

Java: Русские буквы и не только...

Сергей Астахов

1. Введение

Некоторые проблемы настолько сложны, что нужно быть очень умным и очень хорошо информированным, чтобы не быть уверенным в их решении.

Лоренс Дж. Питер
Peter's Almanac

1.1. Кодировки

Когда я только начинал программировать на языке C, первой моей программой (не считая HelloWorld) была программа перекодировки текстовых файлов из основной кодировки ГОСТ-а (помните такую? :-)) в альтернативную. Было это в далёком 1991-ом году. С тех пор многое изменилось, но за прошедшие 10 лет подобные программки свою актуальность, к сожалению, не потеряли. Слишком много уже накоплено данных в разнообразных кодировках и слишком много используется программ, которые умеют работать только с одной. Для русского языка существует не менее десятка различных кодировок, что делает проблему ещё более запутанной.

Откуда же взялись все эти кодировки и для чего они нужны? Компьютеры по своей природе могут работать только с числами. Для того чтобы хранить буквы в памяти компьютера надо поставить в соответствие каждой букве некое число (примерно такой же принцип использовался и до появления компьютеров — вспомните про ту же азбуку Морзе). Причём число желательно поменьше — чем меньше двоичных разрядов будет задействовано, тем эффективнее можно будет использовать память. Вот это соответствие набора символов и чисел собственно и есть кодировка. Желание любой ценой сэкономить память, а так же разобщённость разных групп компьютерщиков и привела к нынешнему положению дел. Самым распространённым

способом кодирования сейчас является использование для одного символа одного байта (8 бит), что определяет общее кол-во символов в 256. Набор первых 128 символов стандартизован (набор ASCII) и является одинаковыми во всех распространённых кодировках (те кодировки, где это не так уже практически вышли из употребления). Англицкие буквы и символы пунктуации находятся в этом диапазоне, что и определяет их поразительную живучесть в компьютерных системах :-). Другие языки находятся не в столь счастливом положении — им всем приходится ютиться в оставшихся 128 числах.

1.2. Unicode

В конце 80-х многие осознали необходимость создания единого стандарта на кодирование символов, что и привело к появлению Unicode. [Unicode](#) — это попытка раз и навсегда зафиксировать конкретное число за конкретным символом. Понятно, что в 256 символов тут не уложишься при всём желании. Довольно долгое время казалось, что уж 2-х то байт (65536 символов) должно хватить. Ан нет — последняя версия стандарта Unicode ([3.1](#)) определяет уже 94140 символов. Для такого кол-ва символов, наверное, уже придётся использовать 4 байта (4294967296 символов). Может быть и хватит на некоторое время... :-)

В набор символов Unicode входят всевозможные буквы со всякими чёрточками и припендюльками, греческие, математические, иероглифы, символы псевдографики и пр. и пр. В том числе и так любимые нами символы кириллицы (диапазон значений 0x0400-0x04ff). Так что с этой стороны никакой дискриминации нет.

Если Вам интересны конкретные коды символов, для их просмотра удобно использовать программу "Таблица символов" из WinNT. Вот, например, диапазон кириллицы:

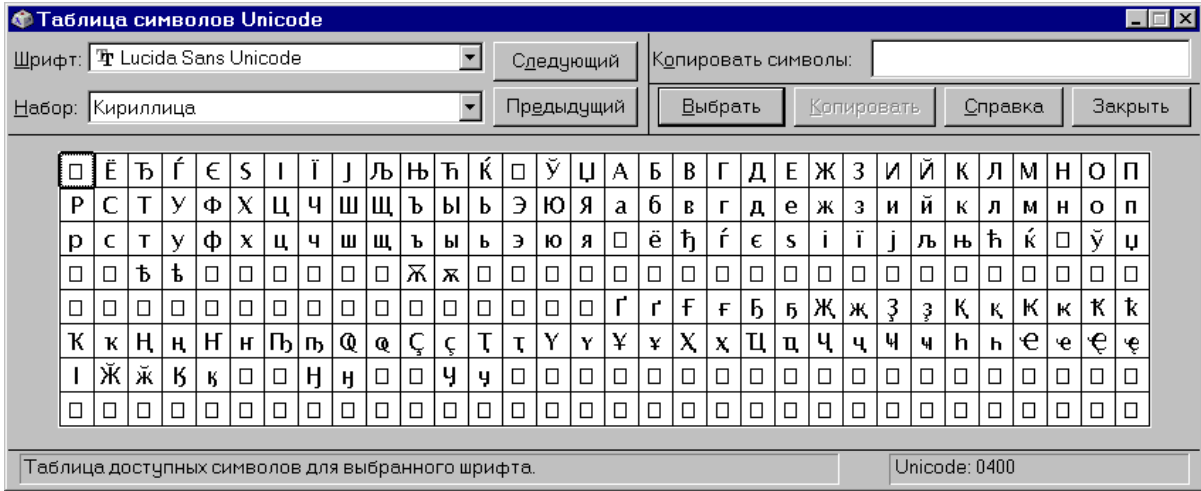


Рисунок 1. Таблица символов из WinNT.

Если у Вас другая OS или Вас интересует официальное толкование, то полную раскладку символов (charts) можно найти на официальном сайте Unicode (<http://www.unicode.org/charts/web.html>).

1.3. Типы char и byte

В Java для символов выделен отдельный тип данных `char` размером в 2 байта. Это часто порождает путаницу в умах начинающих (особенно если они раньше программировали на других языках, например на C/C++). Дело в том, что в большинстве других языков для обработки символов используются типы данных размером в 1 байт. Например, в C/C++ тип `char` в большинстве случаев используется как для обработки символов, так и для обработки байтов — там нет деления. В Java для байтов имеется свой тип — тип `byte`. Таким образом C-ишному `char` соответствует Java-вский `byte`, а Java-вскому `char` из мира C ближе всего тип `wchar_t`. Надо чётко разделять понятия символов и байтов — иначе непонимание и проблемы гарантированы.

Java практически с самого своего рождения использует для кодирования символов стандарт Unicode. Библиотечные функции Java ожидают увидеть в переменных типа `char` символы, представленные кодами Unicode. В принципе, Вы, конечно, можете запихнуть туда что угодно — цифры есть цифры, процессор всё стерпит, но при любой обработке библиотечные функции будут действовать исходя из предположения что им

передали кодировку Unicode. Так что можно спокойно полагать, что у типа `char` кодировка зафиксирована. Но это внутри JVM. Когда данные читаются извне или передаются наружу, то они могут быть представлены только одним типом — типом `byte`. Все прочие типы конструируются из байтов в зависимости от используемого формата данных. Вот тут то на сцену и выходят кодировки — в Java это просто формат данных для передачи символов, который используется для формирования данных типа `char`. Для каждой кодовой страницы в библиотеке имеется по 2 класса перекодировки (`ByteToChar` и `CharToByte`). Классы эти лежат в пакете `sun.io`. Если, при перекодировке из `char` в `byte` не было найдено соответствующего символа, он заменяется на символ `?`.

Кстати, эти файлы кодовых страниц в некоторых ранних версиях JDK 1.1 содержат ошибки, вызывающие ошибки перекодировок, а то и вообще исключения при выполнении. Например, это касается кодировки `KOI8_R`. Лучшее, что можно при этом сделать — сменить версию на более позднюю. Судя по Sun-овскому описанию, большинство этих проблем было решено в версии JDK 1.1.6.

До появления версии JDK 1.4 набор доступных кодировок определялся только производителем JDK. Начиная с 1.4 появилось новое API (пакет `java.nio.charset`), при помощи которого Вы уже можете создать свою собственную кодировку (например поддержать редко используемую, но жутко необходимую именно Вам).

1.4. Класс `String`

В большинстве случаев для представления строк в Java используется объект типа `java.lang.String`. Это обычный класс, который внутри себя хранит массив символов (`char[]`), и который содержит много полезных методов для манипуляции символами. Самые интересные — это конструкторы, имеющие первым параметром массив байтов (`byte[]`) и методы `getBytes()`. При помощи этих методов Вы можете выполнять преобразования из массива байтов в строки и обратно. Для того, чтобы указать какую кодировку при этом использовать у этих методов есть строковый параметр, который задаёт её имя. Вот, например, как можно выполнить перекодировку байтов из КОИ-8 в Windows-1251:

Java: Русские буквы и не только...

```
// Данные в кодировке КОИ-8
byte[] koi8Data = ...;
// Преобразуем из КОИ-8 в Unicode
String string = new String(koi8Data, "KOI8_R");
// Преобразуем из Unicode в Windows-1251
byte[] winData = string.getBytes("Cp1251");
```

Список 8-ми битовых кодировок, доступных в современных JDK и поддерживающих русские буквы Вы можете найти ниже, в разделе "8-ми битовые кодировки русских букв".

Так как кодировка — это формат данных для символов, кроме знакомых 8-ми битовых кодировок в Java также на равных присутствуют и многобайтовые кодировки. К таким относятся UTF-8, UTF-16, Unicode и пр. Например вот так можно получить байты в формате UnicodeLittleUnmarked (16-ти битовое кодирование Unicode, младший байт первый, без признака порядка байтов):

```
// Строка Unicode
String string = "...";
// Преобразуем из Unicode в UnicodeLittleUnmarked
byte[] data = string.getBytes("UnicodeLittleUnmarked");
```

При подобных преобразованиях легко ошибиться — если кодировка байтовых данных не соответствуют указанному параметру при преобразовании из `byte` в `char`, то перекодирование будет выполнено неправильно. Иногда после этого можно выпатчить правильные символы, но чаще всего часть данных будет безвозвратно потеряна.

В реальной программе явно указывать кодовую страницу не всегда удобно (хотя более надёжно). Для этого была введена кодировка по умолчанию. По умолчанию она зависит от системы и её настроек (для русских виндов принята кодировка Cp1251), и в старых JDK её можно изменить установкой системного свойства `file.encoding`. В JDK 1.3 изменение этой настройки иногда срабатывает, иногда — нет. Вызвано это следующим: первоначально `file.encoding` ставится по региональным настройкам компьютера. Ссылка на кодировку по умолчанию запоминается в нутрях при первом преобразовании. При этом используется `file.encoding`, но это преобразование происходит ещё до использования аргументов запуска JVM (собсно, при их разборе). Вообще-то, как утверждают в Sun, это свойство отражает системную кодировку, и она не должна изменяться в командной строке (см., например, комментарии к BugID

[4163515](#)) Тем не менее в JDK 1.4 Beta 2 смена этой настройки опять начала оказывать эффект. Что это, сознательное изменение или побочный эффект, который может опять исчезнуть — Sun-овцы ясного ответа пока не дали.

Эта кодировка используется тогда, когда явно не указано название страницы. Об этом надо всегда помнить — Java не будет пытаться предсказать кодировку байтов, которые Вы передаёте для создания строки String (так же она не сможет прочитать Ваши мысли по этому поводу :-). Она просто использует текущую кодировку по умолчанию. Т.к. эта настройка одна на все преобразования, иногда можно наткнуться на неприятности.

Для преобразования из байтов в символы и обратно следует пользоваться **только** этими методами. Простое приведение типа использовать в большинстве случаев нельзя — кодировка символов при этом не будет учитываться. Например, одной из самых распространённых ошибок является чтение данных побайтно при помощи метода read() из InputStream, а затем приведение полученного значения к типу char:

```
InputStream is = ..;

int b;
StringBuffer sb = new StringBuffer();

while( (b=is.read())!=-1 )
{
    sb.append( (char)b ); // <- так делать нельзя
}

String s = sb.toString();
```

Обратите внимание на приведение типа — "(char)b". Значения байтов вместо перекодирования просто копируются в char (диапазон значений 0-0xFF, а не тот, где находится кириллица). Такому копированию соответствует кодировка ISO-8859-1 (которая один в один соответствует первым 256 значениям Unicode), а значит, можно считать, что этот код просто использует её (вместо той, в которой реально закодированы символы в оригинальных данных). Если Вы попытаетесь отобразить полученное значение — на экране будут или вопросики или кракозяблы. Например, при чтении строки "АБВ" в виндовой кодировке может запросто отобразиться что-то вроде такого: "АБВ". Подобного рода код часто пишут программисты на западе — с

Java: Русские буквы и не только...

английскими буквами работает, и ладно. Исправить такой код легко — надо просто заменить `StringBuffer` на `ByteArrayOutputStream`:

```
InputStream is = ..;

int b;
ByteArrayOutputStream baos = new ByteArrayOutputStream();

while( (b=is.read())!=-1 )
{
    baos.write( b );
}

// Перекодирование байтов в строку с использованием кодировки по умолчанию

String s = baos.toString();

// Если нужна конкретная кодировка —
// просто укажите её при вызове toString():
//
// s = baos.toString("Cp1251");
```

Более подробно о распространённых ошибках смотрите раздел Типичные ошибки.

2. 8-ми битовые кодировки русских букв

Вот основные 8-ми битовые кодировки русских букв, получившие распространение:

Кодировка	Ареал распространения	Основное название в Java
IBM-866	MS-DOS, Windows (OEM-кодировка), OS/2	Cp866
Windows-1251	Windows (Ansi-кодировка)	Cp1251
КОИ-8	Unix, большинство русскоязычных писем в Internet	KOI8_R
ISO-8859-5	Unix	ISO8859_5
Macintosh Cyrillic	Mac	MacCyrillic

Помимо основного названия можно использовать синонимы. Набор их может отличаться в разных версиях JDK. Вот список от JDK 1.3.1:

- Cp1251:
 - Windows-1251
- Cp866:
 - IBM866
 - IBM-866
 - 866
 - CP866
 - CSIBM866
- KOI8_R:
 - KOI8-R
 - KOI8
 - CSKOI8R
- ISO8859_5:
 - ISO8859-5
 - ISO-8859-5
 - ISO_8859-5
 - ISO_8859-5:1988
 - ISO-IR-144
 - 8859_5
 - Cyrillic
 - CSISOLatinCyrillic
 - IBM915
 - IBM-915
 - Cp915
 - 915

Причём синонимы, в отличие от основного имени нечувствительны к регистру символов - такова особенность реализации.

Стоит отметить, что эти кодировки на некоторых JVM могут отсутствовать. Например, с сайта Sun можно скачать две разные версии JRE — US и International. В US версии присутствует только минимум — ISO-8859-1, ASCII, Cp1252, UTF8, UTF16 и несколько вариаций двухбайтового Unicode. Всё прочее есть только в International

Java: Русские буквы и не только...

варианте. Иногда из-за этого можно нарваться на грабли с запуском программы, даже если ей не нужны русские буквы. Типичная ошибка, возникающая при этом:

```
Error occurred during initialization of VM
  java/lang/ClassNotFoundException: sun/io/ByteToCharCp1251
```

Возникает она, как не трудно догадаться, из-за того, что JVM, исходя из русских региональных настроек пытается установить кодировку по умолчанию в Cp1251, но, т.к. класс поддержки таковой отсутствует в US версии, закономерно обламывается.

3. Файлы и потоки данных

Так же как и байты концептуально отделены от символов, в Java различаются потоки байтов и потоки символов. Работу с байтами представляют классы, которые прямо или косвенно наследуют классы `InputStream` или `OutputStream` (плюс класс-уникум `RandomAccessFile`). Работу с символами представляет сладкая парочка классов `Reader/Writer` (и их наследники, разумеется).

Для чтения/записи не преобразованных байтов используются потоки байтов. Если известно, что байты представляют собой только символы в некоторой кодировке, можно использовать специальные классы-преобразователи `InputStreamReader` и `OutputStreamWriter`, чтобы получить поток символов и работать непосредственно с ним. Обычно это удобно в случае обычных текстовых файлов или при работе с многими сетевыми протоколами Internet. Кодировка символов при этом указывается в конструкторе класса-преобразователя. Пример:

```
// Строка Unicode
String string = "...";

// Записываем строку в текстовый файл в кодировке Cp866

PrintWriter pw = new PrintWriter      // класс с методами записи строк
    (new OutputStreamWriter           // класс-преобразователь
     (new FileOutputStream              // класс записи байтов в файл
      ("file.txt"), "Cp866"));

pw.println(string); // записываем строку в файл
```

```
pw.close();
```

Если в потоке могут присутствовать данные в разных кодировках или же символы перемешаны с прочими двоичными данными, то лучше читать и записывать массивы байтов (`byte[]`), а для перекодировки использовать уже упомянутые методы класса `String`. Пример:

```
// Строка Unicode
String string = "...";

// Записываем строку в текстовый файл в двух кодировках (Cp866 и Cp1251)

// класс записи байтов в файл
OutputStream os = new FileOutputStream("file.txt");

// Записываем строку в кодировке Cp866

os.write( string.getBytes("Cp866") );

// Записываем строку в кодировке Cp1251

os.write( string.getBytes("Cp1251") );

os.close();
```

Консоль в Java традиционно представлена потоками, но, к сожалению, не символов, а байтов. Дело в том, что потоки символов появились только в JDK 1.1 (вместе со всем механизмом кодировок), а доступ к консольному вводу/выводу проектировался ещё в JDK 1.0, что и привело к появлению уродца в виде класса `PrintStream`. Этот класс используется в переменных `System.out` и `System.err`, которые собственно и дают доступ к выводу на консоль. По всем признакам это поток байтов, но с кучей методов записи строк. Когда Вы записываете в него строку, внутри происходит конвертация в байты с использованием кодировки по умолчанию, что в случае виндов, как правило, неприемлемо — кодировка по умолчанию будет `Cp1251` (Ansi), а для консольного окна обычно нужно использовать `Cp866` (OEM). Эта ошибка была зарегистрирована ещё в 97-ом году ([4038677](#)) но Sun-овцы исправлять её вроде не торопятся. Так как метода установки кодировки в `PrintStream` нет, для решения этой проблемы можно подменить

стандартный класс на собственный при помощи методов `System.setOut()` и `System.setErr()`. Вот, например, обычное начало в моих программах:

```
...
public static void main(String[] args)
{
    // Установка вывода консольных сообщений в нужной кодировке
    try
    {
        String consoleEnc = System.getProperty("console.encoding", "Cp866");
        System.setOut(new CodepagePrintStream(System.out, consoleEnc) );
        System.setErr(new CodepagePrintStream(System.err, consoleEnc) );
    }
    catch(UnsupportedEncodingException e)
    {
        System.out.println("Unable to setup console codepage: " + e);
    }
}
...
```

Исходники класса `CodepagePrintStream` Вы можете найти на данном сайте: [CodepagePrintStream.java](#).

Если Вы сами конструируете формат данных, я рекомендую Вам использовать одну из многобайтовых кодировок. Удобнее всего обычно формат UTF8 — первые 128 значений (ASCII) в нём кодируются одним байтом, что часто может значительно уменьшить общий объём данных (не зря эта кодировка принята за основу в мире XML). Но у UTF8 есть один недостаток — кол-во требуемых байтов зависит от кода символов. Там, где это критично можно использовать один из двухбайтовых форматов Unicode (`UnicodeBig` или `UnicodeLittle`).

4. Базы данных

Для того, чтобы прочитывать корректно символы из БД, обычно достаточно указать JDBC-драйверу на нужную кодировку символов в БД. Как именно — это зависит от конкретного драйвера. Сейчас уже многие драйвера поддерживают подобную настройку, в отличие от недавнего прошлого. Далее приведены несколько известных мне примеров.

4.1. Мост JDBC-ODBC

Это один из самых часто используемых драйверов. Мост из JDK 1.2 и старше можно легко настроить на нужную кодировку. Это делается добавлением дополнительного свойства `charSet` в набор параметров, передаваемых для открытия соединения с базой. По умолчанию используется `file.encoding`. Делается это примерно так:

```
// Параметры соединения с базой
Properties connInfo = new Properties();

connInfo.put("user", username);
connInfo.put("password", password);
connInfo.put("charSet", "Cp1251");

// Устанавливаем соединение
Connection db =
    DriverManager.getConnection(dataurl, connInfo);
```

4.2. Драйвер JDBC-OCI от Oracle 8.0.5 под Linux

При получении данных из БД, данный драйвер определяет "свою" кодировку при помощи переменной окружения `NLS_LANG`. Если эта переменная не найдена, то он считает что кодировка — ISO-8859-1. Весь фокус в том, что `NLS_LANG` должна быть именно переменной окружения (устанавливаемой командой `set`), а не системное свойство Java (типа `file.encoding`). В случае использования драйвера внутри `servlet engine Apache+Jserv`, переменную окружения можно задать в файле `jserv.properties`:

```
wrapper.env=NLS_LANG=American_America.CL8KOI8R
```

Информацию об этом прислал [Сергей Безруков](#), за что ему отдельное спасибо.

4.3. Драйвер JDBC-thin от Oracle

Сей драйвер вроде как не требует особой настройки. По крайней мере в документации об этом ни слова. По всей видимости у Oracle в протоколе обмена ходит нормальный Unicode. Единственная проблема, с которой многие сталкиваются — некорректная кодировка сообщений об ошибках. Дело в том, что оригинальные драйвера от Oracle

8.1.7 и 9.0.1 содержат некорректно сформированные файлы ресурсов с текстами русских сообщений. Драйвера от 9.0.2 и 9.2.x уже нормальные. Эти файлы ресурсов можно довольно легко пропатчить при помощи утилиты native2ascii. Готовый скрипт патча можно найти здесь: http://java.ksu.ru/sources/oracle/oracle_jdbc_repair.bat (или здесь - [oracle_jdbc_repair.bat](#)). Можно также поменять текущий язык сессии выполнив команду "ALTER SESSION SET NLS_LANGUAGE=English". При этом сообщения станут выдаваться на английском. Ну а самый правильный путь - использовать последние версии драйверов от 9.2.x, благо что они совместимы со старыми версиями Oracle и к тому же содержат исправления других ошибок.

4.4. Драйвер JDBC для работы с DBF (com.hxtt.sql.dbf.DBFDriver, бывший zyh.sql.dbf.DBFDriver)

Данный драйвер только недавно научился работать с русскими буквами. Настройка выполняется немного по разному в зависимости от версии драйвера. Версии Beta 5.4 (от 02.04.2002) и более поздние уже нормально воспринимают настройку charSet. В версиях Beta 5.2 (от 30.07.2001) и Beta 5.3 (30.11.2001), хоть он и сообщает по getPropertyInfo() что он понимает свойство charSet — это фикция. Реально же настроить кодировку можно установкой свойства CODEPAGEID. Для русских символов там доступны два значения - "66" для Cp866 и "C9" для Cp1251. Пример:

```
// Параметры соединения с базой
Properties connInfo = new Properties();

// Кодировка Cp866
connInfo.put("charSet", "Cp866");
connInfo.put("CODEPAGEID", "66");

// Устанавливаем соединение
Connection db = DriverManager.getConnection("jdbc:DBF:/C:/MyDBFFiles", connInfo);
```

Если у Вас DBF-файлы формата FoxPro, то у них своя специфика. Дело в том, что FoxPro сохраняет в заголовке файла ID кодовой страницы (байт со смещением 0x1D), которая использовалась при создании DBF-ки. При открытии таблицы драйвер использует значение из заголовка, а не параметр "CODEPAGEID" (параметр в этом случае используется только при создании новых таблиц). Соответственно, чтобы всё

работало нормально, DBF-файл должен быть создан с правильной кодировкой — иначе будут проблемы.

4.5. MySQL (org.gjt.mm.mysql.Driver)

С этим драйвером тоже всё довольно просто:

```
// Параметры соединения с базой
Properties connInfo = new Properties();

connInfo.put("user", user);
connInfo.put("password", pass);

connInfo.put("useUnicode", "true");
connInfo.put("characterEncoding", "KOI8_R");

Connection conn = DriverManager.getConnection(dbURL, props);
```

4.6. InterBase (interbase.interclient.Driver)

Для этого драйвера работает параметр "charSet":

```
// Параметры соединения с базой
Properties connInfo = new Properties();

connInfo.put("user", username);
connInfo.put("password", password);
connInfo.put("charSet", "Cp1251");

// Устанавливаем соединение
Connection db = DriverManager.getConnection(dataurl, connInfo);
```

Однако не забудьте при создании БД и таблиц указать кодировку символов. Для русского языка можно использовать значения "UNICODE_FSS" или "WIN1251".
Пример:

```
CREATE DATABASE 'E:\ProjectHolding\DataBase\HOLDING.GDB' PAGE_SIZE 4096
DEFAULT CHARACTER SET UNICODE_FSS;
```

Java: Русские буквы и не только...

```
CREATE TABLE RUSSIAN_WORD
(
  "NAME1"  VARCHAR(40) CHARACTER SET UNICODE_FSS NOT NULL,
  "NAME2"  VARCHAR(40) CHARACTER SET WIN1251 NOT NULL,
  PRIMARY KEY ("NAME1")
);
```

В версии 2.01 InterClient присутствует ошибка — классы ресурсов с сообщениями для русского языка там неправильно скомпилированы. Скорей всего разработчики просто забыли указать кодировку исходников при компиляции. Есть два пути исправления этой ошибки:

- Использовать `interclient-core.jar` вместо `interclient.jar`. При этом русских ресурсов просто не будет, и автоматом подхватятся английские.
- Перекомпилировать файлы в нормальный Unicode. Разбор class-файлов — дело неблагодарное, поэтому лучше воспользоваться JAD-ом. К сожалению JAD, если встречает символы из набора ISO-8859-1, выводит их в 8-ричной кодировке, так что воспользоваться стандартным перекодировщиком `native2ascii` не удастся — придётся написать свой ([программа Decode](#)). Если Вам не хочется заморачиваться с этими проблемами - можете просто взять готовый файл с ресурсами (пропатченный jar с драйвером — [interclient.jar](#), отдельные классы ресурсов — [interclient-rus.jar](#)).

4.7. JayBird (org.firebirdsql.jdbc.FBDriver)

Для этого драйвера указание кодировки отличается от InterClient:

```
// Параметры соединения с базой
Properties connInfo = new Properties();

connInfo.put("user", username);
connInfo.put("password", password);
connInfo.put("lc_ctype", "WIN1251");

// Устанавливаем соединение
Connection db = DriverManager.getConnection(dataurl, connInfo);
```

4.8. SAP DB (com.sap.dbtech.jdbc.DriverSapDB)

Для этого драйвера можно включить использование Unicode задав параметр `unicode` в `true`. В противном случае будет использована жёстко зашитая ISO-8859-1, с выпекающими отсюда осточертевшими "?????". Кроме того существует доработанная напильником версия ([sapdbc.zip](#)), которая позволяет задавать параметр `charSet`.

4.9. DataSource (javax.sql.DataSource)

Если получение соединения производится через использование `DataSource`, то поддержку настройки используемой кодировки должна содержать конкретная реализация, которая регистрируется в JNDI. Для примера можно взять реализацию от `InterClient` (`interbase.interclient.DataSource`). В этом классе есть метод `setCharSet()`, используя который можно указать необходимую кодировку. Пример:

```
interbase.interclient.DataSource dataSource =
    new interbase.interclient.DataSource();
dataSource.setServerName("pongo");
dataSource.setDatabaseName("/databases/employee.gdb");
dataSource.setCharSet("Cp1251");

javax.naming.Context context = new javax.naming.InitialContext();
context.bind("jdbc/EmployeeDB", dataSource);
```

Другой пример — реализация от DBF-драйвера (`com.hxtt.sql.HxttDataSource`):

```
Properties prop=new Properties();
prop.setProperty("urlPrefix","jdbc:DBF:");
prop.setProperty("user","aaaaa");
prop.setProperty("password","vvvvccc");
prop.setProperty("database","dbffiles");//Change it to yourdbfdir

prop.setProperty("charSet","Cp866");

DataSource dataSource=new com.hxtt.sql.HxttDataSource(prop);

context.bind(name, dataSource);
```

Но даже настроив JDBC-драйвер на нужную кодировку в некоторых случаях можно нарваться на неприятности. Например, при попытке использования новых

замечательных скролируемых курсоров стандарта JDBC 2 в мосте JDBC-ODBC из JDK 1.3.x довольно быстро можно обнаружить, что русские буквы там просто не работают (метод `updateString()`).

С этой ошибкой связанна небольшая история. Когда я впервые её обнаружил (под JDK 1.3 rc2), я зарегистрировал её на BugParade ([4335564](#)). Когда вышла первая бета JDK 1.3.1, эту ошибку поместили как пофиксеную. Обрадованный, я скачал эту бету, запустил тест — не работает. Написал Sun-овцам об этом — в ответ мне написали, что фикс будет включён в последующие релизы. Ну ладно, подумал я, подождём. Время шло, вышел релиз 1.3.1, а потом и бета 1.4. Наконец я нашёл время проверить — опять не работает. Мать, мать, мать... — привычно откликнулось эхо. После гневного письма в Sun они завели новую ошибку ([4486195](#)), которую отдали на растерзание индийскому филиалу. Индусы пошаманили над кодом, и заявили, что в 1.4 beta3 всё исправлено. Скачал я эту версию, запустил под ним тестовый пример, вот результат — [4546083](#). Как оказалось, та beta3, которую раздают на сайте (build 84) это не та beta3, куда был включён окончательный фикс (build 87). Теперь обещают, что исправление войдёт в 1.4 rc1... Ну, в общем, Вы понимаете :-)

5. Русские буквы в исходниках Java-программ

Как уже упоминалось, при выполнении программы используется Unicode. Исходные же файлы пишутся в обычных редакторах. Я пользуюсь Far-ом, у Вас, наверняка есть свой любимый редактор. Эти редакторы сохраняют файлы в 8-битовом формате, а значит, что к этим файлам также применимы рассуждения, аналогичные приведённым выше. Разные версии компиляторов немного по-разному выполняют преобразование символов. В ранних версиях JDK 1.1.x используется настройка `file.encoding`, которую можно поменять при помощи нестандартной опции `-J`. В более новых (как сообщил [Денис Кокарев](#) — начиная с 1.1.4) был введён дополнительный параметр `-encoding`, при помощи которого можно указать используемую кодировку. В скомпилированных классах строки представлены в виде Unicode (точнее в модифицированном варианте формата UTF8), так что самое интересное происходит при компиляции. Поэтому, самое главное — выяснить, в какой кодировке у Вас исходники и указать правильное значение при компиляции. По умолчанию будет использован всё тот же пресловутый `file.encoding`. Пример вызова компилятора:

```
javac -encoding=KOI8_R ...
```

Кроме использования этой настройки есть ещё один метод — указывать буквы в формате "\uXXXX", где указывается код символа. Этот метод работает со всеми версиями, а для получения этих кодов можно использовать стандартную утилиту `native2ascii`.

Если Вы пользуетесь каким-либо IDE, то у него могут быть свои глюки. Зачастую эти IDE пользуются для чтения/сохранения исходников кодировку по умолчанию — так что обращайте внимание на региональные настройки своей ОС. Кроме этого могут быть и явные ошибки - например довольно неплохая IDE-шка [CodeGuide](#) плохо переваривает заглавную русскую букву "Т". Встроенный анализатор кода принимает эту букву за двойную кавычку, что приводит к тому, что корректный код воспринимается как ошибочный. Бороться с этим можно (заменой буквы "Т" на её код "\u0422"), но неприятно. Судя по всему где-то внутри парсера применяется явное преобразование символов в байты (типа: `byte b = (byte) c`), поэтому вместо кода `0x0422` (код буквы "Т") получается код `0x22` (код двойной кавычки).

Другая проблема встречается у JBuilder, но она больше связана с эргономикой. Дело в том, что в JDK 1.3.0, под которым по умолчанию работает JBuilder, имеется бага ([4312168](#)), из-за которой вновь создаваемые окна GUI при активизации автоматически включают раскладку клавиатуры в зависимости от региональных настроек ОС. Т.е. если у Вас русские региональные настройки, то он будет постоянно пытаться переключиться на русскую раскладку, что при написании программ жутко мешается. На сайте [JBuilder.ru](#) есть парочка патчиков которые меняют текущую локаль в JVM на `Locale.US`, но самый лучший способ — перейти на JDK 1.3.1, в котором данная бага пофиксена.

Начинающие пользователи JBuilder могут также встретиться с такой проблемой — русские буквы сохраняются в виде кодов "\uXXXX". Чтобы этого избежать, надо в диалоге `Default Project Properties`, закладка `General`, в поле `Encoding` поменять `Default` на `Cp1251`.

Если Вы используете для компиляции не стандартный `javac`, а другой компилятор - обратите внимание на то, как он выполняет преобразование символов. Например, некоторые версии IBM-овского компилятора `jikes` не понимают, что бывают кодировки, отличные от ISO-8859-1 :-). Существуют версии, пропатченные на этот счёт,

но часто там тоже зашивается некоторая кодировка — нет такого удобства, как в `javac`.

6. JavaDoc

Для генерации HTML-документации по исходникам используется утилита `javadoc`, входящая в стандартную поставку JDK. Для указания кодировок у неё есть аж 3 параметра:

- `-encoding` — эта настройка задаёт кодировку исходников. По умолчанию используется `file.encoding`.
- `-docencoding` — эта настройка задаёт кодировку генерируемых HTML-файлов. По умолчанию используется `file.encoding`.
- `-charset` — эта настройка задаёт кодировку, которая будет записываться в заголовки генерируемых HTML-файлов (`<meta http-equiv="Content-Type" content="text/html; charset=...">`). Очевидно, что она должна совпадать с предыдущей настройкой. Если данная настройка опущена, то тег `meta` добавляться не будет.

7. Русские буквы в файлах `properties`

Для чтения файлов `properties` используются методы загрузки ресурсов, которые работают специфичным образом. Собственно для чтения используется метод `Properties.load`, который не использует `file.encoding` (там в исходниках жёстко указана кодировка ISO-8859-1), поэтому единственный способ указать русские буквы — использовать формат `"\uXXXX"` и утилиту `native2ascii`.

Метод `Properties.save` работает по разному в версиях JDK 1.1 и 1.2. В версиях 1.1 он просто отбрасывал старший байт, поэтому правильно работал только с английскими буквами. В 1.2 делается обратное преобразование в `"\uXXXX"`, так что он работает зеркально к методу `load`.

Если файлы `properties` у Вас загружаются не как ресурсы, а как обычные файлы конфигурации, и Вас не устраивает такое поведение — выход один, написать собственный загрузчик.

8. Русские буквы в `Servlet`-ах.

Ну, для чего эти самые Servlet-ы нужны, я думаю, Вы в курсе. Если нет — то лучше сначала прочитать документацию. Здесь же рассказывается только об особенностях работы с русскими буквами.

Так в чём же особенности? Когда Servlet посылает ответ клиенту, есть два способа послать этот ответ — через `OutputStream` (метод `getOutputStream()`) или через `PrintWriter` (метод `getWriter()`). В первом случае Вы записываете массивы байтов, поэтому применимы вышеописанные методы записи в потоки. В случае же `PrintWriter`, он использует установленную кодировку. В любом случае необходимо правильно указать используемую кодировку при вызове метода `setContentType()`, для того, чтобы было правильное преобразование символов на стороне сервера. Это указание должно быть сделано перед вызовом `getWriter()` или перед первой записью в `OutputStream`.
Пример:

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    // Установка кодировки ответа
    // Учтите, что некоторые engine не допускают
    // наличие пробела между ';' и 'charset'
    response.setContentType("text/html; charset=Windows-1251");

    PrintWriter out = response.getWriter();

    // Отладочный вывод названия кодировки для проверки
    out.println("Encoding: " + response.getCharacterEncoding());
    ...
    out.close();
}
```

Это по поводу отдачи ответов клиенту. Со входными параметрами, к сожалению не так просто. Входные параметры кодируются браузером побайтно в соответствии с MIME-типом "application/x-www-form-urlencoded". Как рассказал [Алексей Менделев](#) русские буквы браузеры кодируют, используя текущую установленную кодировку. Ну и, разумеется, ничего о ней не сообщают. Соответственно, например, в JSDK версий от 2.0 до 2.2 это никак не проверяется, а то, что за кодировка будет использована для преобразования — зависит от используемого engine. Начиная со спецификации 2.3

Java: Русские буквы и не только...

появилась возможность устанавливать кодировку для `javax.servlet.HttpServletRequest` — метод `setCharacterEncoding()`. Эту спецификацию уже поддерживают последние версии [Resin](#) и [Tomcat](#).

Таким образом, если Вам повезло, и у Вас стоит сервер с поддержкой Servlet 2.3, то всё довольно просто:

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    // Кодировка сообщений
    request.setCharacterEncoding("Cp1251");

    String value = request.getParameter("value");
    ...
}
```

В применении метода `request.setCharacterEncoding()` есть одна существенная тонкость — он должен быть применен **до** первого обращения к запросу за данными (например `request.getParameter()`). Если Вы используете фильтры, которые обрабатывают запрос до того как он приходит в сервлет, есть ненулевая вероятность того, что в одном из фильтров может произойти чтение какого-нибудь параметра из запроса (например для авторизации) и `request.setCharacterEncoding()` в сервлете не сработает.

Потому идеологически более правильно написать фильтр, устанавливающий кодировку запроса. При этом он должен стоять первым в цепочке фильтров в `web.xml`.

Пример такого фильтра:

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

public class CharsetFilter implements Filter
{
    // кодировка
    private String encoding;

    public void init(FilterConfig config) throws ServletException
    {
        encoding = config.getInitParameter("encoding");
    }

    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws ServletException, IOException
    {
        request.setCharacterEncoding(encoding);
        chain.doFilter(request, response);
    }
}
```

```
{
    // читаем из конфигурации
    encoding = config.getInitParameter("requestEncoding");

    // если не установлена — устанавливаем Cp1251
    if( encoding==null ) encoding="Cp1251";
}

public void doFilter(ServletRequest request,
                    ServletResponse response, FilterChain next)
throws IOException, ServletException
{
    request.setCharacterEncoding(encoding);
    next.doFilter(request, response);
}

public void destroy(){}
}
```

И его конфигурации в web.xml:

```
<!--CharsetFilter start-->

<filter>
  <filter-name>Charset Filter</filter-name>
  <filter-class>CharsetFilter</filter-class>
  <init-param>
    <param-name>requestEncoding</param-name>
    <param-value>Cp1251</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>Charset Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<!--CharsetFilter end-->
```

Если же Вам не повезло, и у Вас более старая версия — для достижения результата придётся поизвращаться:

Java: Русские буквы и не только...

- Оригинальный способ работы с кодировками предлагает Russian Apache — [здесь](#) расписано, как именно.
- [Своё](#) решение проблемы так же предложил Вячеслав Педак.
- Ну а самый простейший вариант извлечь такие символы — передавать в комплекте параметров имя кодировки (или, если вы уверены в текущей кодировке браузера, использовать предопределённую кодировку) и использовать метод перекодировки символов:

```
public void doPost(
    HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    // Кодировка сообщений, использованная engine
    // Некоторые используют ISO-8859-1, некоторые кодировку
    // по умолчанию — единообразия тут нет
    String requestEnc = "ISO-8859-1";

    // Кодировка, установленная в браузере
    String clientEnc = request.getParameter("charset");

    if( clientEnc==null ) clientEnc="Cp1251";

    // Получение параметра
    String value = request.getParameter("value");

    //
    if( value!=null )
        value = new String(value.getBytes(requestEnc), clientEnc);
    ...
}
```

9. JSP

Технология JSP (JavaServer Pages) очень похожа на сервлеты. По сути дела сервер, при запросе в первый раз на лету генерит из jsp-страниц код сервлета, компилирует его и запускает его как обычный сервлет. Поэтому у JSP возникают схожие проблемы при работе с русскими буквами. Однако решаются они немного по другому. Есть три места где могут возникнуть трудности — русские буквы внутри самой jsp-страницы, в ответе

клиенту и в запросе от клиента. Первые два решаются заданием в начале страницы тега page:

```
<%@page contentType="text/html;charset=Windows-1251" %>
```

Увидев эту директиву сервер понимает, что страница записана в указанной кодировке, и что в сгенерённый код надо добавить вызов `response.setContentType()` с указанным `contentType`. Если сервер поддерживает спецификацию Servlet 2.3, то он также добавит и вызов `request.setCharacterEncoding()` с нужной кодировкой, таким образом автоматом решая и третью проблему. Для более старых серверов для раскодирования параметров в запросе клиента надо применять ухищрения, аналогичные описанным в разделе по сервлетам.

Для примера, для того, чтобы настроить JSP-форум [Jive](#) для работы с русскими буквами надо откорректировать следующие файлы:

```
/jive/header.jsp  
/jive/admin/header.jsp
```

В них надо в начало добавить строчку

```
<%@page contentType="text/html;charset=UTF-8" %>
```

Вместо UTF-8 можно использовать любую кодировку, поддерживающую русские буквы — всё зависит от вкусов и предпочтений. Информацию об этом прислал [Алексей Епишкин](#), за что ему отдельное спасибо.

В некоторых серверах встречаются баги, связанные с русскими буквами в JSP. Например, сервер Ogiон не любит русскую букву "Т" — он вместо неё в сгенерённый сервлет подставляет символ кавычки. Там во внутренностях есть примерно такой код:

```
...  
switch( charstring.c1(i) )  
...  
public final char c1(int i)  
{  
    if(i < 0 || i >= length)  
        throw new StringIndexOutOfBoundsException(i);  
}
```

```
else  
    return (char)(data[offset + i] & 0xff);
```

`data` — это массив типа `char[]`. Как видно, ошибка тут тривиальна — разработчик почему-то был уверен что символы с кодами больше 255 — это ошибка природы. :-)

10. JavaMail

Пакет `JavaMail` предназначен для работы с электронными письмами. При помощи этого пакета Вы можете отправлять и принимать письма через различные протоколы. Разные протоколы по разному обрабатывают национальные символы. Самые распространённые на данный момент протоколы Internet основаны на старом стандарте RFC-822. Согласно этому стандарту в служебных полях (заголовках) писем разрешено посылать только символы кодировки ASCII, т.е. только латинские буквы (первые 128 символов Unicode). Очевидно, что это неудобно, т.к. часто очень хочется писать, например в поле `Subject` (тема письма) или в полях `From/To` (имя и адрес отправителя/получателя) русский текст. Для того, чтобы решить эту проблему был придуман стандарт кодирования MIME (RFC 2047). Он позволяет в некоторых полях заголовка (не во всех) использовать национальные символы при помощи специального кодирования (`Base64` или `QuotedPrintable`).

Для представления писем в `JavaMail` используется класс `javax.mail.Message`. Это абстрактный класс, реальное же поведение определяется наследниками. Методы, определённые в нём работают только с обычными Java-строками (`String`). Для протоколов Internet обычно используется наследник `javax.mail.internet.MimeMessage`, который помимо базовых методов добавляет методы, в которых можно дополнительно указывать кодировку, которую следует использовать для писем. Для кодирования используется вспомогательный класс `javax.mail.internet.MimeUtility`. Класс `MimeMessage` обычно сам обращается к нему для кодирования/раскодирования заголовков, но, если Вы напрямую обращаетесь к заголовкам (методы `getHeader()/setHeader()/addHeader()`), то для их кодирования/раскодирования Вам придётся обращаться к методам `MimeUtility` самому.

Если Вы не указываете кодировку письма, то будет использована кодировка по умолчанию - обычно используется `file.encoding`, но её можно перекрыть специальной системной настройкой `"mail.mime.charset"`. Это разумно, т.к. часто кодировка по

умолчанию в системе отличается от стандартной кодировки Internet. Для русскоязычных писем в Internet стандартом де-факто стала кодировка КОИ-8. Вы, конечно, можете указать и другую, но шанс, что принимающая сторона не сможет прочитать такое письмо очень велик.

Надо учитывать также, что в JavaMail различаются два стандарта наименования кодировок — стандарт MIME и стандарт Java. Для большинства кодировок имена MIME уже поддерживаются в Java при помощи механизма синонимов. Например, для кодировки "Cp1251" (название Java) существует синоним "Windows-1251" (название MIME). Для тех кодировок, для которых такие синонимы отсутствуют, они поддерживаются внутри JavaMail. Для этого загружается файл `javamail.charset.map` из подкаталога `/META-INF` из того `jar`-файла, откуда был загружен пакет JavaMail. Для указания кодировки при вызове методов JavaMail следует использовать только MIME-имена, в противном случае получатель не сможет распознать использованную кодировку (если только на другом конце не тоже Java :-).

Вот простой пример отправки письма при помощи JavaMail:

```
import java.util.Properties;

import javax.mail.Session;
import javax.mail.Message;
import javax.mail.Transport;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.InternetAddress;

public class MailTest
{
    static final String ENCODING = "koi8-r";
    static final String FROM = "myaccount@mydomail.ru";
    static final String TO = "myaccount@mydomail.ru";

    public static void main(String args[]) throws Exception
    {
        Properties mailProps = new Properties();

        mailProps.put("mail.store.protocol", "pop3");
        mailProps.put("mail.transport.protocol", "smtp");
    }
}
```

Java: Русские буквы и не только...

```
mailProps.put("mail.user", "myaccount");

mailProps.put("mail.pop3.host", "mail.mydomain.ru");
mailProps.put("mail.smtp.host", "mail.mydomain.ru");

Session session = Session.getDefaultInstance(mailProps);

MimeMessage message = new MimeMessage(session);
message.setFrom(new InternetAddress(FROM));
message.setRecipient(Message.RecipientType.TO, new InternetAddress(TO));

message.setSubject("Тестовое письмо", ENCODING);
message.setText("Текст тестового письма", ENCODING);

Transport.send(message);
}
}
```

11. XML/XSL

При разработке формата XML особое внимание уделялось поддержке различных кодировок символов. Для указания того, какая кодировка была использована используется заголовок XML-документа. Пример:

```
<?xml version="1.0" encoding="Windows-1251"?>
```

Если кодировка указана не была, то по умолчанию предполагается кодировка UTF-8. На XML-парсер возложена обязанность корректно прочитать заголовок и использовать соответствующую кодировку для получения Unicode-символов. Разные парсеры могут поддерживать разные наборы кодировок, но UTF-8 обязаны поддерживать все. Здесь также, как и в случае с JavaMail наименования кодировок, описанные в стандарте XML могут расходиться с наименованиями, принятыми в Java. Разные парсеры по разному выходят из положения. Crimson просто использует некоторое кол-во дополнительных синонимов, а в остальном полагается на синонимы кодировок из Java. Xerces же по умолчанию использует внутреннюю таблицу (класс `org.apache.xerces.readers.MIME2Java`), а если не находит там кодировку, то бросает исключение о неподдерживаемой кодировке. В Xerces версии 1.4.0 русских кодировок там всего две — KOI8-R и ISO-8859-5. Однако это поведение по умолчанию можно

изменить при помощи разрешения у парсера специального feature "http://apache.org/xml/features/allow-java-encodings". Если этот feature разрешён (при помощи метода `setFeature()`), то парсер после поиска в таблице будет пытаться использовать стандартный Java-вский механизм и соответственно Java-вский набор кодировок. В случае использования интерфейса SAX сделать это можно таким, например, образом (при использовании JAXP):

```
SAXParserFactory parserFactory = SAXParserFactory.newInstance();  
  
SAXParser parser = parserFactory.newSAXParser();  
parser.getXMLReader().setFeature(  
    "http://apache.org/xml/features/allow-java-encodings", true);
```

Для DOM, к сожалению, подобного механизма feature-ов не предусмотрено, но можно вместо JAXP для создания DOM напрямую использовать класс `org.apache.xerces.parsers.DOMParser`, у которого уже есть метод `setFeature()`.

Если же Xerces используется не напрямую, а посредством другого пакета, то необходимо настроить этот пакет дабы он сам выставлял этот feature. Если же такой возможности не предусмотрено, то остаётся только один выход — править ручками. Для этого можно или подправить список кодировок в классе `org.apache.xerces.readers.MIME2Java` или установить указанный feature как true по умолчанию.

Для чтения документа XML из потока данных обычно используется класс `org.xml.sax.InputSource`. Собственно сам поток может быть представлен или в виде байтового потока (`java.io.InputStream`) или в виде потока символов (`java.io.Reader`). Соответственно ответственность за корректное распознавание кодировки возлагается или на парсер или на того, кто создаёт объект `Reader`. У класса `InputSource` есть так же метод `setEncoding()`, при помощи которого можно явно задать кодировку в случае использования потока байтов.

Работает это всё таким образом:

- Если был задан поток символов (`Reader`), то он будет использован для чтения данных. Кодировка, установленная методом `setEncoding()` при этом игнорируется, как игнорируется и кодировка, указанная в заголовке XML-документа.

Java: Русские буквы и не только...

- Если вместо потока символов был задан поток байтов (`InputStream`), то используется он. Если установлена кодировка методом `setEncoding()`, то используется она, а если нет — то парсер использует кодировку, указанную в заголовке XML-документа.

Если при чтении заголовка XML-документа обнаруживается расхождение между заданной кодировкой и кодировкой из заголовка, то парсеры могут поступать по-разному. `Crimson`, например, при этом выдаёт предупреждение, а `Xerces` молча пропускает.

С чтением XML-документов мы разобрались, теперь перейдём к их созданию. Единого стандарта на создание документов, в отличие от чтения, пока нет. Предполагается, что следующая версия рекомендаций комитета W3C будет включать в себя и создание документов, но пока что создатели парсеров делают кто во что горазд.

В случае с `Crimson` сохранить созданный документ DOM можно при помощи метода `write()` у класса `org.apache.crimson.tree.XmlDocument`. В качестве аргумента можно передать или поток символов (`Writer`) или поток байтов (`OutputStream`). Вместе с потоком можно передать и необходимую кодировку. Если использован поток байтов, а кодировка указана не была, то используется UTF-8. Если использован поток символов вместе с именем кодировки, то имя используется только для записи в заголовок документа. Если `Writer` передан без кодировки, то делается проверка — если это экземпляр `OutputStreamWriter`, то для выяснения что писать в заголовок зовётся его метод `getEncoding()`. Если же это другой `Writer`, то кодировка в заголовок записана не будет, что по стандарту означает кодировку UTF-8. Пример:

```
XmlDocument doc = ...;

OutputStream os = ...;

doc.write(os, "Windows-1251");
```

В `Xerces` для создания документов используются классы из пакета `org.apache.xml.serialize`. Собственно для записи используется класс `XMLSerializer`, а для настройки выходного формата — класс `OutputFormat`. В конструкторе `XMLSerializer` можно передавать как потоки байтов, так и потоки символов. В случае потоков символов используемая кодировка должна совпадать с заданной в

OutputStream. Важно не забыть задать используемую кодировку в OutputStream — в противном случае русские буквы будут представлены в виде кодов, типа такого: "АБВ" для символов "АБВ". Пример:

```
OutputStream os = ...;

OutputStream format = new OutputStream(
    Method.XML, "Windows-1251", true )

XMLSerializer serializer =
    new XMLSerializer(os, format);

serializer.serialize(doc);
```

11.1. Castor XML

Пакет [Castor](#) предназначен для решения проблем долговременного хранения объектов. В числе прочего он содержит в себе подсистему Castor XML, которая по сути дела является надстройкой над XML-парсером и позволяет автоматизировать чтение и запись XML-файлов. Castor XML по умолчанию использует парсер Xerces, поэтому проблемы Xerces переключаются и сюда. В документации к Castor в примерах используются потоки символов (Reader и Writer), а это может привести к рассогласованности между используемой в потоке кодировки и реальной кодировки XML-файла. Как уже говорилось выше, чтобы прочитать при помощи Xerces XML-файл в произвольной кодировке нужно, во первых, использовать потоки байтов, а во вторых, установить специальный feature. К счастью эта возможность предусмотрена в Castor. Для этого нужно скопировать файл castor.properties (взять его можно из каталога org\exolab\castor в файле castor-0.9.3-xml.jar) в подкаталог lib в JRE, и установить там переменную **org.exolab.castor.sax.features**. Пример:

```
# Comma separated list of SAX 2 features that should be enabled
# for the default parser.
#
#org.exolab.castor.features=
org.exolab.castor.sax.features=
http://apache.org/xml/features/allow-java-encodings
```

Java: Русские буквы и не только...

Стоит отметить, что по умолчанию там стоит переменная **org.exolab.castor.features**, но это, очевидно, опечатка — если посмотреть в исходники, то там анализируется **org.exolab.castor.sax.features** (это справедливо для Castor версии 0.9.3 от 03.07.2001). Пример чтения с использованием потоков байтов:

```
public static Object load(
    Class cls, String mappingFile, InputStream is)
    throws Exception
{
    Mapping mapping = loadMapping(cls, mappingFile);

    Unmarshaller unmarshaller = new Unmarshaller(cls);
    unmarshaller.setMapping(mapping);

    return unmarshaller.unmarshal(new InputSource(is));
}
```

Для создания XML-файлов необходимо правильно указать формат для Xerces. Пример:

```
public static void save(
    Object obj, String mappingFile,
    OutputStream os, String encoding)
    throws Exception
{
    Mapping mapping = loadMapping(obj.getClass(), mappingFile);

    try
    {
        XMLSerializer serializer = new XMLSerializer(
            os, new OutputFormat( Method.XML, encoding, true ));

        Marshaller marshaller = new Marshaller(serializer);
        marshaller.setMapping(mapping);

        marshaller.marshal(obj);
    }
    finally { os.flush(); }
}
```

Для загрузки файлов маппинга в этих примерах можно использовать такой код:

```
private static Mapping loadMapping(
    Class cls,String mappingFile)
    throws Exception
{
    ClassLoader loader = cls.getClassLoader();

    Mapping mapping = new Mapping(loader);
    mapping.loadMapping( new InputSource(
        loader.getResourceAsStream(mappingFile)) );

    return mapping;
}
```

11.2. XSL

Спецификация [XSL](#) описывает стандарт на преобразование XML-документов. Когда при помощи XSL выполняется преобразование из одного XML-документа в другой, особых причин для беспокойства нет — и тот и другой являются Unicode-документами, поэтому нет преобразований из символов в байты и обратно, могущих повлиять на результат. Другое дело, когда выполняется преобразование из XML в HTML или вообще в текстовый файл. Формат выходного файла задаётся настройкой тега [xsl:output](#), в котором можно задать используемую кодировку. Пример:

```
<xsl:output encoding="Windows-1251" method='html' indent='yes' />
```

Если XSLT-процессор не знает указанной кодировки, то он должен или выдать ошибку или использовать UTF-8 (или UTF-16). Если формируется HTML, то XSLT-процессор должен добавить тег `meta`, в котором будет указана реально использованная кодировка:

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

Всё бы хорошо, но некоторые XSLT-процессоры не поддерживают данный тег (по спецификации они и не обязаны). В частности пакет [Cocoon](#) его не поддерживает, т.к. [по словам](#) разработчиков он противоречит внутренней архитектуре этого пакета. Вместо этого там поддерживается указание выходного формата при помощи инструкции препроцессора `socoop-format`. Пример вставки этой инструкции в XSL:

Java: Русские буквы и не только...

```
<xsl:processing-instruction name="cocoon-format">
  type="text/html"
</xsl:processing-instruction>
```

Таким образом можно динамически менять выходной формат. Если это не требуется, то можно записать инструкцию и статически (в исходном XML-документе):

```
<?cocoon-format type="text/html"?>
```

Собственно используемая кодировка настраивается для каждого формата отдельно в файле `cocoon.properties`.

Новая версия [Cocoon 2.0](#) кроме управления кодировками позволяет сделать в плане локализации уже гораздо больше. Подробности Вы можете узнать на их [сайте](#).

В случае использования JAXP для генерации выходного потока (пакет `javax.xml.transform`) кроме использования тега `xsl:output` можно использовать методы `setOutputProperty` объекта `Transformer`. Пример сохранения документа в нужной кодировке:

```
TransformerFactory trFactory = TransformerFactory.newInstance();
Transformer transformer = trFactory.newTransformer();

transformer.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, docPublic);
transformer.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, docSystem);

transformer.setOutputProperty( OutputKeys.INDENT, "yes" );
transformer.setOutputProperty( OutputKeys.ENCODING, encoding );

OutputStream os = ...;

StreamResult result = new StreamResult( os );

transformer.transform( source, result );
```

Тут есть один подводный камень — реализация `Transformer` должна поддерживать нужную кодировку. Халан из состава JDK 1.4.0_x и 1.4.1_x поддерживает только две русские кодировки — KOI8-R и ISO-8859-5. Если хочется использовать Windows-1251, то можно воспользоваться механизмом `endorsed`:

1. Создаете каталог %JAVA_HOME%\jre\lib\endorsed
2. Копируете туда jar с пропатченным классом: [XalanRusChars.jar](#)

В JDK 1.4.2 Beta включена новая версия Xalan, которая вроде как уже поддерживает кодировку 1251.

12. FOP

Пакет [FOP](#) предназначен для обработки документов по стандарту XSL FO (Formating Objects). В частности он позволяет создавать PDF-документы на базе документов XML. Для преобразования из исходного XML в FO пакет FOP по умолчанию использует XSLT-процессор [Xalan](#) в паре с Xerces. Для создания итогового изображения в FOP необходимо подключить шрифты, поддерживающие русские буквы. Вот как можно проделать это для версии 0.20.1:

1. В подкаталог conf/fonts (например, в c:\fop-0.20.1\conf\fonts\) скопировать файлы ttf из системного каталога Windows. Для Arial normal/normal, normal/bold, italic/normal и italic/bold нужны файлы arial.ttf, arialbd.ttf, ariali.ttf и arialbi.ttf.
2. Сгенерировать файлы описаний шрифтов (типа arial.xml). Для этого для каждого шрифта нужно выполнить команду (это для Arial normal/normal, всё в одну строку):

```
java -cp .;c:\fop-0.20.1\build\fop.jar;c:\fop-0.20.1\lib\batik.jar;  
c:\fop-0.20.1\lib\xalan-2.0.0.jar;c:\fop-0.20.1\lib\xerces.jar;  
c:\fop-0.20.1\lib\jimi-1.0.jar  
org.apache.fop.fonts.apps.TTFReader fonts\arial.ttf fonts\arial.xml
```

3. В FOP добавить в conf/userconfig.xml описание шрифта с русскими буквами, типа:

```
<font metrics-file="c:\fop-0.20.1\conf\fonts\arial.xml" kerning="yes"  
  embed-file="c:\fop-0.20.1\conf\fonts\arial.ttf">  
  <font-triplet name="Arial" style="normal" weight="normal"/>  
  <font-triplet name="ArialMT" style="normal" weight="normal"/>  
</font>
```

Аналогично добавляются Arial normal/bold, italic/normal и italic/bold.

4. При вызове FOP из командной строки после org.apache.fop.apps.Fop писать -c c:\fop-0.20.1\conf\userconfig.xml Если нужно использовать FOP из сервлета, то нужно в сервлете после строчки

```
Driver driver = new Driver();
```

Java: Русские буквы и не только...

добавить строчки:

```
// Каталог fonts (c:\weblogic\fonts) был создан исключительно для удобства.  
String userConfig = "fonts/userconfig.xml";  
File userConfigFile = new File(userConfig);  
Options options = new Options(userConfigFile);
```

Тогда расположение файлов ttf в файле userconfig.xml можно указать относительно корня сервера приложения, без указания абсолютного пути:

```
<font metrics-file="fonts/arial.xml" kerning="yes"  
embed-file="fonts/arial.ttf">  
  <font-triplet name="Arial" style="normal" weight="normal"/>  
  <font-triplet name="ArialMT" style="normal" weight="normal"/>  
</font>
```

5. В файле FO (или XML и XSL) перед использованием шрифта писать:

```
font-family="Arial"  
font-weight="bold" (Если используется Arial bold)  
font-style="italic" (Если используется Arial italic)
```

Данный алгоритм прислал [Алексей Тюрин](#), за что ему отдельное спасибо.

Если Вы используете встроенный в FOP просмотрщик, то необходимо учесть его особенности. В частности, хотя предполагается, что надписи в нём русифицированы, на самом деле сделано это с ошибкой (в версии 0.19.0). Для загрузки надписей из файлов ресурсов в пакете org.apache.fop.viewer.resources используется собственный загрузчик (класс org.apache.fop.viewer.LoadableProperties). Кодировка чтения там жёстко зафиксирована (8859_1, как и в случае Properties.load()), однако поддержка записи вида "\uXXXX" не реализована. Я сообщил об этой ошибке разработчикам, они включили её исправление в свои планы.

Кроме всего прочего существует сайт посвящённый русификации FOP (<http://www.openmechanics.net/rusfop/>) Там Вы сможете найти дистрибутив FOP с уже исправленными ошибками и подключенными русскими шрифтами.

13. POI

Пакет [Jakarta POI](#) предназначен для работы с документами Microsoft Office. Пока что

более-менее работающей там является только поддержка файлов MS Excel (xls). Особой сложности в работе с русским языком нет, но надо учитывать нюанс, что для работы с ячейкой используется класс `org.apache.poi.hssf.usermodel.Cell`, у которого есть метод `setEncoding(short encoding)`, однако вместо привычных "Cp1255" и "Cp866", необходимо использовать константы `ENCODING_COMPRESSED_UNICODE` (0) и `ENCODING_UTF_16` (1). По умолчанию включен первый режим, а для нормальной работы с русским языком необходимо использовать `ENCODING_UTF_16`. Причем что самое важное, эту установку необходимо выполнять для каждой, создаваемой ячейки. Пример кода:

```
HSSFWorkbook wb = new HSSFWorkbook();
HSSFSheet sheet = wb.createSheet("Sheet1");
HSSFRow row = sheet.createRow( (short)0 );

for( int i = 0; i < 10; i++ )
{
    HSSFCell cell = row.createCell( (short)i );
    cell.setEncoding( (short)cell.ENCODING_UTF_16 );
    cell.setCellValue("Тест русского языка");
}
```

Создать лист с названием содержащим русские символы, к сожалению, не удаётся. Данное описание прислал [Вячеслав Яковенко](#), за что ему отдельное спасибо.

14. CORBA

В стандарте CORBA предусмотрен тип, соответствующий Java-овскому типу `String`. Это тип `wstring`. Всё бы хорошо, но некоторые CORBA-сервера не поддерживают его в полной мере. Типичные исключения, возникающие при спотыкании на русских буквах: **org.omg.CORBA.MARSHAL: minor code 5 completed No** или **org.omg.CORBA.DATA_CONVERSION**. Лучше всего, конечно, заменить CORBA-сервер. К сожалению у меня нет статистики, поэтому я не могу сказать, с какими проблем не будет. Если сменить систему не представляется возможным, можно вместо типа `wstring` использовать тип `string` в паре с нашим любимым преобразованием:

Java: Русские буквы и не только...

```
// Серверная часть
a = new Answer(new String( src.getBytes("Cp1251"), "ISO-8859-1" ));

...

// Клиентская часть
Answer answer=serverRef.getAnswer();
res = new String( answer.msg.getBytes("ISO-8859-1"), "Cp1251" );
```

Тип `wstring` при этом лучше не использовать, потому как тем самым Вы кривость сервера будете компенсировать кривостью своих компонентов, а это практически всегда чревато разнообразными проблемами в будущем.

Вместо `Cp1251` можно использовать любую кодировку русских букв, по желанию. Это будет кодировка, в которой будут передаваться строки в компоненты на других языках. Также, аналогичный код может потребоваться, если необходимо организовать связь с готовыми не-Java компонентами, которые уже использовали тип `string`.

Честно говоря, не лежит у меня душа к таким решениям, ну да что поделаешь, иногда оно единственное.

15. JNI

JNI (Java Native Interface) — это стандарт по взаимодействию с C/C++-ным кодом. Как и следовало ожидать, на этом водоразделе тоже происходит столкновение байтов и символов. Большинство C/C++-ных программ пишется без учёта Unicode, многие программисты даже не знают о нём. Я сам, за 7 лет писательства на C/C++, пока не начал писать на Java, про Unicode знал только по наслышке. Большинство строковых операций в C/C++ сделаны для 8-битового сишного типа `char`. В принципе, есть некоторые подвижки в этом направлении, в частности для Windows NT можно откомпилировать код, который будет взаимодействовать с Unicode-вариантами Win32 API, но, к сожалению, этого часто недостаточно.

Таким образом главная задача — получить тип `char*` из типа `jstring` (JNI-шное отображение `String`) и наоборот. Практически во всех описаниях и примерах JNI для этого используется пара функций `GetStringUTFChars()/ReleaseStringUTFChars()`. Коварные буржуины и здесь приготовили засаду — эти функции формируют массив байтов по стандарту UTF, который соответствует ожидаемому только для

ASCII-символов (первых 128 значений). Русские буквы опять в пролёте. Сишные строки `char*` очень хорошо ложатся на Java-овский тип `byte[]`, но при этом возникает загвоздка в виде ноль-символа. Его нужно добавлять при преобразовании `byte[]->char*` и учитывать при обратном преобразовании. Пример:

```
public void action(String msg) throws java.io.IOException
{
    int res = nAction( msg );
    if( res!=0 ) throw new java.io.IOException( nGetErrorString(res) );
}

private native int nAction(String msg);
private native String nGetErrorString(int error);

...

jbyteArray getStringBytes(JNIEnv *env, jstring str)
{
    if( !str ) return NULL;

    jmethodID getBytes = env->GetMethodID(
        env->GetObjectClass(str), "getBytes", " () [B]");

    jbyteArray buf = (jbyteArray)env->CallObjectMethod(str, getBytes);

    if( !buf ) return NULL;

    // Добавляем ноль-символ
    jsize len = env->GetArrayLength(buf);

    jbyteArray nbuf = env->NewByteArray(len+1);

    if( len!=0 )
    {
        jbyte *cbuf = env->GetByteArrayElements(buf, NULL);

        env->SetByteArrayRegion(nbuf, 0, len, cbuf);

        env->ReleaseByteArrayElements(buf, cbuf, JNI_ABORT);
    }
}
```

Java: Русские буквы и не только...

```
    }

    env->DeleteLocalRef(buf);

    return nbuf;
}

JNIEXPORT jint JNICALL Java_Test_nAction
    (JNIEnv *env, jobject obj, jstring msg)
{
    jbyteArray bmsg = getStringBytes(env,msg);
    if( !bmsg ) return -1;

    jbyte *cmsg = env->GetByteArrayElements(bmsg,NULL);

    printf(cmsg);

    jint res = do_something(cmsg);

    env->ReleaseByteArrayElements(bmsg,cmsg,JNI_ABORT);

    return res;
}

jstring newString(JNIEnv *env, jbyteArray jbuf, int len)
{
    jclass stringClass = env->FindClass("java/lang/String");
    if( !stringClass ) return NULL;

    jmethodID init = env->GetMethodID(stringClass,"<init>","([BII)V");
    if( !init ) return NULL;

    return (jstring)env->NewObject(stringClass,init,jbuf,0,len);
}

jstring newString(JNIEnv *env, const char *buf)
{
    if( !buf ) return NULL;
}
```

```
int bufLen = strlen(buf);

if( bufLen==0 )
{
    return env->NewString( (const jchar *)L"", 0 );
}

jbyteArray jbuf = env->NewByteArray(bufLen);
if( !jbuf ) return NULL;

env->SetByteArrayRegion(jbuf,0,bufLen,(jbyte*)buf);

jstring jstr = newString(env,jbuf,bufLen);

env->DeleteLocalRef(jbuf);

return jstr;
}

JNIEXPORT jstring JNICALL Java_Test_nGetErrorString
(JNIEnv *env, jobject obj, jint error)
{
    char cmsg[256];
    memset(cmsg,0,sizeof(cmsg));

    get_error_string( error,cmsg,sizeof(cmsg) );

    return newString(env,cmsg);
}
```

Тут используется преобразование символов по умолчанию, что вполне естественно при взаимодействиях с системным API. Если же Вам необходима определённая кодовая страница, соответственно нужно добавить её название.

16. GUI (AWT, Swing)

Многие связывают неправильный вывод русских букв с неправильной установкой шрифта. На самом деле в Java всё сложнее и редко действительно связано со шрифтами.

Java: Русские буквы и не только...

Где же действительно лежат наибольшие подводные камни? В основном это связано с неправильной перекодировкой символов. Часть этих проблем и методы их решения описаны выше. Если у Вас все преобразования выполняются корректно, и для вывода используется шрифт Unicode, то есть очень большой шанс, что Ваша программа будет работать правильно.

Если проблемы всё же остались, тут нужно выяснить, где они возникают. Попробуйте запустить приложение под разными JVM, под разными платформами, на разных браузерах. Пример достаточно универсального алгоритма поиска проблем предложен ниже, в разделе Типичные ошибки.

Если программа не работает нигде — значит проблема только в ней и в Ваших руках. Внимательно перечитайте всё, что было написано выше, и ищите. Если же проблема проявляется только в конкретном окружении — значит дело, возможно в настройках. Где именно — зависит от того, какой графической библиотекой Вы пользуетесь. Если AWT — помочь может правильная настройка файла `font.properties.ru`. Пример корректного файла можно взять из Java 2. Если у Вас нет этой версии, можете скачать его с данного сайта: [версия для Windows](#), [версия для Linux](#) (см. также [раздел по Linux](#) ниже). Этот файл задаёт используемые шрифты и кодовые страницы. Если у Вас установлена русская версия OS — просто добавьте этот файл туда, где лежит файл `font.properties`. Если же это английская версия, то нужно, или переписать этот файл вместо `font.properties` или дополнительно сменить текущие региональные настройки на русские. Иногда может сработать настройка `-Duser.language=ru`, но чаще — нет. Тут примерно те же проблемы, что и с `file.encoding` — работает или нет, зависит от JDK (см. ошибку за номером [4152725](#)).

Если кроме русских букв Вам также надо выводить, к примеру, греческие, то обычно достаточно просто правильно указать их кода. Работает всё это примерно таким способом:

По умолчанию в AWT и Swing используются виртуальные шрифты, настраиваемые в `font.properties.ru` (`dialog`, `dialoginput` и т.д.). Эти шрифты виртуальные и при выводе, в зависимости от кода выводимого символа используется один из реальных шрифтов. Например, вот эти строчки:

```
dialog.0=Arial,RUSSIAN_CHARSET
```

```
dialog.1=WingDings, SYMBOL_CHARSET, NEED_CONVERTED  
dialog.2=Symbol, SYMBOL_CHARSET, NEED_CONVERTED
```

задают, что виртуальный шрифт dialog обычного начертания состоит из 3-х шрифтов (Arial, WingDings и Symbol). Далее, вот эти строчки:

```
fontcharset.dialog.0=sun.io.CharToByteCp1251  
fontcharset.dialog.1=sun.awt.windows.CharToByteWingDings  
fontcharset.dialog.2=sun.awt.CharToByteSymbol
```

задают, какие классы нужно использовать для перекодирования из Unicode в кодировку данного шрифта. При выводе символов сначала ищется, в каком шрифте определены выводимые символы. Это определяется тем, какие символы может конвертировать указанные классы. Есть так же дополнительная настройка (exclusion), которая явно задаёт диапазоны символов, которые неприменимы для данного шрифта. Например, вот эта строка

```
exclusion.dialog.0=0100-0400, 0460-ffff
```

задаёт, что при выводе символов с кодами от 0100 до 0400 и от 0460 до ffff шрифт 0 (Arial) использовать не следует. Эта строка нужна, в основном, для оптимизации.

Таким образом, при выводе греческих символов шрифт 0 (Arial) не подходит по exclusion, шрифт 1 (WingDings) не подходит, т.к. в таблице перекодировки CharToByteWingDings они отсутствуют поэтому используется шрифт 2 (Symbol), в котором есть греческие символы.

С библиотекой Swing всё проще — в ней всё рисуется через подсистему Java2D. Надписи в стандартных диалогах (JOptionPane, JFileChooser, JColorChooser) переделать на русский очень просто — достаточно лишь создать несколько файлов ресурсов. Я это уже проделал, так что можете просто взять готовый [файл](#) и добавить его в lib\ext или в CLASSPATH. Единственная проблема, с которой я столкнулся — в версиях JDK начиная с 1.2 rc1 и по 1.3 beta, русские буквы не выводятся под Win9x при использовании стандартных шрифтов (Arial, Courier New, Times New Roman, etc.) из-за ошибки в Java2D. Ошибка весьма своеобразна — со стандартными шрифтами изображения букв отображаются не в соответствии с кодами Unicode, а по таблице Cp1251 (кодировка Ansi). Эта ошибка зарегистрирована в BugParade под номером [4192443](#). По умолчанию в Swing используются шрифты, задаваемые в файле

Java: Русские буквы и не только...

font.properties.ru, так что достаточно заменить их другими — и русские буквы появляются. К сожалению, набор рабочих шрифтов небольшой — это шрифты Tahoma, Tahoma Bold и два набора шрифтов из дистрибутива JDK — Lucida Sans * и Lucida Typewriter * ([пример файла font.properties.ru](http://font.properties.ru)). Чем эти шрифты отличаются от стандартных — мне непонятно.

Начиная с версии 1.3rc1 эта проблема уже исправлена, так что нужно просто обновить JDK. JDK 1.2 уже сильно устарел, так что я не рекомендую им пользоваться. Так же надо учесть, что с оригинальной версией Win95 поставляются шрифты, не поддерживающие Unicode - в этой ситуации можно просто скопировать шрифты из Win98 или WinNT.

17. Типичные ошибки, или "куда делась буква Ш?"

17.1. Буква Ш.

Этот вопрос ("куда делась буква Ш?") довольно часто возникает у начинающих программистов на Java. Давайте разберёмся, куда же она действительно чаще всего девается. :-)

Вот типичная программа а-ля HelloWorld:

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("ЙЦУКЕНГШЩЗХЪ");
    }
}
```

в Far-е сохраняем данный код в файл Test.java, компилируем...

```
C:\>javac Test.java
```

и запускаем...

```
C:\>java Test
```

ЙЦУКЕНГ?ЩЗХЪ

Что же произошло? Куда делась буква Ш? Весь фокус здесь в том, что произошла взаимокompенсация двух ошибок. Текстовый редактор в Far по умолчанию создаёт файл в DOS-кодировке (Cp866). Компилятор же javac для чтения исходника использует file.encoding (если не указано иное ключиком -encoding). А в среде Windows с русскими региональными настройками кодировкой по умолчанию является Cp1251. Это первая ошибка. В результате, в скомпилированном файле Test.class символы имеют неверные коды. Вторая ошибка состоит в том, что для вывода используется стандартный PrintStream, который тоже использует настройку из file.encoding, однако консольное окно в Windows отображает символы, используя кодировку DOS. Если бы кодировка Cp1251 была взаимодназначной, то потери данных бы не было. Но символ Ш в Cp866 имеет код 152, который в Cp1251 не определён, и поэтому отображается на Unicode-символ 0xFFFD. Когда происходит обратное преобразование из char в byte, вместо него подставляется символ '?'.

На аналогичную компенсацию можно нарваться, если прочитать символы из текстового файла при помощи java.io.FileReader, а затем вывести их на экран через System.out.println(). Если файл был записан в кодировке Cp866, то вывод будет идти верно, за исключением опять же буквы Ш.

17.2. Прямая конверсия byte<->char.

Эта ошибка является любимой у зарубежных программистов на Java. Она довольно подробно рассмотрена в начале описания. Если Вы когда-нибудь будете смотреть чужие исходники, то всегда обращайтесь внимание на явную конверсию типов — (byte) или (char). Довольно часто в таких местах закопаны грабли.

17.3. Алгоритм поиска проблем с русскими буквами

Если Вы не представляете себе где в Вашей программе может происходить потеря русских букв, то можно попробовать следующий тест. Любую программу можно рассматривать как обработчик входных данных. Русские буквы — это такие же данные, они проходят в общем случае три стадии обработки: они откуда-то читаются в память программы (вход), обрабатываются внутри программы и выводятся пользователю (выход). Для того, чтобы определить место проблем, надо попробовать

вместо данных зашить в исходник такую тестовую строку: "АБВ\u0410\u0411\u0412", и попробовать её вывести. После этого смотрите, что у Вас вывелось:

- Если Вы увидите "АБВАБВ", значит компиляция исходников и вывод у Вас работают правильно.
- Если Вы увидите "???АБВ" (или любые другие символы кроме "АБВ" на месте первых трёх букв), значит вывод работает правильно, но вот компиляция исходников происходит неверно — скорее всего не указан ключик `-encoding`.
- Если Вы увидите "?????" (или любые другие символы кроме "АБВ" на месте второй тройки букв), значит вывод у Вас работает неверно.

Настроив вывод и компиляцию уже можно легко разобраться и со входом. После настройки всей цепочки проблемы должны исчезнуть.

18. Об утилите `native2ascii`

Эта утилита входит в состав Sun JDK и предназначена для преобразования исходных текстов к ASCII-виду. Она читает входной файл, используя указанную кодировку, а на выходе записывает символы в формате "`\uXXXX`". Если указать ключик `-reverse`, то выполняется обратная конвертация. Эта программа очень полезна для конвертации файлов ресурсов (`.properties`) или для обработки исходников, если Вы предполагаете, что они могут компилироваться на компьютерах с отличными от русских региональными настройками.

Если запустить программу без параметров, она работает со стандартным входом (`stdin`), а не выводит подсказку по ключам, как остальные утилиты. Это приводит к тому, что многие и не догадываются о необходимости указания параметров (кроме, может быть, тех, кто нашёл в себе силы и мужество заглянуть таки в документацию :-). Между тем этой утилите для правильной работы необходимо, как минимум, указать используемую кодировку (ключик `-encoding`). Если этого не сделать, то будет использована кодировка по умолчанию (`file.encoding`), что может несколько расходиться с ожидаемой. В результате, получив неверные коды букв (из-за неверной кодировки) можно потратить весьма много времени на поиск ошибок в абсолютно верном коде.

19. О методе перекодировки символов

Этот метод многие используют неправильно, наверное, не совсем понимая его суть и ограничения. Он предназначен для восстановления верных кодов букв, если они были неверно проинтерпретированы. Суть метода проста: из полученных неверных символов, используя соответствующую кодовую страницу, восстанавливается исходный массив байтов. Затем из этого массива байтов, используя уже корректную страницу, получаются нормальные коды символов. Пример:

```
String res = new String( src.getBytes("ISO-8859-1"), "Cp1251" );
```

Проблем в использовании этого приёма может быть несколько. Например, для восстановления используется неверная страница, или же она может измениться в некоторых ситуациях. Другая проблема может быть в том, что некоторые страницы выполняют неоднозначное преобразование byte <-> char. Смотрите, например, описание ошибки за номером [4296969](#).

Поэтому пользоваться этим методом стоит только в самом крайнем случае, когда уже ничто другое не помогает, и Вы чётко себе представляете, где именно происходит неверное преобразование символов.

20. Русские буквы и MS JVM

Непонятно по каким соображениям, но в ней отсутствуют все файлы кодировок русских букв, кроме Cp1251 (наверное, они таким образом пытались уменьшить размер дистрибутива). Если Вам нужны другие кодировки, например, Cp866, то нужно добавить соответствующие классы в CLASSPATH. Причём классы от последних версий Sun JDK не подходят — у Sun-а уже давно изменилась их структура, поэтому последние версии классов с Microsoft-ом не стыкуются (у MS осталась структура от JDK 1.1.4). На сервере Microsoft, в принципе, лежит полный комплект дополнительных кодировок, но там файл размером около 3 метров, а их сервер докачку не поддерживает :-). Мне удалось таки выкачать этот файл, я его переупаковал jar-ом, можете взять его [отсюда](#) .

Если же Вы пишете апплет, который должен работать под MS JVM и Вам потребовалось прочитать откуда-то (например из файла на сервере) байты в русской кодировке, отличной от Cp1251 (например, в Cp866), то стандартный механизм кодировок Вы уже не сможете использовать — апплетам запрещено добавлять классы

в системные пакеты, коим в данном случае является пакет `sun.io`. Выхода тут два — или перекодировать файл на сервере в Cp1251 (или в UTF-8) или перед преобразованием из байтов в Unicode выполнять конвертацию байтов из нужной кодировки в Cp1251.

21. Руссификация Java под Linux

Скажу сразу — я с Linux не работаю, а приведённая здесь информация получена от читателей данного описания. Если Вы найдёте неточность или захотите дополнить — напишите мне.

При кириллизации JVM в Linux существует две параллельных проблемы:

1. Проблема вывода кириллицы в GUI-компонентах
2. Проблема ввода кириллицы с клавиатуры (в X11)

Проблему вывода можно решить таким способом (данный алгоритм прислал [Artemy E. Kapitula](#)):

1. Установить в X11 нормальные шрифты ttf из Windows NT/200. Я бы рекомендовал Arial, Times New Roman, Courier New, Verdana и Tahoma — причем подключать их лучше не через сервер шрифтов, а как каталог с файлами.
2. Добавить следующий файл [font.properties.ru](#) в каталог `$JAVA_HOME/jre/lib`

Проблема ввода решается примерно таким способом (данный алгоритм прислал [Михаил Иванов](#)):

Настройка ввода русских букв в следующей конфигурации:

- Mandrake Linux 7.1
- XFree86 3.3.6
- IBM Java 1.3.0 (релизная)

Проблема:

IBM Java 1.3 не даёт вводить русские буквы (видны как крокозябры) при том что на лейблах и в менюшках их видно.

Причина:

использование XIM (-> xkb) в AWT. (это не есть плохо само по себе, просто с такими

штуками нужно обращаться осторожно + некоторые прилады xkb не любят).

Решение:

Настроить xkb (и локаль (xkb без локали НЕ РАБОТАЕТ))

Процедура:

1. выставляется локаль (где-нибудь типа в /etc/profile или в ~/.bash_profile)

```
export LANG=ru_RU.KOI8-R
export LC_ALL=ru_RU.KOI8-R
```

2. правится (если это еще не сделано) /etc/X11/XF86Config. В секции Keyboard должно быть примерно следующее:

```
XkbKeycodes      "xfree86"
XkbTypes         "default"
XkbCompat        "default"
XkbSymbols       "ru"
XkbGeometry      "pc"
XkbRules         "xfree86"
XkbModel         "pc101"
XkbLayout        "ru"
XkbOptions       "grp:shift_toggle" # переключение 2-мя шифтами
#XkbOptions      "grp:caps_toggle"  # переключение caps-lock'ом
```

примечание: такая настройка xkb не совместима с xgus (и ему подобными типа kikbd) а посему с ними придется распрощаться.

3. перезапускаются X-ы. Нужно проверить чтобы все работало (типа русские буковки в терминале и приложениях)
4. font.properties.ru -> \$JAVA_HOME/jre/lib
5. fonts.dir -> \$JAVA_HOME/jre/lib/fonts
6. cd \$JAVA_HOME/jre/lib/fonts; rm fonts.scale; ln -s fonts.dir fonts.scale

Теперь русские буквы должны вводиться и выводиться в свинге без проблем.

Как Вы можете заметить, в описании приводились ссылки на различные файлы font.properties.ru для Linux. Отличаются они тем, какие шрифты будут использованы по умолчанию в AWT и Swing. Соответственно Вам нужно выбрать один и пользоваться им.

Java: Русские буквы и не только...

- Linux/font.properties.ru — шрифты Cronyx cyrillic (koi8-r)
- LinuxWinFonts/font.properties.ru - шрифты от Windows
- LinuxIBM13/font.properties.ru — шрифты Lucida из JDK

22. Благодарности

Хочу поблагодарить коллег-программистов, которые помогли в составлении данного описания:

- [Сергей Безруков](#)
- [Денис Кокарев](#)
- [Алексей Менделев](#)
- [Михаил Иванов](#)
- [Константин Соболев](#)
- [Алексей Епишкин](#)
- [Алексей Тюрин](#)
- [Олег Вершинин](#)
- [Artemy E. Karitula](#)
- [Дмитрий Воробьев](#)
- [Станислав Миронов](#)
- [Руслан Хафизов](#)
- [Виталий Беридинских](#)
- [Вячеслав Яковенко](#)
- [Алексей Ковязин](#)
- [Евгений Коновалов](#)

Если Вы заметите неточность в описании или захотите дополнить, то напишите мне об этом, и Ваше имя так же появится в этом списке. :-)

Ну а если Вы просто хотите задать вопрос, который остался нераскрытым, можете задать его на этом форуме: <http://www.java.spb.ru/servlets/index?page=forum&name=rus&count=20>