Глава 7

СТРОКИ

Строка — это застывшая структура данных, и повсюду, куда она передается, происходит значительное дублирование процесса. Это идеальное средство для сокрытия информации.

Алан Дж. Перлис

Строка в языке Java — это основной носитель текстовой информации. Это не массив символов типа **char**, а объект соответствующего класса. Системная библиотека Java содержит классы **String**, **StringBuilder** и **StringBuffer**, поддерживающие работу со строками и определенные в пакете **java.lang**, подключаемом автоматически. Эти классы объявлены как **final**, что означает невозможность создания собственных порожденных классов со свойствами строки. Для форматирования и обработки строк применяются классы **Formatter**, **Pattern**, **Matcher** и другие.

Класс String

Каждая строка, создаваемая с помощью оператора **new** или с помощью литерала (заключенная в двойные апострофы), является экземпляром класса **String**. Особенностью объекта класса **String** является то, что его значение не может быть изменено после создания объекта при помощи какого-либо метода класса, так как любое изменение строки приводит к созданию нового объекта.

Класс **String** поддерживает несколько конструкторов, например: **String(), String(String str)**, **String(byte[] asciichar)**, **String(char[] unicodechar)**, **String(StringBuffer sbuf)**, **String(StringBuilder sbuild)** и др. Эти конструкторы используются для создания объектов класса **String** на основе инициализации значениями из массива типа **char**, **byte** и др. Например, при вызове конструктора

```
new String(str.getBytes(), "UTF-8")
```

можно установить кодировку создаваемому экземпляру в качестве второго параметра конструктора. Когда Java встречает литерал, заключенный в двойные кавычки, автоматически создается объект-литерал типа **String**, на который можно установить ссылку. Таким образом, объект класса **String** можно создать, присвоив ссылке на класс значение существующего литерала или с помощью оператора **new** и конструктора, например:

```
String s1 = "oracle.com";
String s2 = new String("oracle.com");
```

Класс **String** содержит следующие методы для работы со строками:

String concat(String s) или «+» — слияние строк;

boolean equals(Object ob) и **equalsIgnoreCase(String s)** — сравнение строк с учетом и без учета регистра соответственно;

int compareTo(String s) и **compareToIgnoreCase(String s)** — лексикографическое сравнение строк с учетом и без учета их регистра. Метод осуществляет вычитание кодов первых различных символов вызывающей и передаваемой строки в метод строк и возвращает целое значение. Метод возвращает значение нуль в случае, когда **equals()** возвращает значение **true**;

boolean contentEquals(StringBuffer ob) — сравнение строки и содержимого объекта типа **StringBuffer**;

String substring(int n, int m) — извлечение из строки подстроки длины **m-n**, начиная с позиции **n**. Нумерация символов в строке начинается с нуля;

String substring(int n) — извлечение из строки подстроки, начиная с позиции **n**;

int length() — определение длины строки;

int indexOf(char ch) — определение позиции символа в строке;

static String valueOf(*значение***)** — преобразование переменной базового типа к строке;

String toUpperCase()/toLowerCase() — преобразование всех символов вызывающей строки в верхний/нижний регистр;

String replace(char c1, char c2) — замена в строке всех вхождений первого символа вторым символом;

String intern() — заносит строку в «пул» литералов и возвращает ее объектную ссылку;

String trim() — удаление всех пробелов в начале и конце строки;

char charAt(int position) — возвращение символа из указанной позиции (нумерация с нуля);

boolean isEmpty() — возвращает **true**, если длина строки равна **0**;

char[] getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) — извлечение символов строки в массив символов;

static String format(String format, Object… args), **format(Locale l, String format, Object… args)** — генерирует форматированную строку, полученную с использованием формата, интернационализации и др.;

String[] split(String regex), String[] split(String regex, int limit) — поиск вхождения в строку заданного регулярного выражения (разделителя) и деление исходной строки в соответствии с этим на массив строк.

Во всех случаях вызова методов, изменяющих строку, создается новый объект типа **String**.

Эффективной демонстрацией работы методов класса служит преобразование строки в массив объектов типа **String** и их сортировка в алфавитном порядке.

Ниже рассмотрена сортировка массива строк методом выбора, таким образом демонстрируются возможности методов класса.

// # 1 # сортировка # SortArray.java

```
package by.bsu.strings;
public class SortArray {
          public static void main(String[ ] args) {
             String names = "   Alena   Alice     alina albina     Anastasya   ALLA ArinA     ";
             names = names.trim(); // удаление пробелов по краям строки
             // разбиение строки на подстроки, где "пробел" – разделитель
             String a[ ] = names.split(" "); // массив содержит элементы нулевой длины
             for(int j = 0; j < a.length – 1; j++) {
                    for(int i = j + 1; i < a.length; i++) {
                       if(a[i].compareToIgnoreCase(a[j]) < 0) {
                              String temp = a[j];
                              a[j] = a[i];
                              a[i] = temp;
                       }
                    }
             }
             for (String arg : a) {
                    if (!arg.isEmpty()) {
                              System.out.print(arg + " ");
                    }
             }
          }
}
```

albina Alena Alice alina ALLA Anastasya ArinA

Вызов метода **trim()** обеспечивает удаление всех начальных и конечных символов пробелов. Метод **compareToIgnoreCase()** выполняет лексикографическое сравнение строк между собой по правилам Unicode. Оператор **if(!arg.isEmpty())** не позволяет выводить пустые строки.

При использовании методов класса **String**, изменяющих строку, создается новый обновленный объект класса **String**. Сохранить произведенные изменения экземпляра класса **String** можно только с применением оператора присваивания, т. е. установкой ссылки на вновь созданный объект. В следующем примере будет подтвержден тезис о неизменяемости экземпляра типа **String**.

/ # 2 # передача строки по ссылке # RefString.java */*

```
package by.bsu.strings;
public class RefString {
          public static void changeStr(String s) {
                    s = s.concat(" Certified"); // создается новая строка
                    // или  s.concat(" Certified");
                    // или s += " Certified";
```

```
        }
        public static void main(String[ ] args) {
                String str = new String("Java");
                changeStr(str);
                System.out.print(str);
        }
}
```

В результате будет выведена строка:

Java

Поскольку объект был передан по ссылке, любое изменение объекта в методе должно сохраняться и для исходного объекта, так как обе ссылки равноправны. Это не происходит по той причине, что вызов метода **concat(String s)** приводит к созданию нового объекта.

При создании экземпляра класса **String** путем присваивания его ссылки на литерал, последний помещается в так называемый «пул литералов». Если в дальнейшем будет создана еще одна ссылка на литерал, эквивалентный ранее объявленному, то будет произведена попытка добавления его в «пул литералов». Так как идентичный литерал там уже существует, то дубликат не может быть размещен, и вторая ссылка будет на существующий литерал. Аналогично в случае, если литерал является вычисляемым. То есть компилятор воспринимает литералы **"Java"** и **"J" + "ava"** как эквивалентные.

// # 3 # сравнение ссылок и обьектов # EqualStrings.java

```
package by.bsu.strings;
public class EqualStrings {
        public static void main(String[ ] args) {
                String s1 = "Java";
                String s2 = "Java";
                String s3 = new String("Java");
                String s4 = new String(s1);
                System.out.println(s1 + "==" + s2 + " : " + (s1 == s2)); // true
                System.out.println(s3 + "==" + s4 + " : " + (s3 == s4)); // false
                System.out.println(s1 + "==" + s3 + " : " + (s1 == s3)); // false
                System.out.println(s1 + " equals " + s2 + " : " + s1.equals(s2)); // true
                System.out.println(s1 + " equals " + s3 + " : " + s1.equals(s3)); // true
        }
}
```

В результате, например, будет выведено:

Java==Java : true
Java==Java : false
Java==Java : false
Java equals Java : true
Java equals Java : true

Несмотря на то, что одинаковые по значению строковые объекты расположены в различных участках памяти, значения их хэш-кодов совпадают.

Так как в Java все ссылки хранятся в стеке, а объекты — в куче, то при создании объекта **s2** сначала создается ссылка, а затем этой ссылке устанавливается в соответствие объект. В данной ситуации **s2** ассоциируется с уже существующим литералом, так как объект **s1** уже сделал ссылку на этот литерал. При создании **s3** происходит вызов конструктора, т. е. выделение памяти происходит раньше инициализации, и в этом случае в куче создается новый объект.

Существует возможность сэкономить память и переопределить ссылку с объекта на литерал при помощи вызова метода **intern()**.

// # 4 # занесение в пул литералов # DemoIntern.java

```
package by.bsu.strings;
public class DemoIntern {
          public static void main(String[ ] args) {
                    String s1 = "Java"; // литерал и ссылка на него
                    String s2 = new String("Java");
                    System.out.println(s1 == s2); // false
                    s2 = s2.intern();
                    System.out.println(s1 == s2); // true
          }
}
```

В данной ситуации ссылка **s1** инициализируется литералом, обладающим всеми свойствами объекта вплоть до вызова методов. Вызов метода **intern()** организует поиск в «пуле литералов» соответствующего значению объекта **s2** литерала и при положительном результате возвращает ссылку на найденный литерал, а при отрицательном — заносит значение в пул и возвращает ссылку на него.

Классы StringBuilder и StringBuffer

Классы **StringBuilder** и **StringBuffer** являются «близнецами» и по своему предназначению близки к классу **String**, но в отличие от последнего содержимое и размеры объектов классов **StringBuilder** и **StringBuffer** можно изменять.

Основным отличием **StringBuilder** от **StringBuffer** является потокобезопасность последнего. Более высокая скорость обработки есть следствие отсутствия потокобезопасности класса **StringBuilder**. Его следует применять, если не существует вероятности использования объекта в конкурирующих потоках.

С помощью соответствующих методов и конструкторов объекты классов **StringBuffer**, **StringBuilder** и **String** можно преобразовывать друг в друга. Конструктор класса **StringBuffer** (также как и **StringBuilder**) может принимать в качестве параметра объект **String** или неотрицательный размер буфера. Объекты

этого класса можно преобразовать в объект класса **String** методом **toString()** или с помощью конструктора класса **String**.

Следует обратить внимание на следующие методы:

void setLength(int n) — установка размера буфера;

void ensureCapacity(int minimum) — установка гарантированного минимального размера буфера;

int capacity() — возвращение текущего размера буфера;

StringBuffer append(*параметры***)** — добавление к содержимому объекта строкового представления аргумента, который может быть символом, значением базового типа, массивом и строкой;

StringBuffer insert(*параметры***)** — вставка символа, объекта или строки в указанную позицию;

StringBuffer deleteCharAt(int index) — удаление символа;

StringBuffer delete(int start, int end) — удаление подстроки;

StringBuffer reverse() — обращение содержимого объекта.

В классе присутствуют также методы, аналогичные методам класса **String**, такие как **replace()**, **substring()**, **charAt()**, **length()**, **getChars()**, **indexOf()** и др.

/ # 5 # свойства объекта StringBuffer # DemoStringBuffer.java */*

```
package by.bsu.strings;
public class DemoStringBuffer {
        public static void main(String[] args) {
                StringBuffer sb = new StringBuffer();
                System.out.println("длина –> " + sb.length());
                System.out.println("размер –>" + sb.capacity());
                // sb = "Java"; // ошибка, только для класса String
                sb.append("Java");
                System.out.println("строка –> " + sb);
                System.out.println("длина –> " + sb.length());
                System.out.println("размер –> " + sb.capacity());

                sb.append("Internationalization");
                System.out.println("строка –> " + sb);
                System.out.println("длина –> " + sb.length());
                System.out.println("размер –> " + sb.capacity());

                System.out.println("реверс –> " + sb.reverse());
        }
}
```

Результатом выполнения данного кода будет:

длина —> 0
размер —> 16
строка —> Java
длина —> 4

размер —> 16
строка —> JavaInternationalization
длина —> 24
размер —> 34
реверс —> noitazilanoitanretnIavaJ

При создании объекта **StringBuffer** конструктор по умолчанию автоматически резервирует некоторый объем памяти (16 символов), что в дальнейшем позволяет быстро менять содержимое объекта, оставаясь в границах участка памяти, выделенного под объект. Размер резервируемой памяти при необходимости можно указывать в конструкторе. Если длина строки **StringBuffer** после изменения превышает его размер, то емкость объекта автоматически увеличивается, оставляя при этом некоторый резерв для дальнейших изменений. Методом **reverse()** можно быстро изменить порядок символов в объекте.

Если метод, вызываемый объектом **StringBuilder**, производит изменения в его содержимом, то это не приводит к созданию нового объекта, как в случае объекта **String**, а изменяет текущий объект **StringBuilder**.

/ # 6 # изменение объекта StringBuilder # RefStringBuilder.java */*

```
package by.bsu.strings;
public class RefStringBuilder {
          public static void changeStr(StringBuilder s) {
                    s.append(" Certified");
          }
          public static void main(String[ ] args) {
                    StringBuilder str = new StringBuilder("Oracle");
                    changeStr(str);
                    System.out.println(str);
          }
}
```

В результате выполнения этого кода будет выведена строка:

Oracle Certified

Объект **StringBuilder** передан в метод **changeStr()** по ссылке, поэтому все изменения объекта сохраняются и для вызывающего метода.

Для классов **StringBuffer** и **StringBuilder** не переопределены методы **equals()** и **hashCode()**, т. е. сравнить содержимое двух объектов невозможно, следовательно хэш-коды всех объектов этого типа вычисляются так же, как и для класса **Object**. При идентичном содержимом у двух экземпляров, размеры буфера каждого могут отличаться, поэтому сравнение на эквивалентность объектов представляется неоднозначным.

/ # 7 # сравнение объектов StringBuffer и их хэш-кодов # EqualsStringBuffer.java */*

```
package by.bsu.strings;
public class EqualsStringBuffer {
```

```
        public static void main(String[ ] args) {
                StringBuffer sb1 = new StringBuffer();
                StringBuffer sb2 = new StringBuffer(48);
                sb1.append("Java");
                sb2.append("Java");
                System.out.print(sb1.equals(sb2));
                System.out.print(sb1.hashCode() == sb2.hashCode());
        }
}
```

Результатом выполнения данной программы будет дважды выведенное значение **false**.

Сравнить содержимое можно следующим образом:

```
sb1.toString().contentEquals(sb2)
```

Регулярные выражения

Класс **java.util.regex.Pattern** применяется для определения регулярных выражений (шаблонов), для которых ищется соответствие в строке, файле или другом объекте, представляющем последовательность символов. Для определения шаблона применяются специальные синтаксические конструкции. О каждом соответствии можно получить информацию с помощью класса **java.util.regex.Matcher**.

Далее приведены некоторые логические конструкции для задания шаблона.

Если необходимо, чтобы в строке, проверяемой на соответствие, в какой-либо позиции находился один из символов некоторого символьного набора, то такой набор (класс символов) можно объявить, используя одну из следующих конструкций:

[abc]	**a**, **b** или **c**
[^abc]	символ, исключая **a**, **b** и **c**
[a-z]	символ между **a** и **z**
[a-d[m-p]]	между **a** и **d**, или между **m** и **p**

Кроме стандартных классов символов существуют предопределенные классы символов:

.	любой символ
\d или \p{Digit}	[**0-9**]
\D	[^**0-9**]
\s или \p{Space}	[**\t\n\x0B\f\r**]
\S	[^**\s**]
\w	[**0-9_A-Za-z**]
\W	[^**\w**]

\p{Lower}	[**a-z**]
\p{Upper}	[**A-Z**]
\p{Punkt}	!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
\p{Blank}	Пробел или табуляция

При создании регулярного выражения могут использоваться логические операции:

ab	после **a** следует **b**
a|b	**a** либо **b**
(a)	**a**

Скобки кроме их логического назначения также используются для выделения групп.

Для определения регулярных выражений недостаточно одних классов символов, т. к. в шаблоне часто нужно указать количество повторений. Для этого существуют квантификаторы.

a?	**a** один раз или ни разу
a*	**a** ноль или более раз
a+	**a** один или более раз
a{n}	**a** n раз
a{n,}	**a** n или более раз
a{n,m}	**a** от n до m

Существует еще два типа квантификаторов, которые образованы прибавлением суффикса «?» (слабое или неполное совпадение) или «+» («жадное» или собственное совпадение) к вышеперечисленным квантификаторам. Неполное совпадение соответствует выбору с наименее возможным количеством символов, а собственное — с максимально возможным.

Класс **Pattern** используется для простой обработки строк. Для более сложной обработки строк используется класс **Matcher**, рассматриваемый ниже.

В классе **Pattern** объявлены следующие методы:

Pattern compile(String regex) — возвращает **Pattern**, который соответствует **regex**;

boolean matches(String regex, CharSequence input) — проверяет на соответствие строки **input** шаблону **regex**;

String[] split(CharSequence input) — разбивает строку **input**, учитывая, что разделителем является шаблон;

Matcher matcher(CharSequence input) — возвращает **Matcher**, с помощью которого можно находить соответствия в строке **input**.

С помощью метода **matches()** класса **Pattern** можно проверять на соответствие шаблону целую строку, но если необходимо найти соответствия внутри строки, например, определять участки, которые соответствуют шаблону, то класс **Pattern** не может быть использован. Для таких операций необходимо использовать класс **Matcher**.

Начальное состояние объекта типа **Matcher** не определено. Попытка вызвать какой-либо метод класса для извлечения информации о найденном соответствии приведет к возникновению ошибки **IllegalStateException**. Для того, чтобы начать работу с объектом **Matcher**, нужно вызвать один из его методов:

boolean matches() — проверяет, соответствует ли вся информация шаблону;

boolean lookingAt() — поиск последовательности символов, начинающейся с начала строки и соответствующей шаблону;

boolean find() или **boolean find(int start)** — ищет последовательность символов, соответствующих шаблону, в любом месте строки. Параметр **start** указывает на начальную позицию поиска.

Иногда необходимо сбросить состояние экземпляра **Matcher** в исходное, для этого применяется метод **reset()** или **reset(CharSequence input)**, который также устанавливает новую последовательность символов для поиска.

Для замены всех подпоследовательностей символов, удовлетворяющих шаблону, на заданную строку можно применить метод **replaceAll(String replacement)**.

В регулярном выражении для более удобной обработки входной последовательности применяются группы, которые помогают выделить части найденной подпоследовательности. В шаблоне они обозначаются скобками «**(**» и «**)**». Номера групп начинаются с единицы. Нулевая группа совпадает со всей найденной подпоследовательностью. Далее приведены методы для извлечения информации о группах.

String group() — возвращает всю подпоследовательность, удовлетворяющую шаблону;

int start() — возвращает индекс первого символа подпоследовательности, удовлетворяющей шаблону;

int start(int group) — возвращает индекс первого символа указанной группы;

int end() — возвращает индекс последнего символа подпоследовательности, удовлетворяющей шаблону;

int end(int group) — возвращает индекс последнего символа указанной группы;

String group(int group) — возвращает конкретную группу по позиции;

boolean hitEnd() — возвращает истину, если был достигнут конец входной последовательности.

Следующий пример показывает, как можно использовать возможности классов **Pattern** и **Matcher** для поиска, разбора и разбивки строк.

/ # 8 # обработка строк с помощью шаблонов # DemoRegular.java */*

```
package by.bsu.regex;
import java.util.regex.Pattern;
```

```
import java.util.regex.Matcher;
import java.util.Arrays;
public class DemoRegular {
          public static void main(String[ ] args) {
                    // проверка на соответствие строки шаблону
                    Pattern p1 = Pattern.compile("a+y");
                    Matcher m1 = p1.matcher("aaay");
                     boolean b = m1.matches();
                    System.out.println(b);
                    // поиск и выбор подстроки, заданной шаблоном
                    String regex = "(\\w{6,})@(\\w+\\.)([a-z]{2,4})";
                    String s = "адреса эл.почты:blinov@gmail.com, romanchik@bsu.by!";
                    Pattern p2 = Pattern.compile(regex);
                    Matcher m2 = p2.matcher(s);
                    while (m2.find()) {
                              System.out.println("e-mail: " + m2.group());
                    }
                    // разбиение строки на подстроки с применением шаблона в качестве разделителя
                    Pattern p3 = Pattern.compile("\\d+\\s?");
                    String[ ] words = p3.split("java5tiger 77 java6mustang");
                    System.out.print(Arrays.toString(words));
          }
}
```

В результате будет выведено:

true
e-mail: blinov@gmail.com
e-mail: romanchik@bsu.by
[java, tiger, java, mustang]

Использование групп, а также собственных и неполных квантификаторов.

/ # 9 # группы и квантификаторы # Groups.java */*

```
package by.bsu.regex;
public class Groups {
          public static void main(String[ ] args) {
                    String input = "abdcxyz";
                        simpleMatches ("([a-z]*)([a-z]+)", input);
                        simpleMatches ("([a-z]?)([a-z]+)", input);
                        simpleMatches ("([a-z]+)([a-z]*)", input);
                        simpleMatches ("([a-z]?)([a-z]?)", input);
          }
          public static void simpleMatches(String regex, String input) {
                        Pattern pattern = Pattern.compile(regex);
                        Matcher matcher = pattern.matcher(input);
                    if(matcher.matches()) {
                        System.out.println("First group: " + matcher.group(1));
                        System.out.println("Second group: " + matcher.group(2)+ "\n");
```

```
                } else {
                    System.out.println("nothing\n");
                }
        }
}
```

Результат работы программы:

First group: abdcxy
Second group: z

First group: a
Second group: bdcxyz

First group: abdcxyz
Second group:

nothing

В первом случае к первой группе относятся все возможные символы, но при этом остается минимальное количество символов для второй группы.

Во втором случае для первой группы выбирается наименьшее количество символов, т. к. используется слабое совпадение.

В третьем случае первой группе будет соответствовать вся строка, а для второй не остается ни одного символа, так как вторая группа использует слабое совпадение.

В четвертом случае строка не соответствует регулярному выражению, т. к. для двух групп выбирается наименьшее количество символов.

Интернационализация приложения

Класс **java.util.Locale** позволяет учесть особенности региональных представлений алфавита, символов, чисел, дат и проч. Автоматически виртуальная машина использует текущие региональные установки операционной системы, но при необходимости их можно изменять. Для некоторых стран региональные параметры устанавливаются с помощью констант, например: **Locale.US**, **Locale.FRANCE**. Для всех остальных объект **Locale** нужно создавать с помощью конструктора, например:

```
Locale rus = new Locale("ru", "RU");
```

Определить текущий вариант региональных параметров можно следующим образом:

```
Locale current = Locale.getDefault();
```

А можно и изменить для текущего экземпляра (instance) JVM:

```
Locale.setDefault(Locale.CANADA);
```

Если, например, в ОС установлен регион «Беларусь» или в приложении с помощью **new Locale("be", "BY")**, то следующий код (при выводе результатов выполнения на консоль)

```
current.getCountry(); // код региона
current.getDisplayCountry(); // название региона
current.getLanguage(); // код языка региона
current.getDisplayLanguage(); // название языка региона
```

позволяет получить информацию о регионе в виде:

BY
Беларусь
be
белорусский

Для создания приложений, поддерживающих несколько языков, существует целый ряд решений. Самое логичное из них — дублирование сообщений на разных языках в разных файлах с эквивалентными ключами с последующим извлечением информации на основе значения заданной локали. Данное решение основано на взаимодействии классов **java.util.ResourceBundle** и **Locale**. Класс **ResourceBundle** предназначен для взаимодействия с текстовыми файлами свойств (расширение.**properties**). Каждый объект **ResourceBundle** представляет собой набор соответствующих подтипов, которые разделяют одно и то же базовое имя, к которому можно получить доступ через поле **parent**. Следующий список показывает возможный набор соответствующих ресурсов с базовым именем **text**. Символы, следующие за базовым именем, показывают код языка, код страны и тип операционной системы. Например, файл **text_it_CH.properties** соответствует объекту **Locale**, заданному кодом итальянского языка (**it**) и кодом страны Швейцарии (**CH**).

```
text.properties
text_ru_RU.properties
text_it_CH.properties
text_fr_CA.properties
```

Чтобы выбрать определенный объект **ResourceBundle**, следует вызвать один из статических перегруженных методов **getBundle(***параметры***)**. Следующий фрагмент выбирает **text** объекта **ResourceBundle** для объекта **Locale**, который соответствует французскому языку и стране Канада.

```
Locale locale = new Locale("fr", "CA");
ResourceBundle rb = ResourceBundle.getBundle("text", locale);
```

Если объект **ResourceBundle** для заданного объекта **Locale** не существует, то метод **getBundle()** извлечет наиболее общий. Если общее определение файла ресурсов не задано, то метод **getBundle()** генерирует исключительную ситуацию **MissingResourceException**. Чтобы это не произошло, необходимо обеспечить

наличие базового файла ресурсов без суффиксов, а именно: **text.properties** в дополнение к частным случаям вида:

```
text_en_US.properties
text_ru_RU.properties
```

В файлах свойств информация должна быть организована по принципу:

```
#Комментарий
group1.key1 = value1
group1.key2 = value2
group2.key1 = value3…
```

Например:

```
label.button = submit
label.field = login
message.welcome = Welcome!
```

или

```
label.button = принять
label.field = логин
message.welcome = Добро пожаловать!
```

В классе **ResourceBundle** определен ряд полезных методов, в том числе метод **getKeys(),** возвращающий объект **Enumeration**, который применяется для последовательного обращения к элементам. Множество **Set<String>** всех ключей — методом **keySet()**. Конкретное значение по конкретному ключу извлекается методом **getString(String key)**. Отсутствие запрашиваемого ключа приводит к генерации исключения. Проверить наличие ключа в файле можно методом **boolean containsKey(String key)**. Методы **getObject(String key)** и **getStringArray(String key)** извлекают соответственно объект и массив строк по передаваемому ключу.

В следующем примере в зависимости от выбора пользователя известная фраза будет выведена на одном из трех языков.

// # 10 # поддержка различных языков # HamletInternational.java

```
package by.bsu.resource;
import java.io.IOException;
import java.util.Locale;
import java.util.ResourceBundle;
public class HamletInternational {
          public static void main(String[ ] args) {
              for (int i = 0; i < 3: i++) {
                    System.out.println("1 – английский /n 2 – белорусский \n любой – русский ");
                    char i = 0;
                    try {
                              i = (char) System.in.read();
```

```
                } catch (IOException e) {
                        e.printStackTrace();
                }
                String country = "";
                String language = "";
                switch (i) {
                    case '1':
                          country = "US";
                          language = "EN";
                          break;
                    case '2':
                          country = "BY";
                          language = "BE";
                          break;
                }
                Locale current = new Locale(language, country);
                ResourceBundle rb = ResourceBundle.getBundle("property.text", current);
                          String s1 = rb.getString("str1");
                          System.out.println(s1);

                          String s2 = rb.getString("str2");
                          System.out.println(s2);
                    }
        }
}
```

Все файлы следует разместить в каталоге **property** в корне проекта на одном уровне с пакетами приложения.

Файл **text_en_US.properties** содержит следующую информацию:

str1 = To be or not to be?
str2 = This is a question.

Файл **text_be_BY.properties**:

str1 = Быць або не быць?
str2 = Вось у чым пытанне.

Файл **text.properties**:

str1 = Быть или не быть?
str2 = Вот в чём вопрос.

Если в результате выполнения приложения на консоль результаты выдаются в нечитаемом виде, то следует изменить кодировку файла или символов.

Для взаимодействия с **properties**-файлами можно создать специальный класс, экземпляр которого позволит не только извлекать информацию по ключу, но и изменять значение локали, что сделает его удобным для использования при интернационализации приложений.

// # 11 # менеджер ресурсов # ResourceManager.java

```
package by.bsu.resource;
import java.util.Locale;
import java.util.ResourceBundle;
public enum ResourceManager {
          INSTANCE;
          private ResourceBundle resourceBundle;
          private final String resourceName = "property.text";
          private ResourceManager() {
                 resourceBundle = ResourceBundle.getBundle(resourceName, Locale.getDefault());
          }
          public void changeResource(Locale locale) {
                 resourceBundle = ResourceBundle.getBundle(resourceName, locale);
          }
          public String getString(String key) {
                 return resourceBundle.getString(key);
          }
}
```

Экземпляр класса может быть создан только один, и все приложение пользуется его возможностями.

/ # 12 # извлечение информации из файла ресурсов и смена локали # ResourceManagerRun.java */*

```
package by.bsu.resource;
import java.util.Locale;
public class ResourceManagerRun {
public static void main(String[ ] args) {
                    ResourceManager manager = ResourceManager.INSTANCE;
                    System.out.println(manager.getString("str1"));

                    manager.changeResource(new Locale("be","BY"));
                    System.out.println(manager.getString("str1"));
          }
}
```

Качественно разработанное приложение обычно не содержит литералов типа **String**. Все необходимые сообщения хранятся вне системы, в частности, в **properties** файлах. Что позволяет без перекомпиляции кода безболезненно изменять любое сообщение или информацию, хранящуюся вне классов системы.

Интернационализация чисел

Стандарты представления дат и чисел в различных странах могут существенно отличаться. Например, в Германии строка **«1.234,567»** воспринимается как «одна тысяча двести тридцать четыре целых пятьсот шестьдесят семь

тысячных», для русских и французов данная строка просто непонятна и не может представлять число.

Чтобы сделать такую информацию конвертируемой в различные региональные стандарты, применяются возможности класса **java.text.NumberFormat**. Первым делом следует задать или получить текущий объект **Locale** с шаблонами регионального стандарта и создать с его помощью объект форматирования **NumberFormat**. Например:

```
NumberFormat nf = NumberFormat.getInstance(new Locale("RU"));
```

с конкретными региональными установками или с установленными по умолчанию для приложения:

```
NumberFormat.getInstance();
```

Далее для преобразования строки в число и обратно используются методы **Number parse(String source)** и **String format(double number)** соответственно.

В предлагаемом примере производится преобразование строки, содержащей число, в три различных региональных стандарта, а затем одно из чисел преобразуется из одного стандарта в два других.

// # 13 # региональные представления чисел # DemoNumberFormat.java

```
package by.bsu.num;
import java.text.*;
import java.util.Locale;
public class DemoNumberFormat {
          public static void main(String args[ ]) {
                    NumberFormat nfGe = NumberFormat.getInstance(Locale.GERMAN);
                    NumberFormat nfUs = NumberFormat.getInstance(Locale.US);
                    NumberFormat nfFr = NumberFormat.getInstance(Locale.FRANCE);
                    double iGe = 0, iUs = 0, iFr = 0;
                    String str = "1.234,5"; // строка, представляющая число
                    try {
                    // преобразование строки в германский стандарт
                              iGe = nfGe.parse(str).doubleValue();
                    // преобразование строки в американский стандарт
                              iUs = nfUs.parse(str).doubleValue();
                    // преобразование строки во французский стандарт
                              iFr = nfFr.parse(str).doubleValue();
                    } catch (ParseException e) {
                              System.err.print("Error position: " + e.getErrorOffset());
                    }
          System.out.printf("iGe = %f\niUs = %f\niFr = %f", iGe, iUs, iFr);

          // преобразование числа из германского в американский стандарт
          String sUs = nfUs.format(iGe);
          // преобразование числа из германского во французский стандарт
          String sFr = nfFr.format(iGe);
```

```
        System.out.println("\nUs " + sUs + "\nFr " + sFr);
        }
}
```

Результат работы программы:

iGe = 1234,500000
iUs = 1,234000
iFr = 1,000000
Us 1,234.5
Fr 1 234,5

Аналогично выполняются переходы от одного регионального стандарта к другому при отображении денежных сумм с добавлением символа валюты.

Интернационализация дат

Учитывая исторически сложившиеся способы отображения даты и времени в различных странах и регионах мира, в языке создан механизм поддержки всех национальных особенностей. Эту задачу решает класс **java.text.DateFormat**. С его помощью учтены: необходимость представления месяцев и дней недели на национальном языке; специфические последовательности в записи даты и часовых поясов; возможности использования различных календарей.

Процесс получения объекта, отвечающего за обработку регионального стандарта даты, похож на создание объекта, отвечающего за национальные представления чисел, а именно:

```
DateFormat df = DateFormat.getDateInstance(DateFormat.MEDIUM, new Locale("BY"));
```

или по умолчанию:

```
DateFormat.getDateInstance();
```

Константа **DateFormat.MEDIUM** указывает на то, что будут представлены только дата и время без указания часового пояса. Для указания часового пояса используются константы класса **DateFormat** со значением **LONG** и **FULL**. Константа **SHORT** применяется для сокращенной записи даты, где месяц представлен в виде своего порядкового номера.

Для получения даты в виде строки для заданного региона используется метод **String format(Date date)** в виде:

```
String s = df.format(new Date());
```

Метод **Date parse(String source)** преобразовывает переданную в виде строки дату в объектное представление конкретного регионального формата, например:

```
String str = "April 9, 2012";
Date d = df.parse(str);
```

Класс **DateFormat** содержит большое количество методов, позволяющих выполнять разнообразные манипуляции с датой и временем.

В качестве примера рассмотрено преобразование заданной даты в различные региональные форматы.

// # 14 # региональные представления дат # DemoDateFormat.java

```
package by.bsu.dates;
import java.text.DateFormat;
import java.text.ParseException;
import java.util.*;
public class DemoDateFormat {
          public static void main(String[] args) {
                    DateFormat df = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.US);
                    Date d = null;
                    String str = "April 9, 2012";
                       try {
                              d = df.parse(str);
                              System.out.println(d);
                       } catch (ParseException e) {
                              System.err.print("Error position: " + e.getErrorOffset());
                       }
                       df = DateFormat.getDateInstance(DateFormat.LONG, new Locale("ru","RU"));
                       System.out.println(df.format(d));

                       df = DateFormat.getDateInstance(DateFormat.FULL,Locale.GERMAN);
                       System.out.println(df.format(d));
}
```

Результат работы программы:

Mon Apr 9 00:00:00 EEST 2012
9 Апрель 2012 г.
Montag, 9. April 2012

Чтобы получить представление текущей даты во всех возможных региональных стандартах, можно воспользоваться следующим фрагментом кода:

```
Date d = new Date();
Locale[ ] locales = DateFormat.getAvailableLocales();
          for (Locale loc : locales) {
                    DateFormat df = DateFormat.getDateInstance(DateFormat.FULL, loc);
                    System.out.println(loc.toString() + "---> " + df.format(d));
          }
```

В результате будет выведена пара сотен строк, каждая из которых представляет текущую дату в соответствии с региональным стандартом, выводимым перед датой с помощью инструкции **loc.toString()**.

Форматирование строк

Для создания форматированного текстового вывода предназначен класс **java.util.Formatter**. Этот класс обеспечивает преобразование формата, позволяющее выводить числа, строки, время и даты в любом необходимом разработчику виде.

В классе **Formatter** объявлен метод **format()**, который преобразует переданные в него параметры в строку заданного формата и сохраняет в объекте типа **Formatter**. Аналогичный метод объявлен у классов **PrintStream** и **PrintWriter**. Кроме того у этих классов объявлен метод **printf()** с параметрами, идентичными параметрам метода **format()**, который осуществляет форматированный вывод в поток, тогда как метод **format()** сохраняет изменения в объекте типа **Formatter**. Таким образом, метод **printf()** автоматически использует возможности класса **Fomatter** и подобен функции **printf()** языка C.

Класс **Formatter** преобразует двоичную форму представления данных в форматированный текст. Он сохраняет форматированный текст в буфере, содержимое которого можно получить в любой момент. Можно предоставить классу **Formatter** автоматическую поддержку этого буфера либо задать его явно при создании объекта. Существует возможность сохранения буфера класса **Formatter** в файле.

Для создания объекта класса существует более десяти конструкторов. Ниже приведены наиболее употребляемые:

```
Formatter()
Formatter(Appendable buf)
Formatter(Appendable buf, Locale loc)
Formatter(String filename) throws FileNotFoundException
Formatter(String filename, String charset) throws FileNotFoundException,
                                                  UnsupportedEncodingException
Formatter(File outF) throws FileNotFoundException
Formatter(OutputStream outStrm)
Formatter(PrintStream printStrm)
```

В приведенных образцах **buf** задает буфер для форматированного вывода. Если параметр **buf** равен **null**, класс **Formatter** автоматически размещает объект типа **StringBuilder** для хранения форматированного вывода. Параметр **loc** определяет региональные и языковые настройки. Если никаких настроек не задано, используются настройки по умолчанию. Параметр **filename** задает имя файла, который получит форматированный вывод. Параметр **charset** определяет кодировку. Если она не задана, используется кодировка, установленная по умолчанию. Параметр **outF** передает ссылку на открытый файл, в котором будет храниться форматированный вывод. В параметре **outStrm** передается ссылка на поток вывода, который будет получать отформатированные данные. Если используется файл, выходные данные записываются в файл.

Некоторые методы класса:

Formatter format(Locale loc, String fmtString, Object…args) — форматирует аргументы, переданные в аргументе переменной длины **args**, в соответствии со спецификаторами формата, содержащимися в **fmtString**. При форматировании используются региональные установки, заданные в **loc**. Возвращает вызывающий объект. Существует перегруженная версия метода без использования локализации;

Locale locale() — возвращает региональные установки вызывающего объекта;

Appendable out() — возвращает ссылку на базовый объект-приемник для выходных данных;

void flush() — переносит информацию из буфера форматирования и производит запись в указанное место выходных данных, находящихся в буфере. Метод чаще всего используется объектом класса **Formatter**, связанным с файлом;

void close() — закрывает вызывающий объект класса **Formatter**, что приводит к освобождению ресурсов, используемых объектом. После закрытия объекта типа **Formatter** он не может использоваться повторно. Попытка использовать закрытый объект приводит к генерации исключения типа **FormatterClosedException**.

При форматировании используются спецификаторы формата:

Спецификатор формата	Выполняемое форматирование
%a	Шестнадцатеричное значение с плавающей точкой
%b	Логическое (булево) значение аргумента
%c	Символьное представление аргумента
%d	Десятичное целое значение аргумента
%h	Хэш-код аргумента
%e	Экспоненциальное представление аргумента
%f	Десятичное значение с плавающей точкой
%g	Выбирает более короткое представление из двух: **%e** или **%f**
%o	Восьмеричное целое значение аргумента
%n	Вставка символа новой строки
%s	Строковое представление аргумента
%t	Время и дата
%x	Шестнадцатеричное целое значение аргумента
%%	Вставка знака **%**

Также возможны спецификаторы с заглавными буквами: **%A** (эквивалентно **%a**). Форматирование с их помощью обеспечивает перевод символов в верхний регистр.

/ # 15 # форматирование строки при помощи метода format() # SimpleFormatString.java */*

```
package by.bsu.format;
import java.util.Formatter;
public class SimpleFormatString {
          public static void main(String[ ] args){
                    Formatter f = new Formatter(); // объявление объекта
                    // форматирование текста по формату %S, %c
                    f.format("This %s is about %n%S %c", "book", "java", '8');
                    System.out.print(f);
          }
}
```

В результате выполнения этого кода будет выведено:

This book is about
JAVA 8

/ # 16 # форматирование чисел с использованием спецификаторов %x, %o, %a, %g # FormatterDemoNumber.java */*

```
package by.bsu.format;
import java.util.Formatter;
public class FormatterDemoNumber {
          public static void main(String[] args) {
                    Formatter f = new Formatter();
                    f.format("Hex: %x, Octal: %o", 100, 100);
                    System.out.println(f);
                    f = new Formatter();
                    f.format("%a", 100.001);
                    System.out.println(f);
                    f = new Formatter();
                    for (double i = 1000; i < 1.0e+10; i *= 100) {
                              f.format("%g ", i);
                              System.out.println(f);
                    }
          }
}
```

В результате выполнения этого кода будет выведено:

Hex: 64, Octal: 144
0x1.90010624dd2f2p6
1000.00
1000.00 100000
1000.00 100000 1.00000e+07
1000.00 100000 1.00000e+07 1.00000e+09

Все спецификаторы для форматирования даты и времени могут употребляться только для типов **long**, **Long**, **Calendar**, **Date**.

В таблице приведены некоторые из спецификаторов формата времени и даты.

Спецификатор формата	Выполняемое преобразование
%tH	Час (00–23)
%tI	Час (1–12)
%tM	Минуты как десятичное целое (00–59)
%tS	Секунды как десятичное целое (00–59)
%tL	Миллисекунды (000–999)
%tY	Год в четырехзначном формате
%ty	Год в двузначном формате (00–99)
%tB	Полное название месяца («январь»)
%tb или **%th**	Краткое название месяца («янв»)
%tm	Месяц в двузначном формате (1–12)
%tA	Полное название дня недели («пятница»)
%ta	Краткое название дня недели («пт»)
%td	День в двузначном формате (1–31)
%tR	То же, что и **"%tH:%tM"**
%tT	То же, что и **"%tH:%tM:%tS"**
%tr	То же, что и **"%tI:%tM:%tS %Tp"** где **%Tp** = (AM или PM)
%tD	То же, что и **"%tm/%td/%ty"**
%tF	То же, что и **"%tY–%tm–%td"**
%tc	То же, что и **"%ta %tb %td %tT %tZ %tY"**

/ # 17 # форматирование даты и времени # FormatterDemoTimeAndDate.java */*

```
package by.bsu.format;
import java.util.*;
public class FormatterDemoTimeAndDate {
          public static void main(String args[]) {
                    Formatter f = new Formatter();
                    Calendar cal = Calendar.getInstance();

                    // вывод в 12-часовом временном формате
                    f.format("%tr", cal);
                    System.out.println(f);

                    // полноформатный вывод времени и даты
                    f = new Formatter();
                    f.format("%tc", cal);
                    System.out.println(f);

                    // вывод текущего часа и минуты
                    f = new Formatter();
                    f.format("%tl:%tM", cal, cal);
                    System.out.println(f);
```

```
                    // всевозможный вывод месяца
                    f = new Formatter();
                    f.format("%tB %tb %tm", cal, cal, cal);
                    System.out.println(f);
          }
}
```

В результате выполнения этого кода будет выведено:

```
03:28:08 PM
Пт янв 06 15:28:08 EET 2006
3:28
Январь янв 01
```

Спецификатор точности применяется только в спецификаторах формата **%f**, **%e**, **%g** для данных с плавающей точкой и в спецификаторе **%s** — для строк. Он задает количество выводимых десятичных знаков или символов. Например, спецификатор **%10.4f** выводит число с минимальной шириной поля 10 символов и с четырьмя десятичными знаками. Принятая по умолчанию точность равна шести десятичным знакам.

Примененный к строкам спецификатор точности задает максимальную длину поля вывода. Например, спецификатор **%5.7s** выводит строку длиной не менее пяти и не более семи символов. Если строка длиннее, конечные символы отбрасываются.

Ниже приведен пример использования флагов форматирования.

/ # 18 # применение флагов форматирования # FormatterDemoFlags.java */*

```
package by.bsu.format;
import java.util.*;
public class FormatterDemoFlags {
          public static void main(String[] args) {
                    Formatter f = new Formatter();
                    // выравнивание вправо
                    f.format("|%10.2f|", 123.123);
                    System.out.println(f);

                    // выравнивание влево
                    // применение флага '-'
                    f = new Formatter();
                    f.format("|%-10.2f|", 123.123);
                    System.out.println(f);

                    f = new Formatter();
                    f.format("% (d", –100);
                    // применение флага ' ' и '('
                    System.out.println(f);

                    f = new Formatter();
                    f.format("%,.2f", 123456789.34);
```

```
                // применение флага ','
                System.out.println(f);

                f = new Formatter();
                f.format("%.4f", 1111.1111111);
                // задание точности представления для чисел
                System.out.println(f);

                f = new Formatter();
                f.format("%.16s", "Now I know class java.util.Formatter");
                // задание точности представления для строк
                System.out.println(f);
        }
}
```

В результате выполнения этого кода будет выведено:

```
|   123,12|
|123,12    |
(100)
123 456 789,34
1111,1111
Now I know class
```

Задания к главе 7

Вариант А

1. В каждом слове текста *k*-ю букву заменить заданным символом. Если *k* больше длины слова, корректировку не выполнять.
2. В тексте каждую букву заменить ее порядковым номером в алфавите. При выводе в одной строке печатать текст с двумя пробелами между буквами, в следующей строке внизу под каждой буквой печатать ее номер.
3. В тексте после буквы Р, если она не последняя в слове, ошибочно напечатана буква А вместо О. Внести исправления в текст.
4. В тексте после *k*-го символа вставить заданную подстроку.
5. После **каждого** слова текста, заканчивающегося заданной подстрокой, вставить указанное слово.
6. В зависимости от признака (0 или 1) в каждой строке текста удалить указанный символ везде, где он встречается, или вставить его после *k*-го символа.
7. Из текста удалить все символы, кроме пробелов, не являющиеся буквами. Между последовательностями подряд идущих букв оставить хотя бы один пробел.
8. Удалить из текста его часть, заключенную между двумя символами, которые вводятся (например, между скобками «(» и «)» или между звездочками «*» и т. п.).
9. Определить, сколько раз повторяется в тексте каждое слово, которое встречается в нем.

10. В тексте найти и напечатать *n* символов (и их количество), встречающихся наиболее часто.
11. Найти, каких букв, гласных или согласных, больше в каждом предложении текста.
12. В стихотворении найти количество слов, начинающихся и заканчивающихся гласной буквой.
13. Напечатать без повторения слова текста, у которых первая и последняя буквы совпадают.
14. В тексте найти и напечатать все слова максимальной и все слова минимальной длины.
15. Напечатать квитанцию об оплате телеграммы, если стоимость одного слова задана.
16. В стихотворении найти одинаковые буквы, которые встречаются во всех словах.
17. В тексте найти первую подстроку максимальной длины, не содержащую букв.
18. В тексте определить все согласные буквы, встречающиеся не более чем в двух словах.
19. Преобразовать текст так, чтобы каждое слово, не содержащее неалфавитных символов, начиналось с заглавной буквы.
20. Подсчитать количество содержащихся в данном тексте знаков препинания.
21. В заданном тексте найти сумму всех встречающихся цифр.
22. Из кода Java удалить все комментарии (//, /*, /**).
23. Все слова текста встречаются четное количество раз, за исключением одного. Определить это слово. При сравнении слов регистр не учитывать.
24. Определить сумму всех целых чисел, встречающихся в заданном тексте.
25. Из текста удалить все лишние пробелы, если они разделяют два различных знака препинания и если рядом с ними находится еще один пробел.
26. Строка состоит из упорядоченных чисел от 0 до 100000, записанных подряд без пробелов. Определить, что будет подстрокой от позиции *n* до *m*.
27. Определить количество вхождений заданного слова в текст, игнорируя регистр символов и считая буквы «е», «ё», и «и», «й» одинаковыми.
28. Преобразовать текст так, чтобы только первые буквы каждого предложения были заглавными.
29. Заменить все одинаковые рядом стоящие в тексте символы одним символом.
30. Вывести в заданном тексте все слова, расположив их в алфавитном порядке.
31. Подсчитать, сколько слов в заданном тексте начинается с прописной буквы.
32. Подсчитать, сколько раз заданное слово входит в текст.

Вариант В

Создать программу обработки текста учебника по программированию с использованием классов: *Символ*, *Слово*, *Предложение*, *Абзац*, *Лексема*, *Листинг*, *Знак препинания* и др. Во всех задачах с формированием текста заменять табуляции и последовательности пробелов одним пробелом.

Предварительно текст следует разобрать на составные части, выполнить одно из перечисленных ниже заданий и вывести полученный результат.

1. Найти наибольшее количество предложений текста, в которых есть одинаковые слова.
2. Вывести все предложения заданного текста в порядке возрастания количества слов в каждом из них.
3. Найти такое слово в первом предложении, которого нет ни в одном из остальных предложений.
4. Во всех вопросительных предложениях текста найти и напечатать без повторений слова заданной длины.
5. В каждом предложении текста поменять местами первое слово с последним, не изменяя длины предложения.
6. Напечатать слова текста в алфавитном порядке по первой букве. Слова, начинающиеся с новой буквы, печатать с красной строки.
7. Рассортировать слова текста по возрастанию доли гласных букв (отношение количества гласных к общему количеству букв в слове).
8. Слова текста, начинающиеся с гласных букв, рассортировать в алфавитном порядке по первой согласной букве слова.
9. Все слова текста рассортировать по возрастанию количества заданной буквы в слове. Слова с одинаковым количеством расположить в алфавитном порядке.
10. Существует текст и список слов. Для каждого слова из заданного списка найти, сколько раз оно встречается в каждом предложении, и рассортировать слова по убыванию общего количества вхождений.
11. В каждом предложении текста исключить подстроку максимальной длины, начинающуюся и заканчивающуюся заданными символами.
12. Из текста удалить все слова заданной длины, начинающиеся на согласную букву.
13. Отсортировать слова в тексте по убыванию количества вхождений заданного символа, а в случае равенства — по алфавиту.
14. В заданном тексте найти подстроку максимальной длины, являющуюся палиндромом, т. е. читающуюся слева направо и справа налево одинаково.
15. Преобразовать каждое слово в тексте, удалив из него все следующие (предыдущие) вхождения первой (последней) буквы этого слова.
16. В некотором предложении текста слова заданной длины заменить указанной подстрокой, длина которой может не совпадать с длиной слова.

Вариант С

1. Текст из n^2 символов шифруется по следующему правилу:

   — все символы текста записываются в квадратную таблицу размерности n в порядке слева направо, сверху вниз;

   — таблица поворачивается на 90° по часовой стрелке;

— 1-я строка таблицы меняется местами с последней, 2-я — с предпоследней и т. д.

— 1-й столбец таблицы меняется местами со 2-м, 3-й — с 4-м и т. д.

— зашифрованный текст получается в результате обхода результирующей таблицы по спирали по часовой стрелке, начиная с левого верхнего угла.

Зашифровать текст по указанному правилу.

2. Исключить из текста подстроку максимальной длины, начинающуюся и заканчивающуюся одним и тем же символом.
3. Вычеркнуть из текста минимальное количество предложений так, чтобы у любых двух оставшихся предложений было хотя бы одно общее слово.
4. Осуществить сжатие английского текста, заменив каждую группу из двух или более рядом стоящих символов на один символ, за которым следует количество его вхождений в группу. К примеру, строка hellowoooorld должна сжиматься в hel2owo4rld.
5. Распаковать текст, сжатый по правилу из предыдущего задания.
6. Определить, удовлетворяет ли имя файла маске. Маска может содержать символы «?» (произвольный символ) и «*» (произвольное количество произвольных символов).
7. Буквенная запись телефонных номеров основана на том, что каждой цифре соответствует несколько английских букв: 2 — ABC, 3 — DEF, 4 — GHI, 5 — JKL, 6 — MNO, 7 — PQRS, 8 — TUV, 9 — WXYZ. Написать программу, которая находит в заданном телефонном номере подстроку максимальной длины, соответствующую слову из словаря.
8. Осуществить форматирование заданного текста с выравниванием по левому краю. Программа должна разбивать текст на строки с длиной, не превосходящей заданного количества символов. Если очередное слово не помещается в текущей строке, его необходимо переносить на следующую.
9. Изменить программу из предыдущего примера так, чтобы она осуществляла форматирование с выравниванием по обоим краям. Для этого добавить дополнительные пробелы между словами.
10. Добавить к программе из предыдущего примера возможность переноса слов по слогам. Предполагается, что есть доступ к словарю, в котором для каждого слова указано, как оно разбивается на слоги.
11. Пусть текст содержит миллион символов и необходимо сформировать из них строку путем конкатенации. Определить время работы кода. Ускорить процесс, используя класс StringBuilder.
12. Алгоритм Барроуза — Уиллера для сжатия текстов основывается на преобразовании Барроуза — Уиллера. Оно производится следующим образом: для слова рассматриваются все его циклические сдвиги, которые затем сортируются в алфавитном порядке, после чего формируется слово из последних символов отсортированных циклических сдвигов. К примеру, для слова

JAVA циклические сдвиги — это JAVA, AVAJ, VAJA, AJAV. После сортировки по алфавиту получим AJAV, AVAJ, JAVA, VAJA. Значит, результат преобразования — слово VJAA. Реализовать программно преобразование Барроуза — Уиллера для данного слова.

13. Восстановить слово по его преобразованию Барроуза — Уиллера. К примеру, получив на вход VJAA, в результате работы программа должна выдать слово JAVA.
14. В Java код добавить корректные getter и setter-методы для всех полей данного класса при их отсутствии.
15. В тексте нет слов, начинающихся одинаковыми буквами. Напечатать слова текста в таком порядке, чтобы последняя буква каждого слова совпадала с первой буквой следующего слова. Если все слова нельзя напечатать в таком порядке, найти такую цепочку, состоящую из наибольшего количества слов.
16. Текст шифруется по следующему правилу: из исходного текста выбирается 1, 4, 7, 10-й и т. д. (до конца текста) символы, затем 2, 5, 8, 11-й и т. д. (до конца текста) символы, затем 3, 6, 9, 12-й и т. д. Зашифровать заданный текст.
17. В предложении из *n* слов первое слово поставить на место второго, второе — на место третьего и т. д., (*n*-1)-е слово — на место *n*-го, *n*-е слово поставить на место первого. В исходном и преобразованном предложениях между словами должны быть или один пробел, или знак препинания и один пробел.
18. Все слова текста рассортировать в порядке убывания их длин, при этом все слова одинаковой длины рассортировать в порядке возрастания в них количества гласных букв.

Тестовые задания к главе 7

Вопрос 7.1.

Дан код:

```
String s1 = "Minsk";
String s2 = new String("Minsk");
if(s1.equals(s2.intern())){
        System.out.print("true");
} else {
        System.out.print("false");
}
if(s1 == s2){
        System.out.print("true");
```

```
} else{
        System.out.print("false");
}
```

В результате при компиляции и запуске будет выведено (1):

1) truefalse
2) falsetrue
3) truetrue
4) falsefalse
5) ошибка компиляции: заданы некорректные параметры для метода equals()

Вопрос 7.2.

Дан код:

```
StringBuilder sb1 = new StringBuilder("I like Java.");//1
StringBuilder sb2 = new StringBuilder(sb1);//2
if (sb1.equals(sb2)){
        System.out.println("true");
} else {
        System.out.println("false");
}
```

В результате при компиляции и запуске будет выведено (1):

1) true
2) false
3) ошибка компиляции в строке 1
4) ошибка компиляции в строке 2

Вопрос 7.3.

Что будет результатом компиляции и выполнения следующего кода?

```
Pattern p = Pattern.compile("(1*)0");
Matcher m = p.matcher("111110");
System.out.println(m.group(1));
```

1) вывод на консоль строки «111110»;
2) вывод на консоль строки «11111»;
3) ошибка компиляции;
4) ошибка выполнения.

Вопрос 7.4.

Что выведется на консоль при компиляции и выполнении следующих строчек кода (1)?

```
Locale loc = new Locale("ru", "RU");
System.out.println(loc.getDisplayCountry(Locale.US));
```

1) United States;
2) Соединенные Штаты;
3) Россия;
4) Russia;
5) ошибка компиляции.

Вопрос 7.5.

Укажите, какому пакету принадлежат классы, позволяющие форматировать числа и даты: NumberFormat и DateFormat.

1) java.text
2) java.util.text
3) java.util
4) java.lang