Лекция 12

**Тема 2.6.** Интернаци онализа ция программ в Java

[6.1.1. Реализация многоязычных программ](#_2.6.1._Реализация_многоязычных)

[2.6.1.1. Компоненты интернационализации программ](#_2.6.1.1._Компоненты_интернационализ)

[2.6.1.2. Класс Locale](#_2.4.8.2._Класс_Locale)

[2.6.1.3. Кодирование символов в Java](#_2.6.1.3._Кодирование_символов)

[2.6.1.4. Конвертирование строк](#_2.6.1.4._Конвертирование_строк)

[2.6.2. Сравнение символов и строк](#_2.4.9._Сравнение_символов_и строк)

[2.6.2.1. Анализ символов с помощью метода getType() класса Character](#_2.6.2.1._Анализ_символов)

[2.6.2.2. Сравнение и сортировка строк](#_2.6.2.2._Сравнение_и)

[2.6.2.3. Задание правил сравнения строк](#_2.6.2.3._Задание_правил)

[2.6.2.4. Анализ символов, строк и предложений](#_2.6.2.4._Анализ_символов,)

[2.6.3. Изоляция данных, зависящих от Locale, от программы](#_2.6.3._Изоляция_данных,)

[2.6.3.1. Класс ResourceBundle](#_2.6.3.1._Класс_ResourceBundle)

[2.6.3.2. Класс ListResourceBundle](#_2.6.3.2._Класс_ListResourceBundle)

[2.6.3.3. Класс PropertyResourceBundle](#_2.6.3.3._Класс_PropertyResourceBund)

[2.6.4. Работа с датами](#_2.6.4._Работа_с)

[2.6.4.1. Класс Date](#_2.6.4.1._Класс_Date)

[2.6.4.2. Классы Calendar и GregorianCalendar](#_2.4.11.2._Классы_Calendar_и_Gregori)

## 2.6. Интернационализация программ в Java

### 2.6.1. Реализация многоязычных программ

#### 2.6.1.1. Компоненты интернационализации программ

Интернационализация (internationalization) – это процесс проектирования программы таким образом, чтобы она могла использоваться для различных языков и регионов без перепрограммирования. В документации и учебной литературе на английском языке для интернационализации часто используется сокращение i18n, поскольку слово internationalization содержит 18 букв.

Интернационализированная программа имеет следующие характеристики:

* при добавлении локализованных данных эта программы может быть выполнена в любой стране;
* текстовые элементы, такие как сообщения или надписи в графических приложениях и апплетах, не встроены в программу, а хранятся отдельно и доступ к ним происходит во время выполнения программы;
* включение новых языков не требует перекомпиляции программы;
* региональные (зависящие от языка и страны) данные, такие как дата, представление денежных единиц и чисел, выводятся в форматах, принятых в данном регионе.

Локализация – это процесс адаптации программного обеспечения для определенного региона или языка с помощью добавления зависящих от языка или региона компонент, а также перевода текста. Термин localization часто сокращают до l10n, поскольку это слово содержит 10 букв. Обычно подавляющую часть времени, требуемого на локализацию, занимает перевод текста. Другие типы данных, например аудиоданные или изображения, также могут требовать локализации. При локализации производится проверка соответствуют ли форматы представления дат, чисел и денежных единиц региональным и языковым требованиям.

Средства интернационализации и локализации программ в Java включают в себя следующие компоненты:

* определение и установка языка и региона, в котором выполняется программа;
* преобразование текста в различные кодировки символов;
* обработка текста (проверка свойств символов, сравнение строк, определение границ текста);
* изоляция выводимого пользователю текста от программы;
* форматирование выводимых значений даты, времени, чисел, денежных единиц и сообщений;

Приведенные компоненты содержатся в пакетах java.util и java.text. Ниже рассматриваются эти компоненты Java.

#### 2.6.1.2. Класс Locale

Объекты класса Locale представляют отдельные географические или политические области земного шара. Объект класса Locale создается одним из следующих двух конструкторов:

Locale(String language, String country)

Locale(String language, String country, String variant)

Первый параметр – language задает код языка в соответствии со стандартом ISO 639. В соответствии с эим стандартом код языка содержит две строчные буквы, например en – английский язык, de – немецкий язык, fr – французский язык, ru – русский язык или uk – украинский язык.

Второй параметр – country задает двухбуквенный код страны прописными буквами в соответствии со стандартом ISO 3166, например, US – США, GB – Великобритания, DE – Германия, FR – Франция, CA – Канада, RU – Россия или UA – Украина.

Код страны можно опустить, если задать в качестве параметра пустую строку.

Третий параметр – variant во втором конструкторе задает информацию, зависящую от компьютерной платформы. Значение этого параметра не стандартизовано и определяется, если необходимо, производителем программного продукта. В частности, в этом параметре может быть задана информация для Web-браузера, который функционирует на определенной компьютерной платформе, например, WIN – для Windows, MAC – для Macintosh или POSIX – для стандарта Unix.

Можно создавать объекты Locale для любых языков, коды которых задаются в ISO-639 и любые комбинации языка и страны, которые обычно задаются в виде:

код-языка\_код-страны (например, uk\_UA)

Однако это не означает что объекты классов, которые используют Locale, смогут работать с созданным объектом. В документации Java в файле guide\intl\locale.doc.html приведены допустимые комбинации страны и языка (в частности допустимы комбинации ru\_RU и uk\_UA, но не допустима комбинация ru\_UA).

Получить список допустимых языков и допустимых комбинаций язык-страна для какого-либо класса можно с помощью следующего метода:

public static Locale[] getAvailableLocales()

Этот метод определен как в классе Locale, так и в тех классах, которые используют объекты Locale, т.е. в рассматриваемых далее классах Calendar, GregorianCalendar и DateFormat. Так массив допустимых значений языка и комбинаций язык-страна для класса DateFormat можно определить с помощью оператора:

Locale list[] = DateFormat.getAvailableLocales(

Если значение объекта Locale не определено в программе, используется значение языка и страны, определенное в операционной системе. Для Windows эти установки задаются в Панели управления (Control Panel) в пункте Региональные установки (Regional Settings). В соответствии с выбранным языком (русским или украинским) значение объекта Locale по умолчанию будет либо ru\_RU, либо uk\_UA. Получить текущее значение объекта Locale по умолчанию можно с помощью метода класса Locale:

public static Locale getDefault()

Новый объект класса создается с помощью одного из приведенных выше конструкторов, например:

Locale USLocale = new Locale("en", "US");

Если необходимо, чтобы все операции в программе, зависящие от установок язык-страна, выполнялись в соответствии с новой переменной класса Locale, ее нужно сделать переменной Locale по умолчанию с помощью метода:

public static void setDefault(Locale newLocale)

Так, для переменной USLocale такая установка выполняется с помощью оператора:

Locale.setDefault(USLocale);

Помимо приведенных методов, в классе Locale определены также методы, позволяющие получить компоненты объекта Locale:

* public String getLanguage() – получение кода языка;
* public String getCountry() – получение кода страны;
* public String getVariant() – получение информации, зависящей от установки.

Следующие методы позволяют получить не коды, а наименования компонент объекта Locale:

* public final String getDisplayLanguage() – получение наименования языка;
* public final String getDisplayCountry() – получение наименования страны;
* public final String getDisplayVariant() – получение информации, зависящей от установки.

Метод

public final String getDisplayName()

позволяет получить и наименование языка и имя страны в виде язык(страна), например український (Україна).

Вспомогательные методы:

public String getISO3Country()

throws MissingResourceException

public String getISO3Language()

throws MissingResourceException

позволяют получить трехбуквенный код языка и страны для переменной Locale, а методы:

С помощью методов

* public static String[] getISOLanguages()
* public static String[] getISOCountries()

можно получить полные списки двухбуквенных кодов языков и стран (в соответствии со стандартами ISO 639 и ISO 3166).

#### 2.6.1.3. Кодирование символов в Java

Как отмечалось в начале курса, символы в Java представляются в двухбайтовой кодировке Unicode. Так, коды букв латинского алфавита A-Z и a-z лежат соответственно в диапазонах '\u0041' - '\u005A' и '\u0061' - '\u007A'. Коды букв русского алфавита А-Я и а-я лежат соответственно в диапазонах '\u0410' - '\u042F' и '\u0430' - '\u044F'. Исключением являются буквы Ё и ё, коды которых имеют значения '\u0401' и '\u0451'. Буквы украинского алфавита, совпадающие с русскими буквами, имеют те же коды, что и соответствующие русские буквы, а буквы Є, І, Ї, є, і и ї имеют соответственно коды '\u0404', '\u0406', '\u0407', '\u0454', '\u0456' и '\u0457'.

Однако в текстовых редакторах, используемых, в частности, при наборе программ, и при обмене данными между программой и внешними устройствами, в том числе дисплеем и клавиатурой используется байтовая кодировка символов. Для букв латинского алфавита и знаков препинания преобразование кодировки Unicode в байтовую кодировку, называемую кодировкой ASCII, выполняется с помощью отбрасывания старшего байта, а обратное преобразование – с помощью добавления старшего байта с нулевым значением.

С байтовой кодировкой букв других алфавитов, в частности букв русского и украинского алфавита, все обстоит гораздо сложнее. Традиционно для представления букв других алфавитов используются коды второй половины 256-символьной кодовой таблицы (с кодами '0x80' - '0x80'). Такое представление символов разных языков называется кодовыми страницами (code pages).

Для представления букв русского и украинского алфавитов существуют следующие основные кодировки:

* кодировка Windows (кодовая страница CP1251);
* кодировка MS DOS (кодовая страница для русского языка CP866);
* кодировка Unix (кодовая страница KOI8-R для русского языка и отдельная кодовая страница для украинского языка, иногда называемая KOI8-U);
* кодировка Apple Macintosh (кодовая страница MacCyrillic);
* международная кодировка кириллицы ISO8859-5 (относительная кодировка символов в этой кодировке соответствует кодировке Unicode).

В SDK 1.3 поддерживается множество кодовых страниц для различных языков и компьютерных платформ. Список поддерживаемых кодировок можно просмотреть в файле \guide\intl\encoding.doc.html документации SDK1.3. Из перечисленных выше кодировок в таблице представлены кодировки Cp1251, Cp866, KOI8-R (только для русского языка), MacCyrillic и ISO8859\_5.

Ни одна из этих кодовых страниц не является полностью тождественной другой, т.е. одни и те же буквы в этих станицах имеют разные коды.

Для букв других языков на основе латинского алфавита, а это большинство европейских языков, но использующих дополнительные буквы, например, буква ç во французском языке или буква ß в немецком языке, в Java также можно задать различные кодировки:

* ASCII – семибитовый код ASCII;
* ISO8859\_1 – латинский алфавит ISO №1;
* CP1252 – латинская кодировка в Windows.

Кроме того, в Java определены различные представления кодировки Unicode

* UnicodeBig – шестнадцатибитовый формат big-endian преобразования Unicode, порядок байтов определяется обязательным указателем порядка;
* UnicodeBigUnmarked – шестнадцатибитовый формат big-endian преобразования Unicode;
* UnicodeLittle – шестнадцатибитовый формат little-endian преобразования Unicode, порядок байтов определяется обязательным указателем порядка;
* UnicodeLittleUnmarked – шестнадцатибитовый формат little-endian преобразования Unicode;
* UTF-16 – шестнадцатибитовый формат преобразования Unicode, порядок байтов определяется обязательным указателем порядка;
* UTF8 – восьмибитовый формат преобразования Unicode.

Формат UTF8 предназначен для сокращения объема вводимых и выводимых данных. Данные в этом формате представляются по следующему алгоритму:

* символы Unicode в диапазоне '\u0000' - '\u007F' (первые 128 символов) записываются одним байтом (старший байт отбрасывается);
* символы Unicode в диапазоне '\u0080' - '\u07FF' (охватывающем буквы наиболее распространенных национальных алфавитов) записываются двумя байтами следующим образом: символ Unicode с двоичной кодировкой 00000xxxxxyyyyyy (где x и y – 0 или 1) преобразуется к виду 110xxxxx10yyyyyy;
* символы Unicode в диапазоне '\u0800' - '\uFFFF' записываются тремя байтами следующим образом: символ Unicode с двоичной кодировкой xxxxyyyyyyzzzzzz (где x, y и z – 0 или 1) преобразуется к виду 1110xxxx10yyyyyy10zzzzzz.

#### 2.6.1.4. Конвертирование строк

Кодовая страница, используемая по умолчанию для виртуальной машины Java, зависит от операционной системы. Так при установке региональных параметров English(United States) в Windows 98, кодовые страницы будут: 1252 – для Windows и 850 – для MS DOS под Windows. Если установлено, например, Russian или Ukrainian, то соответствующие номера кодовых страниц будут 1251 и 866. Текст программы на языке Java можно набирать в разных текстовых редакторах, работающих в разных кодировках. Так, при использовании приложения Notepad (Блокнот) или интегрированной среды разработки в Windows, текст программы (и соответственно задание констант для типов char и String) будет в кодировке CP1251. Если текст программы набран в среде MS DOS, то текст программы будет в кодировке CP866. Некоторые текстовые редакторы, такие как Word, Bred3 и встроенный редактор оболочки Far, позволяют задать вид кодировки.

При выполнении программы будет использоваться кодовая страница той среды, в которой происходит выполнение. Так, если программа набрана в кодировке 866 и выполняется в среде MS DOS (пакет SDK, как известно, функционирует в среде MS DOS), то оператор

System.out.println("Привет, Java!");

выведет на дисплей сообщение:

Привет, Java!

Если эту же программу, запустить в интегрированной оболочке, то сообщение будет выведено в кодировке 1251:

ЏаЁўҐв, Java!

Обратная ситуация – при выполнении программы, набранной в кодировке Windows, в среде MS DOS, приведет к выводу тем же оператором следующего сообщения:

-ЁштхЄ, Java!

Та же проблема возникает и при вводе данных с клавиатуры, а также при обмене данными с текстовыми файлами в различных кодировках. Именно поэтому во всех примерах в данном курсе все данные вводились и выводились на английском языке, буквы которого, как видно из предыдущего примера, выводятся правильно и в Windows и в MS DOS.

Следует отметить, что графические приложения и апплеты выполняются в среде Windows, поэтому для них кодовой страницей по умолчанию будет CP1251.

Для решения этой проблемы можно либо выполнять программу только в той среде, в которой она была набрана, либо использовать средства перекодировки, предоставляемые Java.

Кодировку символов строки, формируемой из байтового массива можно задать с помощью одного из двух конструкторов класса String:

public String(byte[] bytes, String enc)

throws UnsupportedEncodingException

public String(byte[] bytes, int offset, int length,

String enc) throws UnsupportedEncodingException

В первом параметре bytes задается массив байт, который должен быть преобразован в символьную строку в соответствии с кодовой страницей enc. Во втором конструкторе можно указать начальный индекс offset и длины length части массива bytes, преобразуемой в строку.

Метод класса String:

public byte[] getBytes(String enc)

throws UnsupportedEncodingException

позволяет выполняет обратную задачу – преобразование строки в массив байт в соответствии с заданной кодовой страницей enc.

Исключение UnsupportedEncodingException в конструкторе и методе бросается в том случае, если указано неверное имя кодовой страницы.

Так, если программа использует для записи символов русского и/или украинского языка кодировку Windows и требуется вывести на дисплей строки в кодировке MS DOS, необходимо выполнить следующие действия:

1. Преобразовать строку в массив символов с помощью вызова метода getBytes("Cp866").
2. Преобразовать созданный массив в строку с помощью конструктора public String(byte[] bytes).

Пример.

String greetingRusWin = "Привет";

byte convertArray [] = greetingRusWin.getBytes("Cp866");

String greetingRusDos = new String(convertArray);

System.out.println("MS DOS code: " + greetingRusDos);

Если в строке содержится украинская буква і, то в выводимой строке, перед преобразованием ее в массив, необходимо заменить украинскую букву і на латинскую букву i (поскольку в кодовой странице 866 для представления украинской буквы і используется код латинской буквы i).

Пример.

String greetingUkrWin = "Вітаємо";

greetingUkrWin = greetingUkrWin.replace('і', 'i');

byte convertArray [] = greetingUkrWin.getBytes("Cp866");

String greetingUkrDos = new String(convertArray);

System.out.println("MS DOS code: " + greetingUkrDos);

Если же программа использует для записи символов русского и/или украинского языка кодировку MS DOS и необходимо вывести на дисплей строки в кодировке Windows, необходимо выполнить следующие действия:

1. Преобразовать строку в массив символов с помощью вызова метода getBytes().
2. Преобразовать созданный массив в строку с помощью конструктора public String(byte[] bytes, "Cp866").

Пример.

String greetingRusDos = "Привет";

byte convertArray [] = greetingRusDos.getBytes();

String greetingRusWin = new String(convertArray, "Cp866");

System.out.println("Windows code: " + greetingRusWin);

Перекодирование символов строк из одной кодировки в другую можно автоматизировать, если для определения операционной системы использовать статический метод класса System:

public static String getProperty(String key)

в котором в параметре key задается наименование системного свойства.

Для определения операционной системы можно использовать следующие системные свойства:

* "os.name" – наименование операционной системы;
* "os.version" – версия операционной системы;
* "os.arch" – компьютерная платформа, в которой функционирует операционная система.

К сожалению, такой подход нельзя применить при запуске MS DOS под управлением Windows (и при работе в среде MS DOS и при работе в среде Windows для "os.name" будет выдано одно и то же значение, например "Windows 98").

При необходимости преобразования кодировки при чтении или записи данных в поток также можно использовать приведенный выше подход, однако более удобным является использование классов ввода-вывода InputStreamReader и OutputStreamWriter.

Использование конструкторов

InputStreamReader(InputStream in)

или

OutputStreamWriter(OutputStream out)

преобразует байтовые потоки в символьные (для InputStreamReader) и символьные потоки в байтовые (для OutputStreamWriter) с использованием кодовой страницы по умолчанию.

Если необходимо прочитать или вывести данные в другой кодировке можно использовать следующие формы конструкторов:

InputStreamReader(InputStream in, String enc) throws

UnsupportedEncodingException

или

OutputStreamWriter(OutputStream out, String enc) throws

UnsupportedEncodingException

В параметре enc этих конструкторов задается наименование кодовой страницы для перекодировки данных.

### 2.6.2. Сравнение символов и строк

Использование Unicode позволяет выполнять операции над символами, независимо от языка и кодировки, используемой для ввода-вывода.

#### 2.6.2.1. Анализ символов с помощью метода getType() класса Character

Метод класса Character:

public static int getType(char ch)

позволяет получить тип символа Unicode. Значения типов символов Unicode представлены как свойства (поля) класса Character. Некоторые из этих типов представлены в табл. 6.1.

Таблица 6.1. Значения основных полей класса Character

|  |  |
| --- | --- |
| Поле | Значение |
| static byte CONTROL | Управляющий символ Unicode. |
| static byte CURRENCY\_SYMBOL | Символ валюты Unicode (например, "$") |
| static byte LINE\_SEPARATOR | Разделитель строк в Unicode. |
| static byte SPACE\_SEPARATOR | Пробельный символ Unicode. |
| static byte PARAGRAPH\_SEPARATOR | Разделитель абзацев в Unicode. |
| static byte LOWERCASE\_LETTER | Буква нижнего регистра Unicode. |
| static byte UPPERCASE\_LETTER | Буква верхнего регистра Unicode. |
| static byte MATH\_SYMBOL | Математический символ Unicode. |

Пример:

int symbolType = Character.getType(currentSymbol);

if (symbolType == Character.MATH\_ SYMBOL)

System.out.println(currentSymbol +

" - математический символ");

#### 2.6.2.2. Сравнение и сортировка строк

В задачах анализа текстовой информации часто возникает необходимость в сравнении и сортировке строк. Анализируемые слова могут быть на разных языках, поэтому правила сравнения и сортировки в разных языках будут различными. Однако в каждом языке существует так называемый естественный порядок сравнения, обычно в соответствии с порядком следования букв в алфавите. Так, строка "abc" в английском языке будет при сортировке располагаться до строки "cde", а строка "acd" – между строками "abc" и "cde".

Для сравнения строк в соответствии с естественным порядком сравнения в Java в пакете java.text определен абстрактный класс Collator.

Объект класса Collator создается с помощью методов:

public static Collator getInstance()

для Locale по умолчанию или

public static Collator getInstance(Locale desiredLocale)

для указанной Locale, например,

Collator defaultCollator = Collator.getInstance();

или

Collator enCollator =

Collator.getInstance(new Locale("en", "US"));

Получить все доступные для класса Collator значения Locale можно с помощью метода

public static Locale[] getAvailableLocales()

класса Collator.

Метод getInstance() возвращает объект класса RuleBasedCollator, который является подклассом Collator.

Класс RuleBasedCollator содержит набор правил, определяющих порядок сортировки строк для заданного объекта Locale. Поскольку эти правила инкапсулированы внутри класса RuleBasedCollator, обычно программисту не надо писать специальных методов для правил сравнения строк.

Для того, чтобы сравнить между собой две строки, используется метод

public int compare(Object o1,Object o2)

класса Collator, возвращающий -1, если объект o1 меньше объекта o2, 0, если объекты равны и 1, если объект o1 больше объекта o2. В данной реализации Java сравниваются не сами объекты, а их строковые представления, т.е. параметры метода преобразуются к виду (String) Object o1 и (String) Object o2.

Примеры.

Результат выполнения оператора

enCollator.compare("abc", "acd")

будет -1, результат выполнения оператора

enCollator.compare("abc", "abc")

будет 0, а результат выполнения оператора

enCollator.compare("cde", "acd")

будет 1.

Метод compare() обычно используется для сортировки строк различными методами. Ниже приведен метод, реализующий сортировку строк для объекта Collator с помощью алгоритма «всплывающего пузырька»:

public static void sortStrings(Collator collator,

String[] words)

{

String tmp;

for(int i = 0; i < words.length; i++)

{

for(int j = i + 1; j < words.length; j++)

{

if(collator.compare(words[i], words[j]) > 0)

{

tmp = words[i];

words[i] = words[j];

words[j] = tmp;

}

}

}

}

Этот алгоритм сравнивает две соседние строки и, если первая строка больше второй, строки переставляются местами. Перестановки продолжаются до тех пор, пока все строки не будут отсортированы по возрастанию.

#### 2.6.2.3. Задание правил сравнения строк

Если при сравнении или сортировке строк необходимо использовать правила сравнения, отличные от правил, заданных по умолчанию для объекта Locale, необходимо создать новый объект класса, с использованием конструктора класса RuleBasedCollator:

public RuleBasedCollator(String rules)

throws ParseException

В строке rules задаются правила сравнения символов в строках. Для учета различных особенностей сравнения строк (степени, или силы, сравнения) в классе Collator определены четыре свойства:

* public static final int PRIMARY– различия в букве (например 'е' и 'и');
* public static final int SECONDARY – различия в знаках в букве (например 'е' и 'ё');
* public static final int TERTIARY– различия в регистре буквы (например 'е' и 'Е');
* public static final int IDENTICAL– все различия в буквах являются значимыми.

Методы класса Collator:

public int getStrength()

и

public void setStrength(int newStrength)

позволяют соответственно получить или установить степень сравнения строк.

Элементами строки rules являются символы или последовательности, между которыми задаются следующие отношения сравнения:

* '<' – больше как отличие в букве (первичное отличие);
* ';' – больше, как отличие в знаках в букве (вторичное отличие);
* ',' – больше, как отличие в регистре (третичное отличие);
* '=' – равенство.

Примеры:

1. String rulesString1 = "<A=a<Б=б<e=Е=ё=Ё"; – буквы А и а, Б и б считаются одинаковыми и буква б(Б) следует за буквой а(А), а буквы е(Е) и ё(Ё) считаются одной и той же буквой.

2. String rulesString2 = "<A,a<Б,б"; – буквы А и а, Б и б считаются различными и строки, начинающиеся с а или б идут после строк, начинающихся соответственно с А и Б.

3. String rulesString3 = "<Е,е;Ё,ё"; – буквы е и ё считаются различными и строки, начинающиеся с е или ё идут после строк, начинающихся соответственно с Е и Ё.

Строка обязательно должна начинаться с одного из символов отношения. Однако все символы перед первым символом '<' в строке рассматриваются как игнорируемые. Так строка

String rulesString4 = ",-<А,а<Б,б";

указывает, что символ '-' будет игнорироваться при сравнении слов, т.е. составные слова (типа красно-белый) будут рассматриваться как одно слово.

Правила можно задавать не только одним символом. Так, в испанском языке сочетание ch (звук 'ч') считается одной буквой и можно задать следующее правило сравнения:

String rulesString5 = "<C,c<Ch=CH=cH,ch<D,d";

Согласно этому правилу, все строки, начинающиеся на сочетание ch(CH) следуют после всех строк, начинающихся на букву c.

Символ '&' в строке, задающей правила сравнения, означает логическую операцию 'И'. Так запись

"a < b & ae < e"

означает, что буква b и сочетание букв ae будут при сравнении строк считаться меньшими, чем буква e.

После задания правила можно создать новый объект RuleBasedCollator, например:

String simpleRule = "< a < b < c < d";

RuleBasedCollator simpleCollator =

new RuleBasedCollator(simpleRule);

Получить текущую строку сравнения для объекта RuleBasedCollator можно с помощью метода

public String getRules()

класса RuleBasedCollator.

Если в строке сравнения rules нарушены перечисленные выше синтаксические правила формирования строки, генерируется исключение ParseException.

Изменения в правилах сравнения можно также делать, лишь частично изменяя существующие правила сравнения. Так, добавить с существующим правилам сравнения в английском языке приведенное выше правило для сочетания ae можно с помощью следующей последовательности операторов:

RuleBasedCollator en\_USCollator = (RuleBasedCollator)

Collator.getInstance(new Locale("en", "US"));

String addRules = "& a < b & ae < e";

RuleBasedCollator myCollator =

new RuleBasedCollator(en\_USCollator + addRules);

Данная возможность позволяет также организовать сравнение строк, содержащих символы нескольких языков, например, для английского и русского языков такое объединение правил выглядит следующим образом:

// Создание объекта RuleBasedCollator

// для английского языка

RuleBasedCollator enRulesCollator = (RuleBasedCollator)

Collator.getInstance(new Locale("en", "US"));

// Создание объекта RuleBasedCollator

// для русского языка

RuleBasedCollator ruRulesCollator = (RuleBasedCollator)

Collator.getInstance(new Locale("ru", "RU"));

// Получение правил сравнения для английского языка

String enRules = enRulesCollator.getRules();

// Получение правил сравнения для русского языка

String ruRules = ruRulesCollator.getRules();

// Объединение двух объектов Collator: сначала

// для английского языка, затем для русского языка

RuleBasedCollator combineCollator =

new RuleBasedCollator(enRules + ruRules);

#### 2.6.2.4. Анализ символов, строк и предложений

Приложение, работающее с текстовыми данными должно уметь определять логические границы элемента текста: символа, слова, строки, предложения. Для определения этих границ можно использовать класс BreakIterator из пакета java.text.

Как и в рассмотренных выше классах, результаты вызова методов класса BreakIterator зависят от используемого объекта Locale. Список доступных для класса объектов Locale можно получить с помощью метода

public static Locale[] getAvailableLocales()

класса BreakIterator.

Конструктор класса BreakIterator объявлен с атрибутом protected, поэтому объекты класса BreakIterator можно создать с помощью следующих методов этого класса:

* public static BreakIterator getCharacterInstance() и public static BreakIterator getCharacterInstance(Locale where) – возвращает объект BreakIterator, определяющий логические границы символа для Locale по умолчанию или указанного объекта Locale;
* public static BreakIterator getWordInstance() и public static BreakIterator getWordInstance(Locale where) – возвращает объект BreakIterator, определяющий логические границы слова для Locale по умолчанию или указанного объекта Locale;
* public static BreakIterator getSentenceInstance() и public static BreakIterator getSentenceInstance(Locale where) – возвращает объект BreakIterator, определяющий логические границы предложения для Locale по умолчанию или указанного объекта Locale;
* public static BreakIterator getLineInstance() и public static BreakIterator getLineInstance(Locale where) – возвращает объект BreakIterator, определяющий логические границы строки для Locale по умолчанию или указанного объекта Locale.

Как и в любом итераторе, в классе BreakIterator определен виртуальный курсор, который можно перемещать по текстовой строке с помощью следующих методов класса BreakIterator:

* public abstract int current() – возвращает текущую границу в текстовой строке;
* public abstract int next() – возвращает границу, следующую за текущей границей;
* public abstract int next(int n) – возвращает n-ую границу, следующую за текущей границей;
* public abstract int previous() – возвращает границу, предшествующую текущей границе;
* public abstract int following(int offset) – возвращает первую границу, следующую за заданным смещением в строке;
* public int preceding(int offset) – возвращает последнюю границу, предшествующую заданному смещению в строке;
* public abstract int first() – возвращает первую границу в строке;
* public abstract int last() – возвращает последнюю границу в строке.

Метод класса BreakIterator

public boolean isBoundary(int offset)

позволяет определить является ли значение символа в указанной позиции offset граничным символом.

Свойство класса BreakIterator

public static int BreakIterator.DONE

позволяет определить окончание просмотра строки. Если позиция, возвращаемая методами next() или previous(), равна BreakIterator.DONE, это означает, что просмотр строки окончен.

Метод класса BreakIterator

public abstract CharacterIterator getText()

позволяет получить просматриваемую текстовую строку как объект интерфейса CharacterIterator, а методы

public abstract void setText(CharacterIterator newText)

public void setText(String newText)

устанавливают текстовую строку для просмотра, определенную соответственно как объект класса CharacterIterator или как объект класса String.

Рассмотрим подробнее анализ различных элементов текста.

Анализ символов используется при выделении отдельных символов и перемещении курсора в строке по одному символу. Например, объект для определения границ символов для Locale по умолчанию создается с помощью оператора

BreakIterator charIterator =

BreakIterator.getCharacterInstance();

Далее операции с символами в строке проводятся с использованием созданного объекта charIterator, например, вывод позиций всех границ символов в строке testString выполняется с помощью следующих операторов:

// Формирование строки для анализа

String testString =

"Приветствуем Вас в России! Как Ваши дела? " +

" Очень хорошо. ";

// Установка текста для просмотра

// с помощью заданного итератора

charIterator.setText(testString);

// Установка первой границы элемента в строке

int boundary = charIterator.first();

// Получение границ элементов в цикле

while (boundary != BreakIterator.DONE)

{

// Вывод текущей границы

System.out.print(boundary);

}

// Получение следующей границы

boundary = charIterator.next();

Во многих языках (в том числе, русском, украинском и английском) понятие символа Unicode и символа языка совпадают. Однако в некоторых языках (например, арабском) символ языка, называемый пользовательским символом, формируется, как правило, из нескольких символов Unicode. В таких языках границы пользовательских символов не совпадают с границами символов Unicode.

Если в приложении необходимо выполнять обработку отдельных слов в строке (например, выделение, копирование, перемещение или удаление слов в текстовом редакторе) необходимо создать объект BreakIterator для определения границ слов, например:

BreakIterator wordIterator =

BreakIterator.getWordInstance();

Далее с помощью этого объекта можно выполнять необходимые операции над словами. Поскольку, как видно из предыдущего примера, при анализе используется только объект BreakIterator, а не тип объекта, можно переписать предыдущий пример в виде отдельного метода:

public void positionList(String target,

BreakIterator iterator)

{

// Установка текста для просмотра

// с помощью заданного итератора

iterator.setText(testString);

// Установка первой границы элемента в строке

int boundary = iterator.first();

// Получение границ элементов в цикле

while (boundary != BreakIterator.DONE)

{

// Вывод текущей границы

System.out.print(boundary);

}

// Получение следующей границы

boundary = iterator.next();

}

С помощью оператора

positionList(testString, wordIterator);

можно получить позиции границ слов в строке. Такие символы, как пробелы, знаки пунктуации и табуляции, также считаются отдельными словами. Чтобы выделить, например, только слова из букв и числа в строке, необходимо проанализировать первый символ – является ли он буквой или числом.

Позиции допустимого переноса слов в строке можно получить, создав, например, следующий объект класса BreakIterator:

BreakIterator lineIterator =

BreakIterator.getLineInstance();

Этот объект можно использовать, например, в текстовом редакторе для переноса текста на следующую строку. Позиции переноса определяются по правилам, принятым для данного объекта Locale.

И, наконец, обработку предложений можно выполнить, создав, например, следующий объект класса BreakIterator:

BreakIterator sentenceIterator =

BreakIterator.getSentenceInstance();

Используя определенный выше метод positionList, с помощью оператора

positionList(testString, sentenceIterator);

можно получить границы предложений в строке testString.

### 2.6.3. Изоляция данных, зависящих от Locale, от программы

От языка и региона, в котором используется программа, зависит не только кодировка данных и работа с символами и строками. Интерфейс с пользователем в программе, в частности, в графическом приложении, обычно выполняется на том языке или языках того региона, где предполагается использовать эту программу. Так, программа, в которой все надписи и сообщения выводятся на украинском языке, вряд ли будет пользоваться спросом за пределами Украины, даже если она является очень необходимой для пользователей других стран. Переписывание исходного текста программы для другого языка иногда, при неудачном проектировании программы (когда, например, размеры компонент увязываются с надписями на них), становится довольно сложной задачей.

Для решения этой задачи в Java предлагаются средства, позволяющие изолировать объекты, которые зависят от языка и региона, от текста программы, т.е. программа автоматически корректируется на этапе выполнения с учетом объектов Locale по умолчанию или заданных объектов Locale.

#### 2.6.3.1. Класс ResourceBundle

Для изоляции объектов, зависящих от языка и региона, используется класс ResourceBundle в пакете java.util.

Пакет, или набор, ресурсов (resource bundle) содержит объекты, зависящие от значения Locale. Когда эти ресурсы необходимы программе, они загружаются из пакета, в соответствии с заданным значением Locale.

Объект класса ResourceBundle обычно задается с помощью одного из следующих конструкторов:

public static final ResourceBundle

getBundle(String baseName) throws MissingResourceException

и

public static final ResourceBundle

getBundle(String baseName, Locale locale)

В первом конструкторе пакет ресурсов задается для объекта Locale по умолчанию, во второй – для заданного в конструкторе значения locale.

Концептуально пакет ресурсов представляет собой набор подклассов с одним и тем же базовым именем, задаваемым в параметре baseName. Подкласс с именем baseName определяет объекты, используемые для значения Locale по умолчанию.

Имена подклассов для данного базового имени, кода языка, кода страны и значения, зависящего от производителя (варианта) образуются следующим образом:

1. имя\_базового\_класса + "\_" + код\_языка + "\_" + код\_страны + "\_" + имя\_варианта

2. имя\_базового\_класса + "\_" + код\_языка + "\_" + код\_страны + "\_" +

имя\_варианта + ".properties"

3. имя\_базового\_класса + "\_" + код\_языка + "\_" + код\_страны

4. имя\_базового\_класса + "\_" + код\_языка + "\_" + код\_страны + ".properties"

5. имя\_базового\_класса + "\_" + код\_языка

6. имя\_базового\_класса + "\_" + код\_языка + ".properties"

7. имя\_базового\_класса + "\_" + код\_языка\_по\_умолчанию + "\_" +

код\_страны\_по\_умодчанию + "\_" + имя\_варианта\_по\_умолчанию

8. имя\_базового\_класса + "\_" + код\_языка\_по\_умолчанию + "\_" +

код\_страны\_по\_умодчанию + "\_" + имя\_варианта\_по\_умолчанию +

".properties"

9. имя\_базового\_класса + "\_" + код\_языка\_по\_умолчанию + "\_" +

код\_страны\_по\_умодчанию

10. имя\_базового\_класса + "\_" + код\_языка\_по\_умолчанию + "\_" +

код\_страны\_по\_умодчанию + "\_" + ".properties"

11. имя\_базового\_класса + "\_" + код\_языка\_по\_умолчанию

12. имя\_базового\_класса + "\_" + код\_языка\_по\_умолчанию + ".properties"

13. имя\_базового\_класса

14. имя\_базового\_класса + ".properties"

Если имя пакета ресурсов имеет в конце суффикс ".properties", то ресурс должен быть определен не в подклассе, а в файле с указанным именем.

Так для ресурса, заданного с помощью оператора

ResourceBundle myBundle =

ResourceBundle.getBundle("MyResource",

new Locale("en", "US");

и Locale по умолчанию ru\_RU, полный набор возможных подклассов и файлов содержит следующие элементы:

MyResource\_en\_US (подкласс)

MyResource\_en\_US.properties (файл)

MyResource\_en (подкласс)

MyResource\_en.properties (файл)

MyResource\_ru\_RU (подкласс)

MyResource\_ru\_RU.properties (файл)

MyResource\_ru (подкласс)

MyResource\_ru.properties (файл)

MyResource (подкласс)

MyResource.properties (файл)

Нет необходимости задавать все приведенные выше подклассы и файлы. Если пакет ресурсов для заданного значения Locale (в нашем примере – MyResource\_en\_US) не существует, то в поиске наиболее подходящего соответствия просматривается весь список возможных значений имен пакета в том порядке, в котором эти значения приведены выше. Таким образом, следующим ищется файл MyResource\_en\_US.properties и, если такого файла не существует, подкласс MyResource\_en и т.д. Если не найден ни один из указанных в списке ресурсов, генерируется исключительная ситуация MissingResourceException.

Метод класса ResourceBundle

public Locale getLocale()

возвращает значение Locale для объекта ResourceBundle.

Объекты ResourceBundle содержат массивы пар «ключ-значение». Ключ должен быть только объектом String. Если значение задается в подклассе, оно может быть объектом любого класса, но если значение задается в файле, оно может быть только объектом String.

Метод класса ResourceBundle

protected abstract Object handleGetObject(String key)

throws MissingResourceException,

который должен переопределяться в подклассах класса ResourceBundle, возвращает значение для заданного в его параметре ключа key.

Пусть пакет myBundle используется для задания надписей на двух кнопках "OK" и "Отменить".

Тогда необходимо определить следующие подклассы:

// По умолчанию (Locale=ru\_RU)

abstract class MyResource extends ResourceBundle

{

public Object handleGetObject(String key)

{

if (key.equals("okKey"))

return "OK";

else if (key.equals("cancelKey"))

return "Отменить";

else

return null;

}

}

// Английский язык (Locale=en\_US)

public class MyResource\_en\_US extends MyResource

{

public Object handleGetObject(String key)

{

// Поскольку класс MyResource обрабатывает

// кнопку с ключом "okKey", описывать ее

// обработку в этом классе нет необходимости

if (key.equals("cancelKey"))

return "Cancel";

else

return null;

}

}

#### 2.6.3.2. Класс ListResourceBundle

Список ресурсов можно определять также в классе, расширяющем класс ListResourceBundle. Класс ListResourceBundle является абстрактным подклассом класса ResourceBundle и содержит метод задания списка ключей и значений:

protected abstract Object[][] getContents()

который должен быть переопределен в конкретном подклассе класса ListResourceBundle, как показано на приведенном ниже примере:

public class MyResource extends ListResourceBundle

{

public Object [][] getContents ()

{

return contents;

}

private Object [][] contents =

{

{"okKey", "OK"},

{"cancelKey", "Отменить"}

};

}

#### 2.6.3.3. Класс PropertyResourceBundle

Класс PropertyResourceBundle является конкретным подклассом класса ResourceBundle, который управляет ресурсами при доступе к ключам и значениям, заданным в файле свойств. В отличие от класса ListResourceBundle, нет необходимости создавать подклассы данного класса, поскольку метод ResourceBundle.getBundle() автоматически ищет соответствующий файл свойств и создает объект PropertyResourceBundle, который ссылается на него.

Ключ и значение записываются в файле свойств следующим образом:

ключ = значение

Каждая пара «ключ-значение» должна располагаться на отдельной строке.

В файле свойств могут быть также заданы комментарии. Строки комментариев должны содержать символ "#" в первой позиции.

Файлы свойств можно создавать двумя способами: вручную и с использованием методов класса Properties, который, как и классы, связанные с управлением пакетами ресурсов, находится в пакете java.util.

Так, при создании вручную файла MyResource.properties, он будет иметь следующий вид:

okKey = OK

cancelKey = Отменить

а содержимое файла MyResource\_en\_US.properties будет содержать следующие строки:

okKey = OK

cancelKey = Cancel

Однако, если в файле свойств используется кодировка, отличная от кодировки ISO8859\_1, как, например, в файле MyResource.properties, то программа неверно перекодирует значения в строках файла (вместо русских букв будут выводиться буквы западноевропейских алфавитов).

Чтобы буквы интерпретировались правильно, необходимо использовать находящуюся в подкаталоге bin каталога, содержащего Java SDK, утилиту native2ascii.exe. Эта утилита преобразует символы не в кодировке Unicode (за исключением символов кодировки ISO8859\_1) в символы Unicode и выполняет обратное преобразование.

Формат запуска программы:

native2ascii [опции] [исходный-файл [выводной-файл]]

Необязательный параметр опции может иметь значения:

-reverse и/или

-encoding имя-кодовой-страницы

Если опция -reverse задана, выполняется обратное преобразование – из кодировки Unicode (если она опущена, выполняется преобразование в кодировку Unicode).

Если задана опция -encoding, то при преобразовании используется кодировка имя-кодовой-страницы, заданная в параметре опции. Если параметр имя-кодовой-страницы опущен, используется значение системного свойства "file.encoding".

Параметры исходный-файл и выводной-файл задают соответственно имена исходного файла и файла, который будет содержать результаты преобразования символов файла. Если выводной-файл опущен для вывода используется стандартный вывод. Если же опущен исходный-файл, для ввода используется стандартный ввод.

Объект класса Properties можно создать с помощью одного из следующих конструкторов:

public Properties()

public Properties(Properties defaults)

Оба конструктора создают пустой список свойств, однако второй конструктор использует ранее созданные список свойств, как список свойств по умолчанию (этот список содержит значения для ключей, которые не были найдены в основном списке).

При создании файла свойств с помощью класса Properties можно использовать следующие методы этого класса:

* public Object setProperty(String key, String value) – установка ключа и значения для свойства из списка свойств объекта Properties;
* public String getProperty(String key) – получение свойства по его ключу;
* public void store(OutputStream out, String header) throws IOException – сохранение пар «ключ-значение» в выводном потоке out с комментарием header;
* public void load(InputStream inStream) throws IOException – загрузка пар «ключ-значение» из входного потока inStream;
* public void list(PrintStream out) и public void list(PrintWriter out) – печать набора свойств в заданный выводной поток out (используется для отладки).

Кроме того, в классе System определен метод

public static Properties getProperties(),

позволяющий получить список системных свойств виртуальной машины Java (JVM) на данном компьютере в виде объекта класса Properties.

Создание файла MyResource.properties при использовании приведенных методов можно выполнить с помощью следующих операторов:

Properties myProperties = new Properties();

myProperties.setProperty("okKey", "OK");

myProperties.setProperty(("cancelKey", "Отменить");

myProperties.store(new

FileOutputStream("MyResource.properties"),

"Default property file");

При записи списка свойств в файл значения (если они не в кодировке ISO8859\_1) будут автоматически преобразованы в кодировку Unicode. Однако следует иметь в виду, что при перекодировке используется значение системного свойства "file.encoding", поэтому, например, строки в Windows, заданные в кодировке MS DOS, должны быть преобразованы в кодировку Windows перед записью в файл.

Доступ к ресурсу выполняется в программе с помощью одного из следующих методов класса ResourceBundle:

public final String getString(String key)

throws MissingResourceException

public final String[] getStringArray(String key)

throws MissingResourceException

public final Object getObject(String key)

throws MissingResourceException

Первый метод и второй методы можно использовать, когда известно, что значение или значения, соответствующие ключу key, являются объектами String. Третий метод требует явного приведения типа к типу возвращаемого значения.

Для объекта myBundle получение надписей для заданной Locale и помещение надписей на кнопки можно выполнить с помощью следующих операторов:

String okLabel = myBundle.getString("okKey");

Button okButton = new Button(okLabel);

String cancelLabel = myBundle.getString("cancelKey");

Button cancelButton =new Button(cancelLabel);

Для получения надписей будет сначала просматриваться класс MyResource\_en\_US. Если класс с таким именем не будет найден, в текущем каталоге будет проведен поиск файла MyResource\_en\_US.properties. Если нет файла с этим именем, а также подклассов и файлов, содержащих в названии только имя языка, будут просматриваться подкласс MyResource, а затем файл MyResource.properties. Таким образом, ресурсы можно либо определять в подклассе, либо помещать в файл.

### 2.6.4. Работа с датами

#### 2.6.4.1. Класс Date

В версии JDK 1.0 работа с датами и временем была реализована с помощью класса Date, в котором можно было получить значение текущей даты (год, месяц, число, день недели) и времени (часы, минуты и секунды), а также установить значения этих величин. Класс Date находится в пакете java.util. Однако, начиная с JDK 1.1, концепция работы с датами и временем была существенно изменена. Хотя классом Date можно пользоваться, практически все конструкторы и методы этого класса объявлены отмененными (deprecated). Это означает, что в будущих версиях JDK(SDK) эти конструкторы и методы могут быть не реализованы. В классе Date оставлен только конструктор Date(), создающий новый объект типа Date, которому значение количества миллисекунд, прошедших с начала условной точки отсчета – полночь по Гринвичу 1 января 1970 года. Кроме того, оставлены два метода:

public Boolean after(Date when)

public Boolean before(Date when).

Эти методы возвращают значение true, если время в миллисекундах соответственно больше или меньше указанного в параметре when времени, а также метод

public int compareTo(Date anotherDate),

возвращающий 0, если сравниваемые даты равны, значение больше нуля, если значение сравниваемой даты больше anotherDate и меньше нуля – в противном случае.

#### 2.6.4.2. Классы Calendar и GregorianCalendar

Для работы с датами и временем введен класс Calendar, который также находится в пакете java.util. Этот класс реализован как абстрактный базовый класс для преобразования между объектом Date, представляющим время в миллисекундах и набором целых полей, таких как YEAR, MONTH, DAY, HOUR и т.д. Подклассы Calendar интерпретируют дату и время в соответствии с конкретной календарной системой.

В настоящее время в Java реализован только один подкласс класса Calendar: GregorianCalendar – григорианский календарь. Этот календарь используется практически во всем мире. В дальнейшем предполагается представление других календарных систем, в частности, лунных календарей.

В классе GregorianCalendar реализованы следующие конструкторы:

* GregorianCalendar() – создает григорианский календарь с использованием текущего времени во временной зоне по умолчанию и с локальными характеристиками по умолчанию.
* GregorianCalendar(int year, int month, int date) – создает григорианский календарь с данной датой во временной зоне по умолчанию и с локальными характеристиками по умолчанию.
* GregorianCalendar(int year, int month, int date, int hour, int minute) – создает григорианский календарь с данной датой и временем во временной зоне по умолчанию и с локальными характеристиками по умолчанию.
* GregorianCalendar (int year, int month, int date, int hour, int minute, int second) – создает григорианский календарь с данной датой и временем во временной зоне по умолчанию и с локальными характеристиками по умолчанию.
* GregorianCalendar(Locale aLocale) – создает григорианский календарь с текущими датой и временем во временной зоне по умолчанию и с заданными локальными характеристиками.
* GregorianCalendar(TimeZone zone) – создает григорианский календарь с текущими датой и временем в заданной временной зоне и с локальными характеристиками по умолчанию.
* GregorianCalendar (TimeZone zone, Locale aLocale) – создает григорианский календарь с текущей датой и текущим временем в заданной временной зоне и с заданными локальными характеристиками.

Класс GregorianCalendarсодержит два поля:

static int BC

static int AD,

содержащие соответственно значения «до нашей эры» и «нашей эры».

Кроме того, класс GregorianCalendarнаследует следующие поля класса Calendar (все поля имеют модификаторы public static final и тип int):

* от SUNDAY до SATURDAY – значения дней недели;
* от JANUARY до DECEMBER – значения месяцев;
* DAY\_OF\_WEEK, MONTH, DAY\_OF\_MONTH, DAY\_OF\_WEEK\_IN\_MONTH, DAY\_OF\_YEAR, WEEK\_OF\_YEAR, YEAR – соответственно значения дня недели, номера месяца, дня месяца, порядкового дня недели в месяце, номера дня в году, номера недели в году и года;
* DATE – день месяца;
* HOUR и HOUR\_OF\_DAY – соответственно время до полудня или после полудня (используется вместе с полями AM и PM) и время дня (от 0 до 24);
* DST\_OFFSET и ZONE\_OFFSET – смещение летнего времени в миллисекундах и смещение от Гринвича в миллисекундах;
* MINUTE, SECOND, MILLISECOND – значения минут, секунд и миллисекунд.

Получить значения указанных выше полей можно с помощью метода

public int get(int field),

а установить значение поля – с помощью метода

public void set(int field, int value),

например:

Calendar calendar = new GregorianCalendar();

int year1, year2;

year2 = calendar.get(calendar.YEAR);

year1 = calendar.set(calendar.YEAR, 1995);

Для одновременной установки нескольких полей можно использовать следующие формы метода set():

public final void set(int year, int month, int date)

public final void set(int year, int month,

int date, int hour, int minute)

public final void set(int year, int month,

int date, int hour, int minute, int second)

Методы

public abstract void roll(int field, boolean up)

public void roll(int field, int amount)

обеспечивают прокрутку заданного поля на одну единицу вверх (up равно true) или вниз (up равно false) или прокрутку на заданное количество единиц amount.

Методы

public final void clear(int field)

public final void clear()

очищают соответственно указанное поле или все временные поля.