнено, выбрасывается исключение CloneNotSupportedException. Этот интерфейс — лишь маркер, никаких методов в нём нет. Для успешного клонирования он должен присутствовать где-то в иерархии клонируемого класса, приводимый ниже пример сохранит работоспособность и если `implements Cloneable` переместится к BaseData.

// CloneTest.java

```
package ru.test1;

class IncludedData {
    int aaa;
    IncludedData(int aaa) {
        this.aaa = aaa;
    }
}
class Data implements Cloneable {
    private IncludedData foo;
    int i;

    public Data(int a, int ii) {
        foo = new IncludedData(a);
        i = ii;
    }
    public void incrData() {
        i++;
    }
    public Object clone() throws CloneNotSupportedException {
        return super.clone(); // just call Object.clone()
    }
}
public class CloneTest {
    public static void main(String[] args)
                        throws CloneNotSupportedException {
        Data d1, d2, d = new Data(1, 11);
        d1 = d;
        d.incrData();
                 // now d1 keeps 12 also
        d2 = (Data)d.clone();
        d.incrData();
                // d2 keeps 12, but d1 and d -- 13
    }
```

[10] Полезно, наверное, подчеркнуть важное различие с целями Стауструпа. В концентрированном виде различие содержится в `#define private public`.

```
}
//
```

Стандартный clone реализует семантику побитового копирования, так что foo окажется общим для d и d2. То есть, копировние получилось не глубокое, а на один уровень вложенности. Как достичь глубокого копирования — понятно: после super.clone() надо вызвать clone на данные объекта в неэстетичном стиле `((Data)o).d = (IncludedData)d.clone();`.

Может возникнут вопрос — а зачем всё это надо, если можно вместо `return super.clone();` написать просто `new Data(...)`. Чтобы ситуация стала ясна, рекомендуется вспомнить о данных родительских классов.

## 4.3 Приложения базовых техник ООП

### 4.3.1 Пример: навигация по текстам на Swing-е

Для начала рассмотрим небольшой пример. Идея здесь — в создании средства быстрой навигации по исходным текстам на основе внешнего вида этих исходных текстов.

Для рисования кнопок используется Swing — pure Java библиотека GUI. Для нас сейчас Swing не представляет интереса, неплохое введение находится на `http://java.sun.com/docs/books/tutorial/uiswing/index.html`, полезны также входящие в поставку JDK примеры SwingSet2.

// Navigate.java

```
package ru.test;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.border.Border;

class Pair_JFrame {
    JFrame fileFrame, navFrame;
    Pair_JFrame(String fileName) {
        this.fileFrame = new JFrame(fileName);
        this.navFrame = new JFrame(fileName);
    }
}

class MainWinData {
    String cwd;
    MainWinData(String cwd) {
        this.cwd = cwd;
    }
}

public class Navigate {
    private static JFrame CreateNavigate(
        final String filePath,
        final String fileName)
        throws FileNotFoundException, IOException {
        String currStr, totalStr = "";
        int maxWidth = -1, lineNum = 1;
        final ArrayList strLength = new ArrayList();
        final TreeMap y2pos = new TreeMap();
```

```
final Pair_JFrame pair = new Pair_JFrame(fileName);

pair.fileFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

BufferedReader in =
    new BufferedReader(new FileReader(filePath + '/' + fileName));
y2pos.put(new Integer(0), new Integer(0));
while (null != (currStr = in.readLine())) {
    totalStr += currStr + "\n";
    strLength.add(new Integer(currStr.length()));
    if (currStr.length() > maxWidth)
        maxWidth = currStr.length();
    y2pos.put(new Integer(lineNum++), new Integer(totalStr.length()));
}
in.close();
final int fMaxWidth = maxWidth;
final JTextArea textAr = new JTextArea(totalStr);

Container contMainFrame = pair.fileFrame.getContentPane();
JScrollPane spTextAr = new JScrollPane(textAr);
Dimension currDim =  spTextAr.getToolkit().getScreenSize();
currDim.setSize(currDim.width/3, currDim.height/3);
spTextAr.setPreferredSize(currDim);
pair.fileFrame.getContentPane().setLayout(
    new BoxLayout(pair.fileFrame.getContentPane(), BoxLayout.Y_AXIS));
contMainFrame.add(spTextAr);
pair.fileFrame.pack();
pair.fileFrame.setVisible(true);

// canvas ---------------------------------------------------------
Canvas lineCanv = new Canvas() {
    public void paint(Graphics g) {
        int lineNum = 0;
        g.setColor(Color.red);
        for (Iterator e = strLength.iterator();
            e.hasNext();
            lineNum++) {
            int strL = ((Integer) e.next()).intValue();
            g.drawLine(0, lineNum, strL, lineNum);
        }
    }
};
lineCanv.setSize(maxWidth, strLength.size());
lineCanv.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent ev) {
        int y = (int) ev.getPoint().getY();
        if (y >= y2pos.size())
            y = y2pos.size() - 1;
        int pos = ((Integer) y2pos.get(new Integer(y))).intValue();
        textAr.setCaretPosition(pos);
    }
});
pair.navFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

pair.fileFrame.addWindowListener(new WindowAdapter() {
    public void windowClosed(WindowEvent e) {
        pair.fileFrame = null;
        if (null != pair.navFrame) {
            pair.navFrame.dispose();
            pair.navFrame = null;
```

```
            }
        }
    });
    pair.navFrame.addWindowListener(new WindowAdapter() {
        public void windowClosed(WindowEvent e) {
            pair.navFrame = null;
            if (null != pair.fileFrame) {
                pair.fileFrame.dispose();
                pair.fileFrame = null;
            }

        }
    });

    JScrollPane spLineCanv = new JScrollPane(lineCanv);
    pair.navFrame.getContentPane().add(spLineCanv);
    pair.navFrame.pack();
    pair.navFrame.setVisible(true);
    return pair.navFrame;
}
private static String[] createFileList(final String cwd) {
    String[] fileList = (new File(cwd)).list();
    Arrays.sort(fileList);
    String res[] = new String[fileList.length + 1];
    res[0] = "..";
    for (int i = 0; i < fileList.length; i++)
        res[i + 1] = fileList[i];
    return res;
}
public static void main(String[] args) throws FileNotFoundException {
    final MainWinData d = new MainWinData("/home/u1305/exper/pvfs/mgr");
    final LinkedList createdWins = new LinkedList();

    final JList fileList = new JList(createFileList(d.cwd));
    final JFrame fileFrame = new JFrame(d.cwd);
    JButton closeAllButton = new JButton("close all");
    Container contFileFrame = fileFrame.getContentPane();
    JScrollPane spFileList = new JScrollPane(fileList);

    closeAllButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            for (Iterator it = createdWins.iterator(); it.hasNext();)
                 ((JFrame) it.next()).dispose();
            createdWins.clear();
        }
    });

    contFileFrame.setLayout(new BoxLayout(contFileFrame, BoxLayout.Y_AXIS));

    fileList.addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent ev) {
            // jdk-1.4 has getButtons, but...
            if (! ((ev.getModifiers() & InputEvent.BUTTON1_MASK)
                   == InputEvent.BUTTON1_MASK))
                return;
            fileList.setSelectedIndex(
                fileList.locationToIndex(ev.getPoint()));
            String name = (String) fileList.getSelectedValue();
            if ((new File(d.cwd + '/' + name)).isDirectory()) {
                if ("..".equals(name))
```

```
                    d.cwd = d.cwd.substring(0, d.cwd.lastIndexOf('/'));
                else
                    d.cwd += '/' + name;
                fileList.setListData(createFileList(d.cwd));
                fileFrame.setTitle(d.cwd);
            } else if ((new File(d.cwd + '/' + name)).isFile())
                // it's ordinal file, not directory
                try {
                    createdWins.add(CreateNavigate(d.cwd, name));
                } catch (FileNotFoundException e) {
                    e.printStackTrace();
                    // rethrow there!!!
                } catch (IOException e) {
                    e.printStackTrace();
                    // rethrow there!!!
                }
            }
        });

        closeAllButton.setAlignmentX(Component.CENTER_ALIGNMENT);

        contFileFrame.add(Box.createRigidArea(new Dimension(0, 10)));
        contFileFrame.add(spFileList);
        contFileFrame.add(Box.createRigidArea(new Dimension(0, 10)));
        contFileFrame.add(closeAllButton);
        contFileFrame.add(Box.createRigidArea(new Dimension(0, 10)));
        fileFrame.pack();

        fileFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fileFrame.setVisible(true);
    }
}
//
```

Для начала — краткий обзор того, что мы видим, и некоторые внесистемные комментарии. Итак, для начала открывается окно fileFrame с содержимым текущего каталога и кнопкой closeAllButton. Если ткнуть мышкой в файл — файл откроется в 2-х окнах: в одном будет его текст, а в другом — панель быстрой навигации. Два порождённых окна, хотя и внешне независимы, связаны по закрытию — если закрыть одно, другое тоже закроется. Кнопка closeAllButton позволяет закрыть все такие окна. Идея этого примера возникла из анализа нужд навигации: обычная линейка прокрутки позволяет мне попасть куда-нибудь в верхнюю треть файла, но про функции она ничего не знает. Классические Class Browser-ы показывают (естественно) имена, но зачастую интересны не имена, а куски кода[11].

Программа вынуждена хранить текущий каталог в явном виде (d.cwd). Это сделано так потому, что в Яве поменять текущий каталог невозможно (интересно, что узнать его можно). Нет проблем вызвать `chdir(2)`, пользуясь JNI[12], но вот, например, IBM VisualAge for Java 3.x от chdir-а самоуничтожается.

### 4.3.2 Вложенные классы

Ещё группа языковых возможностей, востребованных при написании сортировки и во многих других важных областях, включая GUI, сетевизмы и пр. — замыкания, анонимные функции и близкие идеи.

[11] Деятели, рассуждающие о том, что "значит, не всё в порядке с инкапсуляцией" и что "большие модули создавать не нужно", могут отправляться переписывать в предлагаемом ими стиле чужие программы. После успешного возвращения дискуссия о большом и маленьком может быть продолжена.

[12] Java Native Interface, API для вызывания из Явы функций на C

Рассмотрим создание кнопок. Самая интересная вещь в этом, в общем, понятном процессе — обработчик нажатия. Здесь от языка требуются две веши:

- Удобство в написании обработчиков. Вполне частая ситуация — необходимость небольшого и специфичного обработчика для каждой кнопки. Это значит, что лучше бы обработчику не быть методом (функцией), ведь от изобилия маленьких функций страдает читаемость. Для решения проблемы применяются **анонимные функции**. Пример: `closeAllButton.addActionListener(new ActionListener(){` и далее по тексту.

- Удобства в передаче им параметров. Хитрость здесь в том, что обработчик создаётся в одном контексте, а работает в другом, потому параметры бывают 2-х сортов: параметры создания и параметры вызова обработчика. Не следует смешивать параметры создания и глобальные данные: глобальные одинаковы для всех, а вот параметры создания могут быть для каждого свои, на то они и параметры. К примеру, разные lineCanv видят каждый свою textAr, иначе бы всё сломалось. Языковый механизм, обеспечивающий это, называется **замыканием**. Идея в том, что замыкание запоминает локальный контекст, в котором создаётся[13].

Для реализации обеих возможностей разработчки Явы предлагают использовать один и тот же механизм: вложенные классы (inner classes). С точки зрения языка в районе `closeAllButton.addActionListener(new ActionListener() {` происходит вот что. Создаётся экземпляр безымянного класса, реализующего интерфейс `java.awt.event.ActionListener`. Поскольку этот интерфейс содержит лишь метод `public void actionPerformed(ActionEvent e)`, его мы тут и реализуем. Ничто не мешало написать

```
public static void main(String[] args) throws FileNotFoundException {
...
Container contFileFrame = fileFrame.getContentPane();
JScrollPane spFileList = new JScrollPane(fileList);

class CloseAllActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        for (Iterator it = createdWins.iterator(); it.hasNext();)
    ((JFrame) it.next()).dispose();
            createdWins.clear();
        }
}
closeAllButton.addActionListener(new CloseAllActionListener());
...
```

, кроме несомненного снижения читаемости (`CloseAllActionListener` должен находиться в теле main, иначе он `createdWins` не увидит).

Можно оформить всё по-другому, без замыканий: хранить `createdWins` в экземпляре `CloseAllActionListener`, инициализируя в конструкторе, и пр. Тут 2 высказывания:

- Это выглядит отталкивающе.

- Насколько я понимаю, на самом деле (т.е. на уровне JVM) всё так и происходит, замыкания реализованы без специальной поддержки в run-time[14].

Пытливый слушатель уже заметил, что все данные, к которым обращаются из замыканий, по настоятельному совету транслятора снабжены атрибутом `final`. Чтобы осознать, почему это так, можно вспомнить о том, что мы видим не объекты, а указатели на них.

[13] В C возвращение вниз по стеку указателя на содержимое стека — трудноуловимая ошибка, но здесь не C.

[14] Упражнение: поразмышлять, причём тут сборка мусора.

Автору не раз доводилось разглядывать дискуссии про `final` в стиле: “А вот как `final` влияет на производительность?”, “Раньше влиял, а вот на HotSpot JVM...”, “Существуют трансляторы, для которых ..., но вот массовые к ним не относятся”. Интересно, что в “Design and Evolution of C++” Страуструп пишет что-то вроде “Некоторые пользователи надеялись, что пересмотр концепции `const` и введение `mutable` открывает дорогу серьёзным оптимизациям кода. Вряд ли это так. Основное достоинство — повышение ясности кода...”[15] “Design and Evolution“ опубликована в 94-м, и иногда возникает впечатление, что архитекторы Явы нарочно выбирали решения, изъяны которых подробно разбирает Страуструп[16].

Наконец, ещё сознательная девиация примера — игнорирования “традиционной” объектной системы и использование для инкапсуляции замыканий. Не то, чтобы этот путь был особенно замечательным, но представлять, что так делать тоже можно — по-моему, полезно.

### 4.3.3 Listener-ы. Паттерн Publish-Subscribe. Использование при реализации GUI и распределённых систем.

Если чуть-чуть задуматься о схеме уничтожения окон, станет понятно, что она никуда не годиться. При любви к навешиванию ярлыков, ярлык здесь однозначен: “жёсткая связанность”. И впрямь, обработчик закрытия каждого окна должен знать все окна, которые которые ему предстоит закрыть. Легко себе представить, во что это выльется, если окон станет побольше. Решение здесь настолько хорошо известно, что даже названий у него много — это паттерн Publish-Subscribe[17]. Говоря простыми словами, вместо унылого явного вызова нужно написать 2 вещи:

1. регистратора, куда будут звонить все заинтересованные в событиях и говорить: “Если ЭТО случится, сообщите нам”.

2. оповещателя, который, когда ЭТО случается, обзванивает всех.

Видно, что, кроме возможности просто узнать, кто заинтересован в событиях и отсутствия жёсткой заданности заинтересованных, мы получили ещё важную возможность — возможность удалять и добавлять слушателей событий на лету, в динамике. Ява содержит основу для (совсем простой) реализации паттерна в виде класса java.util.Observable (то, за чем наблюдаем) и интерфейса java.util.Observer (интерфейс наблюдателя). Пользоваться этим можно примерно так:

```
import java.util.Observable;
import java.util.Observer;
import javax.swing.JFrame;

class CloseObserver implements Observer {
    public void update(Observable o, Object arg) {
        System.out.println("recv: "+arg);
    }
}
class OWindow extends Observable {
    void winClosed() {
        setChanged();
        notifyObservers("bbb");
    }
}
public class DemoOb {
    public static void main(String[] args) {
        OWindow o = new OWindow();
        CloseObserver obs1 = new CloseObserver(),
```

[15] D&E, 13.3.3

[16] “Денег у нас хватает. У нас ума не хватает.”

[17] Он же Observer-Observable

```
                    obs2 = new CloseObserver();
        o.addObserver(obs1);
        o.addObserver(obs2);
        o.winClosed();
    }
}
```

Комментария заслуживает разве лишь `setChanged()`. Идея тут в том, что пока наблюдаемый не менялся — попытки оповестить наблюдателей игнорируются. Вызов `notifyObservers` сбрасывает флаг МЕНЯЛИСЬ в НЕТ, т.е после каждого `setChanged() notifyObservers` кого бы то нибыло оповестит лишь однажды. Все вызовы — синхронные, так что пока один наблюдатель не вернётся из своего update, следующий управление не получит.

Ясно:

1. Для оповещателей о закрытии эта схема избыточна.
2. Предлагаемые Sun `java.util.Observable` и `import java.util.Observer` крайне незамысловаты, и возникает мысль "А зачем это надо, я и сам такое за 3 минуты напишу". Оказывается, эта мысль посетила и кое-кого внутри Sun, так что Swing пользуется своей реализацией паттерна Publisher[18], игнорируя вышерассмотренное. Никаких проблем!

Что тут не столь очевидно и весьма важно: перейдя к подписке, мы снизили связанность классов, и у этого кроме плюсов есть и минусы: доля динамики возросла, потому статически (читая текст) разобраться, что происходит, стало труднее. Это довольно важный пункт, дальнейшие примеры — в 6.3.1.

### 4.3.4 Weak Reference

*Подраздел предназначен для дополнительного чтения.*

Типичная проблема, возникающая при использовании паттерна Publisher — утечки памяти. Проблема возникает вот где: всё замечательно, пока объект договаривается слушать оповещения о разных событиях, но что делать, когда этот объект более не нужен? Сборщик мусора его не освободит, пока на него есть ссылки, а ссылки мы щедро раздавали всем, к кому записывались в слушатели. Конечно, можно им позвонить и попросить, чтобы про нас забыли[19], но для этого, по крайней мере, нужно знать всех, к кому мы записались. В результате возрастает связанность, от которой мы уходили, внедряя Observer-ов (к примеру, память из под Observable не освободится, пока на него есть ссылки, а ссылок у нас — в каждом Наблюдателе). Видно, что проблема, в общем, решаемая, но на пути есть масса мелких хитростей.

Альтернативой ручному манипулированию может служить использование слабых (**weak**) ссылок, см. `java.lang.ref.WeakReference`. Слабость ссылок состоит в том, что если на объект нет иных ссылок, кроме слабых, то мусоросборщик может объект собрать. Идея их использования в следующем. Вместо того, чтобы (скажем) хранить в Observer-е `JFrame`, мы поместим туда `new WeakReference(myJFrame)`, и теперь никаким явным удалением можно не озадачиваться. Естественно, что если от JFrame не останется никакого следа, кроме слабых ссылок в Наблюдателях, то ничего работать не будет, но кропотливой работы всё равно стало меньше. Получить из WeakReference то, что мы туда положили можно, вызвав `get`: она вернёт Object (к настоящему типу нужно приводить ручками, но так уж в Яве ведётся), если же процесс освобождения произошёл, то `get` вернёт `null`. Необходимо понимать, что слабая ссылка хранится внутри экземпляра класса WeakReference, ссылка же на этот экземпляр самая обыкновенная, а никакая не слабая.

[18] См. `java.util.EventListener` и далее по ссылкам об использовании. См. также 6.3.1

[19] Для java.util.Observable это делается вызовом deleteObserver(Observer o).

## 4.4 Работа с метаинформацией

### 4.4.1 Пример: JDBC

Всякая разумная современная система программирования должна предоставлять унифицированный интерфейс с РСУБД. "Унифицированный" здесь означает "не меняющийся при смене РСУБД". Все знают, что (нетривиальный) SQL при смене РСУБД приходится кроить, ну, а вот интерфейс меняться не должен. Низкоуровневое API, решающее в Яве эту задачу, называется JDBC. Рассмотрим базовый пример. К качестве СУБД используется PostgreSQL 7.1.3.

Делается здесь вот что. Имеем таблицу `user`, хранящую логины и пароли пользователей, а также дополнительную информацию о пользователе. Таблица `messages` хранит сообщения пользователей и время, когда было получено сообщение. `serial` используется для автоматической выдачи нового ключа при добавлении записи.

```
drop table messages;
drop table users;
drop sequence messages_mid_seq;
create table users (login varchar(30) primary key,
    password varchar(20), userdata varchar(60));
create table messages (login varchar(30), date timestamp,
    mid serial PRIMARY KEY, message text,
    foreign key (login) references users);
```

Примитивная прослойка, обеспечивающая работу с базой в терминах предметной области, может выглядеть так:

// JDBCDemo.java

```
package ru.test;

import java.sql.*;
import java.util.ArrayList;
import java.util.Iterator;

class InternalAccessError extends RuntimeException {
    private final Exception e;
    public InternalAccessError(final Exception e, String descr) {
        super(descr);
        this.e = e;
    }
    public String toString() {
        return e.getMessage();
    }
}
class NoSuchUser extends Exception {
    private String login;
    public NoSuchUser(String login) {
        this.login = login;
    }
}
class MessageStruct {
    MessageStruct(String login, String message, Timestamp date) {
        this.login = login;
        this.message = message;
        this.date = date;
    }
    public String toString() {
        return login + ": '" + message + "' at " + date;
    }
    String login, message;
    Timestamp date;
```

```
}
class UserDataStruct {
    String login, userData;
    UserDataStruct(String login, String userData) {
        this.login = login;
        this.userData = userData;
    }
    public String toString() {
        return "user " + login + " info '" + userData + "'";
    }
}
public class JDBCDemo {
    private Connection conn;

    public JDBCDemo(
        String dbName,
        String dbUser,
        String dbPassword) {
        try {
            Class.forName("org.postgresql.Driver");
        } catch (ClassNotFoundException e) {
            throw new InternalAccessError(e, "driver loading");
        }
        try {
            conn =
                DriverManager.getConnection(
                    "jdbc:postgresql://localhost/" + dbName,
                    dbUser,
                    dbPassword);
        } catch (SQLException e) {
            throw new InternalAccessError(e, "DB connecting");
        }
    }
    private void closeStatement(Statement stmt) {
        try {
            if (null != stmt)
                stmt.close();
        } catch (SQLException e) {
            throw new InternalAccessError(e, "close statement");
        }
    }
    public void addUser(
        String login,
        String password,
        String userdata) {
        Statement stmt = null;
        try {
            stmt = conn.createStatement();
            stmt.executeUpdate(
                "insert into users(login, password, userdata) values ('"
                    + login
                    + "', '"
                    + password
                    + "', '"
                    + userdata
                    + "')");
        } catch (SQLException e) {
            throw new InternalAccessError(e, "access to users");
        } finally {
            closeStatement(stmt);
```

```
    }
}
public void delUser(String login) throws NoSuchUser {
    Statement stmt = null;
    try {
        stmt = conn.createStatement();
        // referencial integrity reqs it
        stmt.executeUpdate(
            "delete from messages where login='" + login + "'");
        int res =
            stmt.executeUpdate(
                "delete from users where login='" + login + "'");
        if (0 == res)
            throw new NoSuchUser(login);
    } catch (SQLException e) {
        throw new InternalAccessError(e, "access to users");
    } finally {
        closeStatement(stmt);
    }
}
public void addMessage(String login, String message) {
    Statement stmt = null;
    try {
        stmt = conn.createStatement();
        stmt.executeUpdate(
            "insert into messages(login, date, message) values ('"
                + login
                + "', '"
                + new Timestamp(System.currentTimeMillis())
                + "', '"
                + message
                + "')");
    } catch (SQLException e) {
        throw new InternalAccessError(e, "access to users");
    } finally {
        closeStatement(stmt);
    }
}
public ArrayList getLastMessages(int num_of_messages) {
    ArrayList messList = new ArrayList(num_of_messages);
    Statement stmt = null;
    try {
        stmt = conn.createStatement();
        ResultSet rs =
            stmt.executeQuery(
                "select * from messages order by date desc limit "
                    + num_of_messages
                    + ";");
        while (rs.next())
            messList.add(
                new MessageStruct(
                    rs.getString("login"),
                    rs.getString("message"),
                    rs.getTimestamp("date")));
    } catch (SQLException e) {
        throw new InternalAccessError(e, "access to users");
    } finally {
        closeStatement(stmt);
    }
    return messList;
```

```
    }
    public ArrayList getAllUserInfo() {
        ArrayList userDataList = new ArrayList();
        Statement stmt = null;
        try {
            stmt = conn.createStatement();
            ResultSet rs =
                stmt.executeQuery(
                    "select login, userdata from users order by login;");
            while (rs.next())
                userDataList.add(
                    new UserDataStruct(
                        rs.getString("login"),
                        rs.getString("userdata")));
        } catch (SQLException e) {
            throw new InternalAccessError(e, "access to users");
        } finally {
            closeStatement(stmt);
        }
        return userDataList;
    }
    public static void main(String[] args) {
        JDBCDemo dbComm = new JDBCDemo("test", "u1305", "");

        try {
            dbComm.delUser("u1305");
        } catch (NoSuchUser e) {}
        try {
            dbComm.delUser("root");
        } catch (NoSuchUser e) {}

        dbComm.addUser("u1305", "q", "me");
        dbComm.addUser("root", "q1", "me 2");
        for (int i = 0; i < 100000; i++)
            dbComm.addMessage("u1305", "q-q! " + i);
        ArrayList msgs = dbComm.getLastMessages(10);
        for (Iterator e = msgs.iterator(); e.hasNext();) {
            System.out.println((MessageStruct) e.next());
        }
        System.out.println(dbComm.getAllUserInfo());
    }
}
//
```

Всё, как видим, просто и понятно. Для того, чтобы получить желанный ответ на `select`, нужно выполнить следующие шаги:

1. Загрузить JDBC-драйвер для базы, с которой надеетесь соединиться. (`Class.forName("org.postgresql.Driver");`)
2. Создать соединение с базой под названием messages, распооженной на машине um16 (здесь нужен просто IP-адрес) (`Connection conn = DriverManager.getConnection("jdbc:postgresql://um16/messages", dbUser, dbPassword);` )
3. Создать Statement.
4. Выполнить запрос. (`ResultSet rs = statement.executeQuery(“select * from users”)`)
5. Выбрать из ResultSet-а понравившиеся ответы.

6. Отдать ненужные ресурсы (т.е., ResultSet-ы, Statement-ы и коннекты).

От СУБД здесь зависят только п.п.1–2. Может возникнуть вопрос: а зачем нужны ResultSet-ы, если в языке уже есть контейнеры. Ну, помещали бы в какой-нибудь LinkedList результаты select-а, и не умножали сущностей. Тут есть 2 слоя ответов. На начальном уровне всё просто: результат запроса может оказаться размером в гигабайт, весь его загружать из СУБД может и не захотеться, потому обычные контейнеры и не пригодны. На более высоком уровне совершенно непонятно, почему из ResultSet-а нельзя получить Iterator.

Обсудим теперь вопрос о параллельности. Ясно, что в реальной жизни нужно уметь исполнять несколько запросов к СУБД одновременно. Если просто попытаться совместно использовать экземпляр JDBCDemo в разных нитях, то всё сломается. Формально ничто не мешает приспособить вышеприведённый код к такой ситуации: нужно просто создавать в каждом параллельном исполнителе свой экземпляр JDBCDemo, и всё. Но на самом деле, обычно так не делают по следующим причинам:

- Коннект с СУБД — доростоящий ресурс: его установление отнимает значительное время.
- Коннект с СУБД — доростоящий ресурс: одновременное существование тысячи коннектов не будет способствовать счастью СУБД.
- Коннект с СУБД — доростоящий ресурс: распостранённая политика лицензирования состоит в том, что деньги берутся как раз за максимальное число одновременных коннектов.

Стандартный ход в этой ситуации — кеширование ресурса. В применении к СУБД это называется **пул коннектов**[20]. Основная идея состоит во введении промежуточного хранилища (собственно пула), откуда достаются коннекты и куда они помещаются по прекращении потребности в них. Как и любой кеш, пул не спешит отдавать ресурсы (т.е., закрывать коннекты) в надежде, что коннект вскоре снова потребуется. В тоже время, когда-нибудь ресурс отдавать всё равно будет нужно, и изредко случается, что логика приложения конфликтует с логикой освобождения ресурсов. Пытливый слушатель может поймать нас здесь на передёргивании — каким это волшебным образом введение пула поможет одновременно исполнять тысячу запросов? Ответ, в общем, понятен: никаким не поможет, потому что это уже для самой СУБД многовато. Ясно, однако, что ситуация “не более 20-ти одновременно” вообще говоря гораздо более приемлема для приложений, чем “строго по одному”.

Конечно же, существуют готовые реализации пулов, как в составе больших продуктов (напр., Tomcat), так и отдельно-стоящие (напр., PoolMan), несложно, впрочем, и свой написать, если у вас есть какая-то самобытная идея, готовыми реализациями не покрываемая.

Буквально несколько слов о производительности. Видимо, первое, чему обучаются Юные Программисты при дискуссиях о ней — произнесению заклинания “Всё зависит от задачи”[21]. Чтобы проектировать, однако, нужны хоть какие-то, хоть совсем оценочные оценки. Ощущение здесь такое: 100 insert-ов в сек. — это мало и это “все умеют”, 10 тыс. — скорее много, и могут возникнуть проблемы. Авторы Рамблера гордятся 22 тыс., но там у них совсем не РСУБД общего назначения. Конечно, можно таких триггиров навесить, что и 10 в сек. потребуют экзотического железа, но никто ведь и не сомневается, что всё от задачи зависит. Кстати, приведённый выше пример даёт about 600 insert-ов в сек (ну, он совсем игрушечный).

Если посмотреть на код с эстетической точки зрения, особенно представив, какие методы добавятся в реальной жизни, можно прийти к выводу, что он достаточно

[20] Все и без меня хорошо знают, что бывают ещё пулы нитей, сокетов и пр. Особенного расцвета достигла ситуация с памятью, когда этих пулов получается несколько вложенных: аллокатор класса – системный malloc – дисковый буфер ОС

[21] Это, естественно, правильно, но информации не несёт ровно никакой. Довольно забавно бывает, когда человек вообще ничего о производительности сказать не может, кроме этого заклинания.

простой и достаточно уродливый. А именно, повторное использование напоминает название книжки «Вблизи Абсолютного Нуля».

### 4.4.2 Универсальные исполнители. Reflection API

Если почитать, например, обработчиков select-ов, то станет понятно, что они устроены весьма однообразно и различаются лишь типами данных, которые извлекаются из ResultSet-а и помещаются в контейнер. Существующие в Яве средства работы с метаинформацией позволяют написать универсальный код, решающий эту задачу. Схематично говоря, эти средства, называемые также Reflection API, дают возможность узнать, какие реализует ли класс данный интерфейс, узнать все методы класса (с типами), вызвать метод класса, указав имя метода в виде текстовой строки, и пр.

Для повышения понятности неизменившиеся куски из примера выброшены.

// JDBCDemo.java

```
package ru.test1;

import java.lang.reflect.*;
import java.sql.*;
import java.util.*;

class InternalAccessError extends RuntimeException {
    private final Exception e;
    public InternalAccessError(final Exception e, String descr) {
        super(descr);
        this.e = e;
    }
    public String toString() {
        return e.getMessage();
    }
}
class NoSuchUser extends Exception {
    private String login;
    public NoSuchUser(String login) {
        this.login = login;
    }
}
class MessageStruct {
    public MessageStruct() {}
    MessageStruct(String login, String message, Timestamp date) {
        this.login = login;
        this.message = message;
        this.date = date;
    }
    public String toString() {
        return login + ": '" + message + "' at " + date;
    }
    public String login, message;
    public Timestamp date;
}
class UserDataStruct {
    public String password;
    public String login, userData;
    public UserDataStruct() {}
    public UserDataStruct(String login, String userData) {
        this.login = login;
        this.userData = userData;
    }
    public String toString() {
```

```
        return "user "+login+" pass '"+password+"' info '"+userData+"'";
    }
}
public class ReflDemo {
    ...

    public ArrayList getLastMessages(int num_of_messages) {
        return execUniversalQuery(
            "select * from messages order by date desc limit "
                + num_of_messages
                + ";",
            MessageStruct.class);
    }
    public ArrayList getAllUserInfo() {
        return execUniversalQuery(
            "select login, userdata from users order by login;",
            UserDataStruct.class);
    }
    public ArrayList execUniversalQuery(String request, Class resultType) {
        ArrayList resList = new ArrayList();
        Statement stmt = null;
        try {
            stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(request);

            final Field[] resFields = resultType.getFields();
            final Class[] params_of_get = { String.class };

            while (rs.next()) {
                Object newRow =
                    resultType.getConstructor(null).newInstance(null);
                for (int i = 0; i < resFields.length; i++) {
                    String fieldName = resFields[i].getName();
                    String fieldType =
                        resFields[i].getType().getName();
                    if ("int".equals(fieldType))
                        // because getInt, not getint
                        fieldType = "Int";
                    else
                        fieldType =
                            fieldType.substring(
                                fieldType.lastIndexOf('.') + 1);

                    final Object[] args_of_get = { fieldName };
                    resFields[i].set(
                        newRow,
                        ResultSet
                            .class
                            .getDeclaredMethod(
                                "get" + fieldType,
                                params_of_get)
                            .invoke(rs, args_of_get));
                }
                resList.add(newRow);
            }
        } catch (SQLException e) {
            throw new InternalAccessError(e, "access to users");
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
```

```
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        } finally {
            closeStatement(stmt);
        }
        return resList;
    }
 ...
}
//
```

Метод `execUniversalQuery(` и есть этот самый универсальный исполнитель запросов. Идея здесь следующая. Первым делом мы узнаём, какие поля есть в классе, куда нам нужно помещать ответ. Каждая итерация затем устроена так:

- Создаём объект класса-результата, вызывая конструктор без параметров (вы правильно заметили — их раньше не было).
- Для каждого поля результата выясняем имя и тип поля, вызываем соответствующий `getТИП` для `rs` и, наконец, присваиваем текущему полю объекта то, что `getТИП` вернул.
- Добавляем в контейнер с результатами текущую строчку.

Таким образом, чтобы всё работало, от класса, хранящего строчку результата, требуется:

- public конструктор без параметров,
- public данные с типом, совпадающим с извлекаемым из СУБД типом и именем, совпадающим с именем колонки (других public данных быть не должно.

Наконец, 3 замечания.

1. Можно, естественно, соптимизировать и передать конструктору сразу то полезное, что из ResultSet-а достали. К сожалению, можно узнать лишь типы параметров конструктора, но не имена. Имена же нам нужны для того, чтобы знать, какую колонку куда присваивать.
2. От public данных можно отказаться, введя, к примеру, `void setLogin(String)` для доступа к String login. Разобрав все такие set-ы, можно получить требуемую функциональность без нарушения инкапсуляции.
3. Пытливый читатель уже заметил, что везде, где мы пользуемся мета-информацией, возникает public. Если попытаться так же спроста достучаться к не-public данным, доступа не будет. Здесь мы вновь сталкиваемся с объединением в Яве черт языка и ОС. Это ограничение обусловлено защитой и могут быть ослаблены через механизм policy. См. `http://java.sun.com/j2se/1.4/docs/guide/security/permissions.html`.

Как всегда, интересные вопросы возникают, когда мы задумываемся о производительности. С переходом от JDK 1.3 к 1.4 средства Reflection API были радикально ускорены: `invoke` лишь немногим дольше обычного вызова (в 1.3 разница в 2 раза), однако, кроме `invoke`, есть ещё методы наподобие `getDeclaredMethod`, а вот с ними всё не столь радужно — они раза в 2 с лишним медленне обычных (впрочем, в 1.3 было в 5 раз медленнее). Если оказывается, что производительности generic-реализации[22] не хватает, выручить могут такие реализационные решения:

[22] А люди, знающие C++, уже давно догадались, что это generic и есть.

- Кеширование результатов `getDeclaredMethod`.
- Переписывание критичных по времени запросов в специализированную форму и использование generic только для менее критичных по времени[23].
- Совсем другой подход применён в AutoDeployer-е jBoss-а. Об этих замечательных людях — см. 6.3.

Интересно, что Sun в своём учебном Java PetStore реализовала весь доступ к БД унылым способом. Утверждается, что за производительностью они не гнались, так что причины остаются мне непонятны.

Каждый, хочется надеяться, осознал, что generic реализация оказывается очень к месту совсем не только в интерфейсе с БД. К примеру, затосковав от обилия повторяющегося кода, автор однажды написал универсальный искатель в таблицах. Ну, в HTML-таблице нужно было подсветить "интересную" запись среди всех остальных. Хотя, возможно, в этом случае правильная иерархия классов была бы более изящным решением.

[23] Здесь вежливые слушатели иногда спрашивают: "А не призыв ли это заниматься глупостями — сначала мы написали специализированных исполнителей, потом Универсального, а потом опять за специализированных принимаемся?" Ну, отчасти да, но представте 50 исполнителей запросов в стиле первой версии `getLastMessages(...`

# 5 Низкоуровневое удалённое взаимодействия

В Яве, естественно, есть сокеты и пр., но это всё много где ещё есть. Совсем кратко рассмотрим то, что обладает (минимальной) экзотикой.

## 5.1 Коротко и непонятно о нитях

Нити (threads, они же легковесные процессы, они же (ужас) потоки) в Яве есть, эта область ничем от нитей в других системах не отличается. В целях самозамкнутости рассмотрим небольшой пример.

// RemWinUser.java

```
package ru.test;

import java.util.LinkedList;

class SillyTread implements Runnable {
    private static final Object lock_currNum = new Object(),
        emptyQueue = new Object();
    private static int currNum = 0;
    private final int myNum;
    private static final LinkedList queue = new LinkedList();

    public SillyTread() {
        synchronized (lock_currNum) {
            myNum = currNum++;
        }
        Thread t = new Thread(this, "my " + myNum);
        t.setDaemon(true);
        t.start();
    }
    public void run() {
        try {
            while (true) {
                final String elem;
                synchronized (queue) {
                    while (0 == queue.size())
                        queue.wait();
                    elem = (String) queue.removeFirst();
                }
                System.out.println(myNum + ": got " + elem);
                synchronized (queue) {
                    synchronized (emptyQueue) {
                        if (0 == queue.size())
                            emptyQueue.notify();
                    }
                }
                Thread.sleep(1);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public static void addToQueue(final String elem) {
        synchronized (queue) {
            queue.addLast(elem);
            queue.notify();
        }
    }
```

```
    public static void waitEmptyQueue() throws InterruptedException {
        synchronized (emptyQueue) {
            while (0 != queue.size())
                emptyQueue.wait();
        }
    }
}

public class ThrSample {
    public static void main(String[] args) {
        for (int i = 0; i < 50; i++)
            SillyTread.addToQueue("a" + i + "b");

        new SillyTread();
        new SillyTread();
        new SillyTread();
        new SillyTread();
        new SillyTread();
        new SillyTread();
        try {
            SillyTread.waitEmptyQueue();
        } catch (InterruptedException e) {}
    }
}
//
```

Делается тут вот что: существуют нити-исполнители, которые достают строчки из общей очереди и печатают их. Сначала мы добавляем в очередь 50 строчек, а потом создаём 6 нитей.

Хитрый трюк связан с ожиданием завершения. Если напмсать "по-простому", то Ява-машина не закончится, пока не завершится последняя нить либо не будет вызван `System.exit(...)`. Это не совсем удобно, потому что тогда прийдётся уговаривать все нити завершиться либо принудительно прибить опять-таки всех. Для решения проблемы сщестует возможность пометить нить как "демоническую", что значит "её завершения можно не ждать". После того, как это сделано, возникает новая интереснаяа проблема — главная нить должна ждать исчерпания очереди. Для этого вводится монитор `emptyQueue`. Задачка для обдумывания: как обойтись одним монитором `queue` и почему сделано не так.

## 5.2 Сериализация

Нет сомнений, что настоящий полноценный объект должен по специальному указанию уметь превращаться в безжизненный поток байтов, а затем из этого потока вновь возрождаться. По ходу дела безжизненные данные могут быть переданы на другой компьютер и пр., но после возрождения объект должен оказаться "тем же самым". Нет особых проблем сделать это на любом языке программирования, но вот Ява предоставляет встроенные возможности.

Сам процесс называется **сериализацией**, и в чём-то похож на клонирование. Для того, чтобы оказаться сериализуемым, класс должен реализовать интерфейс `java.io.Serializable`, который никаких функций не содержит и служит лишь маркером. Если это условие выполнено для класса и всех его данных, запись-чтение объектов могут выполняться методами `writeObject/readObject` из классов `java.io.ObjectOutputStream/ObjectInputStream`. Рассмотрим пример.

// SerialTest.java

```
package ru.test;
import java.awt.*;
import java.awt.event.*;
```

```
import java.io.*;
import javax.swing.*;
class OuterData implements Serializable {
    final JFrame fr;
    final String headerText;
    final JButton butt;
    /**
     * Create new OuterData
     * @param header text to frame header
     * @param buttonText text to button
     */
    public OuterData(
        final String headerText,
        final String buttonText) {
        this.headerText = headerText;
        fr = new JFrame(headerText);
        butt = new JButton(buttonText);
        final Container cont = fr.getContentPane();
        cont.add(butt);
        fr.pack();
        fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public void makeVisible() {
        butt.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println(headerText);
            }
        });
        fr.setVisible(true);
    }
}
public class SerialTest {
    public static void main(String[] args) throws IOException {
        if (1 != args.length) {
            System.out.println("Usage: SerialTest old|new");
            return;
        }
        if (args[0].equals("new")) {
            OuterData o = new OuterData("aaa", "OK");
            ObjectOutputStream oo =
                new ObjectOutputStream(
                    new FileOutputStream("frame.dat"));
            oo.writeObject(o);
            oo.close();
            System.exit(0);
        } else if (args[0].equals("old")) {
            ObjectInputStream oi =
                new ObjectInputStream(
                    new FileInputStream("frame.dat"));
            OuterData o=null;
            try {
                o = (OuterData) oi.readObject();
            } catch (ClassNotFoundException e) {}
            o.makeVisible();
        } else {
            System.out.println("Usage: SerialTest old|new");
            return;
        }
    }
}
```

//

Несколько комментариев.

- Обычно о сериализации думают в применении к деревьям, массивам и пр. В примере мы видим, что и кнопки сериализуются вполне успешно. Это наводит на глубокие мысли, хотя и не ясно, какие именно.
- Как обычно в Яве, `readObject` возвращает `Object`. Его нужно явно приводить к тому типу, который нужен.
- Пытливый читатель заметил, что `readObject` может генерировать `ClassNotFoundException`. Понятно, что это означает: что мы прочитали объект класса, которого у нас нет. И действительно, при сериализации сериализуются лишь данные, но не код[24]. Кстати говоря, размеры сериализованных объектов весьма скромные, например, `frame.dat` получается что-то вроде 9КБ.
- `ActionListener` добавляется уже после десериализации. Если переместить добавление обработчика в конструктор, то десериализованная кнопка никаких событий обрабатывать не будет. Так происходит потому, что обработчик на самом деле принадлежит не кнопке.
- Проделаем нехитрый эксперимент. Создадим `frame.dat`, после чего добавим в `OuterData` поле целого типа, и попытаемся запустить десериализацию. Нам объяснят, что по чём, выбросив исключение `java.io.InvalidClassException` и объяснят, что классы несовместимы. Понятно, что в реальной жизни такое ограничение — слишком жёсткое, и нужно уметь старые данные новым кодом. Как это делается — см. Java Object Serialization Specification, гл.5 (`http://java.sun.com/j2se/1.3/docs/guide/serialization/`).

Довольно часто бывает, что данные сериализовать либо бессмысленно (открытый сокет), либо недопустимо (plain-text пароли). Такие данные нужно описывать с ключевым словом `transient`. После десериализации указатели на объекты будут равны `null`, а, к примеру, `int` — 0.

Если объект хочет полностью взять процесс сериализации в свои руки, он должен реализовывать интерфейс `java.io.Externalizable`. В этом случае пользователю необходимо реализовать методы `writeExternal` и `writeExternal`, которые и будут вызваны в соответствующие моменты. Методам передаётся аргумент `ObjectOutput`, куда нужно записывать то, что хочется в, возможно, самобытном формате. Осмысленное применение этой возможности, к примеру — сжатие при сериализации. Если же использовать "автоматический" `java.io.Serializable`, то никакого сжатия не будет (ведь никто не знает, что хранящийся массив байтов — именно изображение).

## 5.3 RMI

RMI (Remote Method Invocation) — это стандартное низкоуровневое средство удалённого взаимодействия в Яве. В базовом варианте работающая система состоит из 3-х частей: сервера (см. `RemWin.java`), клиента (см. `RemWinUser.java`) и интерфейса сервера (см. `RemWinI.java`) Идея тут вот в чём: первым делом сервер регистрирует (где-то), какие объекты у него можно вызывать, затем клиент звонит в этот репозиторий и пытается получить интерфейс удалённого объекта, с которым и работает, вызывая его методы.

[24] Механизм, обеспечивающий загрузку кода в Яве, называется ClassLoader, туда и нужно смотреть, если вам, например, нужна загрузка из экзотического источника, скажем, сокета.

// RemWinI.java

```
package ru.test;

import java.net.MalformedURLException;
import java.rmi.AlreadyBoundException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

public class RemWin extends UnicastRemoteObject implements RemWinI {
    private int currSessId = 0;
    public RemWin() throws RemoteException {}
    /**
     * @see ru.test.RemWinI#login(String, String)
     */
    public Integer login(String userName, String userPassword)
        throws RemoteException {
        return new Integer(currSessId++);
    }
    public RemotePrinterI getRemotePrinter(Integer sessionHandle)
        throws RemoteException {
        RemotePrinterI p = new RemotePrinter();
        return p;
    }

    public static void main(String[] args)
        throws
            RemoteException,
            MalformedURLException,
            AlreadyBoundException {

        LocateRegistry.createRegistry(1099);
        RemWin remWin = new RemWin();
        Naming.bind("//localhost:1099/RemWin", remWin);
    }
}
//
```

// RemWinI.java

```
package ru.test;

import java.rmi.*;

/** Interface to remote window creator
 */
public interface RemWinI extends Remote {
/**
 * Method login.to remote window system
 * @param userName
 * @param userPassword
 * @return Integer session handle or null
 */
Integer login(final String userName, final String userPassword)
throws RemoteException;
RemotePrinterI getRemotePrinter(Integer sessionHandle)
throws RemoteException;
}
//
```

// RemWinUser.java

```
package ru.test;

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

public class RemWinUser {

public static void main(String[] args)
throws MalformedURLException, RemoteException, NotBoundException {
RemWinI remWin =
(RemWinI) Naming.lookup("//localhost:1099/RemWin");
Integer in = remWin.login("u1305", "qqq");
System.out.println(in);
RemotePrinterI p = remWin.getRemotePrinter(in);
p.print("aaa");
}
}
//
```

// RemotePrinter.java

```
package ru.test;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class RemotePrinter
extends UnicastRemoteObject
implements RemotePrinterI {
RemotePrinter() throws RemoteException {}

public void print(final String s) {
System.out.println(s);
}
}
//
```

// RemotePrinterI.java

```
package ru.test;

import java.rmi.*;

public interface RemotePrinterI extends Remote {
void print(final String s) throws RemoteException;

}
//
```

Если бы в примере RemotePrinter не расширял UnicastRemoteObject, то попытка вернуть его из удалённого метода бы провалилась: серверная сторона выбросила бы исключение “Не могу сериализовать”. Если же попытаться “успокоить” run-time и реализовать интерфейс `java.io.Serializable`, поведение примера существенно изменится: печать будет происходить на стороне клиента. Возврат Integer проблем не вызывает, потому что он и так уже сериализованный.

Если проделать всё, что описано выше, ничего, тем не менее, работать не будет. Это потому, что носутствует важная часть, отвечающая за посылку и приём аргументов. Её генерирует утилита `rmic`, примерно вот так:

```
rmic -depend -classpath ~/workspace/examples ru.test.RemWin
```

На самом деле, она генерирует Явовские исходники, а затем напускает на них компилятор. Посмотреть, что делается на самом деле можно, добавив ключ `-keep`.

## 5.4 Сервлеты

Все примеры пробовались в Tomcat 4.0.3. Для того, чтобы всё заработало, необходимо добавить в server.xml строчку `<Context path="/samples" docBase="samples" />`.

Сервлет в примитивном варианте выглядит примерно так:

// HelloSess.java

```
package ru.test;

import java.io.*;
import javax.servlet.http.*;

public class HelloSess extends HttpServlet {
    public void service(
        HttpServletRequest req,
        HttpServletResponse resp) throws IOException {
            HttpSession sess = req.getSession();

            resp.setContentType("text/html");
            PrintWriter out = resp.getWriter();

            out.println("<html> last from");
            String privHost = (String)sess.getAttribute("prehost");
            if (null != privHost)
                out.println(privHost);
            else
                out.println("nowhere");
            sess.setAttribute("prehost", req.getRemoteHost());
            out.println("/<html>");
            out.close();
        }
}
//
```

Обсуждения здесь заслуживает следующее. Во-первых, это сессии, которые мы по мере необходимости достаём из запроса. Во-вторых, это модель параллельности. Подвох в следующем. Код выглядит и работает совершенно невинно, если не задумываться о том, что будет, когда объявяться два пользователя сразу. Оказывается, тогда будут одновременно вызваны методы `service` для **одного экземпляра** класса `ru.test.HelloSess`. Есть уозможность потребовать, чтобы в разных вызовах использовались разные объекты, реализовав (маркерный) интерфейс `SingleThreadModel`, но статические данные по-прежнему будут подвергаться испытанием параллельностью. Программист, не учитывающий этого, просто-таки гарантирует себе интересную жизнь на этапе внедрения (отлаживая всё в "настольном" режиме, очень легко понаписать километры кода, даже не задумавшись о параллельности).

# 6 J2EE

Если в собственно Яве причудливо перемешались черты языка и ОС, то архитектура J2EE представляет собой причудливую смесь чудес Объектно-Ориентированного Проектирования и реальной потребности приложений в ОС следующего поколения.

По крайней мере, продираться скозь толстые Sun-овские спецификации очень помогает представление, что "компонент"[25] — это такой процесс, только очень специальный.

Неверно воспринимать обсуждаемые вопросы как специфически Явовские — просто объем денег, вложенных в Явовскую инфраструктуру, позволяет именно здесь обсуждать идейно важные вопросы, которые в менее развитых средах маскируются техническими трудностями (но никуда не исчезают!).

## 6.1 Принципы J2EE

Назначение технологии J2EE — облегчить написание бизнес-логики, располагающейся на среднем уровне классической 3-звёнки. Принцип облегчения следующий: всё, что нужно всем приложениям, реализует система, на долю реализатора конкретного приложения остаётся лишь написать специфичные детали. Например, создание "сессии" — обычно дорогостоящая операция, и потому сессии обычно создают впрок, ещё до прихода клиентов. Так вот, если используется J2EE, то разработчик приложения напишет 3 присваивания в инициализаторе, администратор приложения укажет, сколько клиентов ожидается, а остальное сделает "движок" (сервер приложений, AppServer).

В состав J2EE входит масса разнообраных технологий: Servlet API (обработчики HTTP-запросов), JTA (управление транзакциями), JMS (передача асинхронных сообщений), JMX и пр., но, из-за ограниченности курса, нас будут интересовать почти исключительно EJB.

Enterprise Java Beans — это серверные компоненты[26], реализующие логику приложения, они бывают 3-х сортов.

- Session Beans. Компоненты для представления процессов, связанных с текущей сессией ограниченным временем жизни. Подразделяются ещё на 2 подсорта:
  - Stateless. Компонент, не хранящий никакого состояния, связанного с конкретным клиентом. В качестве (глупой, но запоминающейся) ментальной модели можно предложить Очень Быстрого Считателя Синусов: для всех клиентов значение синуса одинаково, но компонент, возможно, захочет кешировать вычисленные синусы, а не перевычислять их.
  - Stateful. Компонент с состоянем, могущий быть привязанным к конкретному клиенту.
- Entity Beans. Проще всего воспринимать как (облагороженную) строчку в реляционной БД.
- Message-Driven Beans. Оживают асинхронно, по приходу сообщений.

[25] Bean, естественно.

[26] Пугаться не стоит, во всей Явовской идеологии компонент — это просто объект, удовлетворяющий набору ограничений, некотороые из них проверяются транслятором, некоторые в run-time, а некоторые вообще не проверяются.

### 6.1.1 Примеры EJB

## 6.2 Паттерны для middleware

Ситуация с книжками-рассказками про паттерны для J2EE — очень характерная. Что в них, в сущности, от EJB и J2EE? Среда исполнения, для которой примеры приводятся. Конечно же, если писать на C (и на чём угодно), "типичные решения типичных проблем" при разработке middleware останутся прежними.

Рассмотрим, к примеру, классический паттерн **Session facade**. Его идея — в следующем. Если клиенту необходимо получить доступ к нескольким EB в рамках одной транзакции, то необходимо создать единственный SB (это и есть фасад сессии), из которого уже и обращаться к EB. Что здесь специфически-Явовского? Совсем ничего: в любом разумном рассказе про иерехическую организацию систем, коечно же, написано, что перескакивать через уровни при обращении нехорошо, но там это один абзац. А вот в книжке Маринеску [4] эта мысль изложена на 8 страницах, с (печальными) примерами неиспользования и (печальными) примерами злоупотребления, картинками на UML и пр.

## 6.3 JBoss: реализация

Из вышеизложенного понятно, что сервер приложений, в котором крутятся EJB — весьма непростая штука. Сейчас рынок серверов приложений переживает весьма забавную стадию: когда всё это только начиналось, года 2–3 назад, сервера приложений исчислялись дюжинами, ныне же выделилась "тройка лидеров"[27], а все остальные потихоньку усыхают. Здесь, видимо, уместна аналогия с рынком РСУБД, где тоже осталось полдюжины игроков, тем более что цены на продукты вполне похожие: просить $5k за лицензию на один процессор считается вполне нормальным. Естественно, Open Source Community не могло остаться в стороне, и существуют как минимум 2 "продукта" со свободной лицензией: JOnAS (`http://www.objectweb.org/jonas/index.html`) и JBoss (`http://www.jboss.org/`).

Нас, впрочем, будут интересовать не воспоминания о будущем рынка и не инженерное сравнение серверов приложений, а их внутренняя организация. Благо, в тексты можно заглянуть. Заглядывать мы будем в JBoss.

### 6.3.1 JMX и MBean-ы

*Подраздел предназначен для дополнительного чтения.*

Свободные разработчики JBoss-а сильно гордятся его гибкой, модульной и "микроядерной" архитектурой[28]. Её основа — ещё одна компонентная модель, разработанная Саном Java Management eXtensions (`http://java.sun.com/products/JavaManagement/`), документация "от производителя" больше всего напомнила мне RFC на SNMP: описыватся Универсальное Управление Всем, гибкое, динамическое и расширяемое, при этом совершенно непонятно, к чему бы в текущей реальности это применить. А вот разработчикам JBoss-а оказалось понятно. Коротко рассмотрим возможности JMX, которые используются в JBoss-е.

В JMX выделяют 3 иерархических уровня:

- Ресурсы (облагороженные), которыми нужно управлять.
- Агенты, которые управляют ресурсами.
- Распределённые сервисы, управляющие агентами.

[27] Bea, IBM и IPlanet, недавно поглощённый Саном.

[28] Вообще, термин "микроядро" стал моден среди Явовских разработчиков через 10 лет после разработчиков ядер ОС; пока это, скорее, просто buzzword. Вот ещё один любопытный проект, заявляющий о своей "микроядерности": `http://jakarta.apache.org/avalon/phoenix/index.html`

Для управления ресурсы представляются в унифицированном виде MBean-ов. Стандарт определяет 4 типа MBean-ов (упорядочены по степени возрастания динамики): standard, dynamic, model, и open. В настоящее время в реализации JBoss-а используются только 2 первых типа.

- Стандатные MBean-ы — это просто обычные объекты классов, реализующие обеспечивающий управление интерфейс.
- Динамические реализуют интерфейс `javax.management.DynamicMBean`, через который можно получать интерфейс управления в run-time.

Как и обычно в компонентных моделях, для внешнего мира MBean состоит из:

- Значениями атрибутов, доступных по имени.
- Методами и конструкторами.
- Событиями, которые компонент генерирует.

С атрибутами и методами всё понятно, кратко обсудим модель событий. Понятно, в общем, зачем они нужны: если метод — это способ *попросить* объект о чём-то, то событие даёт возможность объекту *оповестить* внешний мир о чём-то. В Явовской модели событий все события наследуют от `java.util.EventObject`. События JMX наследуют от `javax.management.Notifications`[29].

### 6.3.2 Inside JBoss

# 7 Упражнения

На основе разработанных в курсе “Технология программирования” проектов, реализовать прототипы в среде J2EE. Понятно, что досконально изучить используемые технологии времени у нас нет, поэтому при написании нужно активно перерабатывать “под себя” примеры из лекций. Собственно говоря, в “реальной жизни”, как правило, “досконально изучаемая” технология успевает устареть раньше, чем закончится изучение, так что тренируйтесь...

[29] Которая наследует от `java.util.EventObject`.