# Часть 2.
# ИСПОЛЬЗОВАНИЕ КЛАССОВ И БИБЛИОТЕК

*Во второй части книги рассмотрены вопросы использования классов Java при работе со строками и файлами, для хранения объектов, при создании пользовательских интерфейсов, многопоточное и сетевое программирование с использованием классов из пакетов* ***java.util***, ***java.text***, ***java.net***, ***java.io***, ***java.awt***, ***javax.swing*** *и др.*

*Из-за ограниченности объема книги детальное рассмотрение библиотек классов невозможно. Подробное описание классов и методов можно найти в документации по языку Java, которой необходимо пользоваться каждому Java-программисту.*

## Глава 7
## ОБРАБОТКА СТРОК

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. Кроме того, 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 (заключённая в двойные апострофы), является объектом 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. При этом ссылку на объект класса **String** можно изменить так, чтобы она указывала на другой объект и тем самым на другое значение.

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.getChars(), "UTF-8")
```

где **str** – строка в формате Unicode, можно установить необходимый алфавит с помощью региональной кодировки в качестве второго параметра конструктора, в данном случае кириллицу. Когда Java встречает литерал, заключенный в двойные кавычки, автоматически создается объект 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 = "sun.com";
String s2 = new String("sun.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 регистра. Метод осуществляет вычитание кодов символов вызывающей и передаваемой 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**;

**byte[] getBytes()**, **getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** – извлечение символов строки в массив байт или 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)**, **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**.

В следующем примере массив символов и целое число преобразуются в объекты типа **String** с использованием методов этого класса.

```
/* пример # 1 : использование методов: DemoString.java */
package chapt07;

public class DemoString {
   static int i;

   public static void main(String[] args) {
      char s[] = { 'J', 'a', 'v', 'a' }; // массив
      // комментарий содержит результат выполнения кода
      String str = new String(s); // str="Java"
      if (!str.isEmpty()) {
         i = str.length(); // i=4
         str = str.toUpperCase(); // str="JAVA"
         String num = String.valueOf(6); // num="6"
         num = str.concat("-" + num); // num="JAVA-6"
         char ch = str.charAt(2); // ch='V'
         i = str.lastIndexOf('A'); // i=3 (-1 если нет)
         num = num.replace("6", "SE"); // num="JAVA-SE"
         str.substring(0, 4).toLowerCase(); // java
         str = num + "-6";// str="JAVA-SE-6"
         String[] arr = str.split("-");
         for (String ss : arr)
            System.out.println(ss);
      } else { System.out.println("String is empty!");

      }
   }
}
```

В результате будет выведен массив строк:

```
JAVA
SE
6
```

При использовании методов класса **String**, изменяющих строку, создается новый измененный объект класса **String**. Сохранить изменения в объекте класса **String** можно только с применением оператора присваивания, т.е.

установкой ссылки на этот новый объект. В следующем примере будет выведено последнее после присваивания значение **str**.

```
/* пример # 2 : передача строки по ссылке: RefString.java */
package chapt07;

public class RefString {
     public static void changeStr(String s) {
          s.concat(" Microsystems");// создается новая строка
     }
     public static void main(String[] args) {
          String str = new String("Sun");
          changeStr(str);
          System.out.println(str);
     }
}
```

В результате будет выведена строка:

```
Sun
```

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

В следующем примере рассмотрены особенности хранения и идентификации объектов на примере вызова метода **equals()**, сравнивающего строку **String** с указанным объектом и метода **hashCode()**, который вычисляет хэш-код объекта.

```
/* пример # 3 : сравнение ссылок и объектов: EqualStrings.java */
package chapt07;

public class EqualStrings {
     public static void main(String[] args) {
          String s1 = "Java";
          String s2 = "Java";
          String s3 = new String("Java");
          System.out.println(s1 + "==" + s2 +
                                  " : " + (s1 == s2)); // true
          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
          System.out.println(s1.hashCode());
          System.out.println(s2.hashCode());
          System.out.println(s3.hashCode());
     }
}
```

В результате, например, будет выведено:

```
Java==Java : true
Java==Java : false
Java equals Java : true
Java equals Java : true
2301506
2301506
2301506
```

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

Т.к. 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, то есть 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()**.

```
// пример # 4 : применение intern() : DemoIntern.java
package chapt07;

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 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.

Ниже рассмотрена сортировка массива строк методом выбора.

```
// пример # 5 : сортировка: SortArray.java
package chapt07;

public class SortArray {
     public static void main(String[] args) {
String a[] = {"  Alena", "Alice  ", " alina",
"  albina", "   Anastasya", "   ALLA ", "AnnA  "};
          for(int j = 0; j < a.length; j++)
               a[j] = a[j].trim().toLowerCase();
          for(int j = 0; j < a.length - 1; j++)
            for(int i = j + 1; i < a.length; i++)
```

```
            if(a[i].compareTo(a[j]) < 0)    {
                String temp = a[j];
                a[j] = a[i];
                a[i] = temp;
            }
        int i = -1;
        while(++i < a.length)
            System.out.print(a[i] + " ");
    }
}
```

Вызов метода **trim()** обеспечивает удаление всех начальных и конечных символов пробелов. Метод **compareTo()** выполняет лексикографическое сравнение строк между собой по правилам Unicode.

## 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 и единственным отличием **StringBuilder** от **StringBuffer** является потокобезопасность последнего. В версии 1.5.0 был добавлен непотокобезопасный (следовательно, более быстрый в обработке) класс **StringBuilder**, который 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.

```
/* пример # 6 : свойства объекта StringBuffer: DemoStringBuffer.java */
package chapt07;

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

Результатом выполнения данного кода будет:

```
длина ->0
размер ->16
строка ->Java
длина ->4
размер ->16
реверс ->avaJ
```

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 ёмкость объекта автоматически увеличивается, оставляя при этом резерв для дальнейших изменений. С помощью метода **reverse()** можно быстро изменить порядок символов в объекте.

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

```
/* пример # 7 : изменение объекта StringBuffer: RefStringBuffer.java */
package chapt07;

public class RefStringBuffer {
     public static void changeStr(StringBuffer s) {
          s.append(" Microsystems");
     }
     public static void main(String[] args) {
          StringBuffer str = new StringBuffer("Sun");
          changeStr(str);
          System.out.println(str);
     }
}
```

В результате выполнения этого кода будет выведена строка:

```
Sun Microsystems
```

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

Для класса **StringBuffer** не переопределены методы **equals()** и **hashCode()**, т.е. сравнить содержимое двух объектов невозможно, к тому же хэш-коды всех объектов этого типа вычисляются так же, как и для класса **Object**.

```
/*пример # 8 : сравнение объектов StringBuffer и их хэш-кодов:
 EqualsStringBuffer.java */
package chapt07;

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

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

## 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 С.

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(OutStream outStrm)
```

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.

В классе объявлены следующие методы:

**Formatter format(String fmtString, Object...args)** – форматирует аргументы, переданные в аргументе переменной длины **args** (количество аргументов в списке вывода не фиксировано), в соответствии со спецификаторами формата, содержащимися в **fmtString**. Возвращает вызывающий объект;

**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;

**IOException ioException()** – если объект, приемник отформатированного вывода, генерирует исключение типа **IOException**, возвращает это исключение. В противном случае возвращает **null**;

**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 **FormatterClosedEception**;

**String toString()** – возвращает объект типа **String**, содержащий отформатированный вывод.

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 **%** |

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

```
/*пример # 9 : форматирование строки при помощи метода format():
 SimpleFormatString.java */
package chapt07;
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",'6');
            System.out.print(f);
      }
}
```

В результате выполнения этого кода будет выведено:

```
This book is about
JAVA 6
```

```
/*пример # 10 : форматирование чисел с использованием спецификаторов %x,
%o, %a, %g: FormatterDemoNumber.java */
package chapt07;
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**" |

```
/*пример # 11 : форматирование даты и времени:
FormatterDemoTimeAndDate.java */
package chapt07;
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 на использование флагов форматирования.

```
/*пример # 12: применение флагов форматирования: FormatterDemoFlags.java */
package chapt07;
```

```
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
```

У класса **Formatter** есть полезное свойство, которое позволяет задавать аргумент, к которому следует применить конкретный спецификатор формата. По умолчанию соответствие между спецификаторами и аргументами, на которые они воздействуют, устанавливается в соответствии с порядком их следования, слева

направо. Это означает, что первый спецификатор формата соответствует первому аргументу, второй спецификатор – второму аргументу и т. д. Однако, используя порядковый номер или индекс аргумента, можно указать явное соответствие спецификатора формата аргументу.

Порядковый номер аргумента указывается за знаком **%** в спецификаторе формата и имеет следующий формат: **N$**. Символ **N** обозначает порядковый номер нужного аргумента, нумерация аргументов начинается с единицы.

```
/*пример #13: применение порядкового номера аргумента:
FormatterDemoArguments.java */
package chapt07;
import java.util.Formatter;

public class FormatterDemoArguments {
      public static void main(String[] args) {
            Formatter f = new Formatter();
            Number n[] = { 4, 2.2, 3, 1.1 };
            f.format("%4$.1f %2$.1f %3$d %1$d", n[0], n[1],
                  n[2], n[3]);
            System.out.println(f);
      }
}
```

В результате выполнения этого кода будет выведено:

```
1,1 2,2 3 4
```

Такой же вывод легко получить, используя метод **printf()** в виде:

```
System.out.printf("%4$.1f %2$.1f %3$d %1$d", n[0], n[1],
                                          n[2], n[3]);
```

## Лексический анализ текста

Класс **StringTokenizer** содержит методы, позволяющие разбивать текст на лексемы, отделяемые разделителями. Набор разделителей по умолчанию: пробел, символ табуляции, символ новой строки, перевод каретки. В задаваемой строке разделителей можно указывать другие разделители, например «= , ; : ».

Класс **StringTokenizer** имеет конструкторы:

```
StringTokenizer(String str);
StringTokenizer(String str, String delimiters);
StringTokenizer(String str, String delimiters,
                                    Boolean delimAsToken);
```

Некоторые методы:

**String nextToken()** – возвращает лексему как **String** объект;

**boolean hasMoreTokens()** – возвращает **true**, если одна или несколько лексем остались в строке;

**int countToken()** – возвращает число лексем.

Класс был реализован в самой первой версии языка. Однако в настоящее время существуют более совершенные средства по обработке текстовой информации – регулярные выражения.

## 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 основные логические конструкции для задания шаблона.

Если в строке, проверяемой на соответствие, необходимо, чтобы 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]]** | либо между **a** и **d**, либо между **m** и **p** |
| **[e-z&&[dem]]** | **e** либо **m** (конъюнкция) |

Кроме стандартных классов символов, существуют предопределенные классы символов:

| | |
|---|---|
| **.** | любой символ |
| **\d** | [**0-9**] |
| **\D** | [**^0-9**] |
| **\s** | [ **\t\n\x0B\f\r**] |
| **\S** | [**^\s**] |
| **\w** | [**a-zA-Z_0-9**] |
| **\W** | [**^\w**] |
| **\p{javaLowerCase}** | ~ **Character.isLowerCase()** |
| **\p{javaUpperCase}** | ~ **Character.isUpperCase()** |

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**.

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

**boolean matches(String regex, CharSequence input)** – проверяет на соответствие строки **input** шаблону **regex**.

**String pattern()** – возвращает строку, соответствующую шаблону.

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

**String[] split(CharSequence input, int limit)** – разбивает строку **input** на не более чем **limit** частей.

С помощью метода **matches()** класса **Pattern** можно проверять на соответствие шаблону целой строки, 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 строка шаблону;

**boolean lookingAt()** – пытается найти последовательность символов, начинающуюся с начала строки и соответствующую шаблону;

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

Иногда необходимо сбросить состояние **Matcher**'а 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)**.

Для того чтобы ограничить поиск границами входной последовательности, применяется метод **region(int start, int end)**, а для получения значения этих границ – **regionEnd()** и **regionStart()**. С регионами связано несколько методов:

**Matcher useAnchoringBounds(boolean b)** – если установлен в **true**, то начало и конец региона соответствуют символам **^** и **$** соответственно.

**boolean hasAnchoringBounds()** – проверяет закрепленность границ.

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.

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

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

**String group()** – возвращает всю подпоследовательность, удовлетворяющую шаблону;

**String group(int group)** – возвращает конкретную группу;

**int groupCount()** – возвращает количество групп;

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

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

**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.

```
/* пример #14 : обработка строк с помощью шаблонов :  DemoRegular.java */
package chapt07;
import java.util.regex.*;

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+)@(\\w+\\.)(\\w+)(\\.\\w+)*";
            String s =
             "адреса эл.почты:mymail@tut.by и rom@bsu.by";
            Pattern p2 = Pattern.compile(regex);
            Matcher m2 = p2.matcher(s);
            while (m2.find())
              System.out.println("e-mail: " + m2.group());

//разбивка строки на подстроки с применением шаблона в качестве разделителя
            Pattern p3 = Pattern.compile("\\d+\\s?");
            String[] words =
                  p3.split("java5tiger 77 java6mustang");
```

```
            for (String word : words)
                  System.out.println(word);
    }
}
```

В результате будет выведено:

```
true
e-mail: mymail@tut.by
e-mail: rom@bsu.by
java
tiger
java
mustang
```

Следующий пример показывает, как использовать группы, а также собственные и неполные квантификаторы.

```
/* пример # 15 : группы и квантификаторы : Groups.java */
package chapt07;

public class Groups {
    public static void main(String[] args) {
        String input = "abdcxyz";
        myMatches("([a-z]*)([a-z]+)", input);
        myMatches("([a-z]?)([a-z]+)", input);
        myMatches("([a-z]+)([a-z]*)", input);
        myMatches("([a-z]?)([a-z]?)", input);
    }
    public static void myMatches(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));
        } else
            System.out.println("nothing");
        System.out.println();
    }
}
```

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.

В классе **Matcher** объявлены два полезных метода для замены найденных подпоследовательностей во входной строке.

**Matcher appendReplacement(StringBuffer sb, String replacement)** – метод читает символы из входной стоки и добавляет их в **sb**. Чтение останавливается на **start()-1** позиции предыдущего совпадения, после чего происходит добавление в **sb** строки **replacement**. При следующем вызове этого метода производится добавление символов, начиная с символа с индексом **end()** предыдущего совпадения.

**StringBuffer appendTail(StringBuffer sb)** – добавляет оставшуюся часть символов из входной последовательности в **sb**. Как правило, вызывается после одного или нескольких вызовов метода **appendReplacement()**.

## Интернационализация текста

Класс **java.util.Locale** позволяет учесть особенности региональных представлений алфавита, символов 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 других стран объект **Locale** нужно создавать с помощью конструктора:

```
Locale myLocale = new Locale("bel", "BY");
```

Получить доступ к текущему варианту региональных параметров можно следующим образом:

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

Если, например, в ОС установлен регион «Россия» или в приложении с помощью **new Locale("ru", "RU")**, 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:

```
RU
Россия
ru
русский
```

Для создания приложений, поддерживающих несколько языков, существует целый ряд решений. Самое логичное из них – использование взаимодействия классов **java.util.ResourceBundle** и **Locale**. Класс **ResourceBundle** предназначен в первую очередь для работы с текстовыми файлами свойств (расширение **.properties**). Каждый объект **ResourceBundle** представляет собой набор объектов 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_de_CH.properties** соответствует объекту **Locale**, заданному кодом языка немецкого (**de**) и кодом страны Швейцарии (**CH**).

```
text.properties
text_ru.properties
text_de_CH.properties
text_en_CA_UNIX.properties
```

Чтобы выбрать определенный объект **ResourceBundle**, следует вызвать метод **ResourceBundle.getBundle(**параметры**)**. Следующий фрагмент выбирает **text** объекта **ResourceBundle** для объекта **Locale**, который соответствует английскому языку, стране Канаде и платформе UNIX.

```
Locale currentLocale = new Locale("en", "CA", "UNIX");
ResourceBundle rb =
   ResourceBundle.getBundle("text", currentLocale);
```

Если объект **ResourceBundle** для заданного объекта **Locale** не существует, то метод **getBundle()** извлечет наиболее общий. В случае если общее определение файла ресурсов не задано, то метод **getBundle()** генерирует исключительную ситуацию **MissingResourceException**. Чтобы этого не произошло, необходимо обеспечить наличие базового файла ресурсов без суффиксов, а именно: **text.properties** в отличие от частных случаев вида:

```
text_en_US.properties
text_bel_BY.properties
```

В файлах свойств информация должна быть организована по принципу:

```
key1 = value1
key2 = value2
...
```

Например: **str1 = To be or not to be?**

Перечисление всех ключей в виде **Enumeration<String>** можно получить вызовом метода **getKeys()**. Конкретное значение по ключу извлекается методом **String getString(String key)**.

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

*// пример # 16 : поддержка различных языков: HamletInternational.java*

```
package chapt8;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.util.Locale;
```

```
import java.util.ResourceBundle;

public class HamletInternational {
      public static void main(String[] args) {
            String country = "", language = "";
            System.out.println("1 - Английский");
            System.out.println("2 - Белорусский");
            System.out.println("Любой символ - Русский");
            char i = 0;
            try {
                  i = (char) System.in.read();
            } catch (IOException e1) {
                  e1.printStackTrace();
            }
            switch (i) {
             case '1':
                  country = "US";
                  language = "EN";
                  break;
             case '2':
                  country = "BY";
                  language = "BEL";
            }
            Locale current = new Locale(language, country);
            ResourceBundle rb =
            ResourceBundle.getBundle("text", current);
            try {
                  String st = rb.getString("str1");
                  String s1 =
            new String(st.getBytes("ISO-8859-1"), "UTF-8");
                  System.out.println(s1);

                  st = rb.getString("str2");
                  String s2 =
            new String(st.getBytes("ISO-8859-1"), "UTF-8");
                  System.out.println(s2);
            } catch (UnsupportedEncodingException e) {
                  e.printStackTrace();
            }
      }
}
```

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

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

Файл **text_bel_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.
```

### 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.

```
// пример # 17 : региональные представления чисел: DemoNumberFormat.java
package chapt07;
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 = "1.234,567";//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) {
                  e.printStackTrace();
            }
```

```
        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("\n" + sUs + "\n" + sFr);
        }
}
```

Результат работы программы:

```
iGe = 1234,567000
iUs = 1,234000
iFr = 1,000000
1,234.567
1 234,567
```

Аналогично выполняются переходы от одного регионального стандарта к другому при отображении денежных сумм.

### 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 dat = df.format(new Date());
```

С помощью метода **Date parse(String source)** можно преобразовать переданную в виде строки дату в объектное представление конкретного регионального формата, например:

```
String str = "April 3, 2006";
Date d = df.parse(str);
```

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

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

```
// пример # 18 : региональные представления дат: DemoDateFormat.java
package chapt07;
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 3, 2006";
            try {
                  d = df.parse(str);
                  System.out.println(d);
            } catch (ParseException e) {
                  e.printStackTrace();
            }
            df =
DateFormat.getDateInstance(DateFormat.FULL,
                              new Locale("ru","RU"));
            System.out.println(df.format(d));

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

            d = new Date();
      //загрузка в объект df текущего времени
            df = DateFormat.getTimeInstance();
      //представление и вывод времени в текущем формате дат
            System.out.println(df.format(d));
      }
}
```

Результат работы программы:

```
Mon Apr 03 00:00:00 EEST 2006
3 Апрель 2006 г.
Montag, 3. April 2006
05:45:16
```

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 будут выведены две сотни строк, каждая из которых представляет текущую дату в соответствии с региональным стандартом, выводимым перед датой с помощью инструкции **`loc.toString()`**.

### *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 русском 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 `P`, если она не последняя в слове, ошибочно напечатана буква `A` вместо `O`. Внести исправления в текст.
4. В тексте слова заданной длины заменить указанной подстрокой, длина которой может не совпадать с длиной слова.
5. В тексте после `k`-го символа вставить заданную подстроку.
6. После каждого слова текста, заканчивающегося заданной подстрокой, вставить указанное слово.
7. 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.
8. Из небольшого текста удалить все символы, кроме пробелов, не являющиеся буквами. Между последовательностями подряд идущих букв оставить хотя бы один пробел.
9. Из текста удалить все слова заданной длины, начинающиеся на согласную букву.
10. Удалить из текста его часть, заключенную между двумя символами, которые вводятся (например, между скобками ‘(’ и ‘)’ или между звездочками ‘*’ и т.п.).
11. В тексте найти все пары слов, из которых одно является обращением другого.
12. Найти и напечатать, сколько раз повторяется в тексте каждое слово, которое встречается в нем.
13. В тексте найти и напечатать `n` символов (и их количество), встречающихся наиболее часто.
14. Найти, каких букв, гласных или согласных, больше в каждом предложении текста.

15. В стихотворении найти количество слов, начинающихся и заканчивающихся гласной буквой.
16. Напечатать без повторения слова текста, у которых первая и последняя буквы совпадают.
17. В тексте найти и напечатать все слова максимальной и все слова минимальной длины.
18. Напечатать квитанцию об оплате телеграммы, если стоимость одного слова задана.
19. В стихотворении найти одинаковые буквы, которые встречаются во всех словах.
20. В тексте найти первую подстроку максимальной длины, не содержащую букв.
21. В тексте определить все согласные буквы, встречающиеся не более чем в двух словах.
22. Преобразовать текст так, чтобы каждое слово начиналось с заглавной буквы.
23. Подсчитать количество содержащихся в данном тексте знаков препинания.
24. В заданном тексте найти сумму всех встречающихся цифр.
25. Из кода Java удалить все комментарии (//, /*, /**).
26. Дан текст на английском языке. Пусть все слова встречаются четное количество раз, за исключением одного. Определить это слово. При сравнении слов регистр не учитывать.
27. Определить сумму всех целых чисел, встречающихся в заданном тексте.
28. Из английского текста удалить все пробелы, если он разделяет два различных знака препинания и если рядом с ним находится еще один пробел.
29. Строка состоит из упорядоченных чисел от 0 до 100000, записанных подряд без пробелов. Определить, что будет подстрокой от позиции n до m.
30. Определить количество вхождений заданного слова в текст, игнорируя регистр символов и считая буквы «е», «ё», и «и», «й» одинаковыми.
31. Преобразовать текст так, чтобы только первые буквы каждого предложения были заглавными.
32. Заменить в тексте все шаблоны типа %user%Бендер%/user% на <a href=”http://www.my.by/search.htm?param=Бендер”>Бендер</a>.
33. В Java код добавить корректные getter и setter-методы для всех полей данного класса, при их отсутствии.
34. Вывести все предложения заданного текста в порядке возрастания количества слов в каждом из них.

***Вариант В***

1. В тексте нет слов, начинающихся одинаковыми буквами. Напечатать слова текста в таком порядке, чтобы последняя буква каждого слова совпадала с первой буквой последующего слова. Если все слова нельзя напечатать в таком порядке, найти такую цепочку, состоящую из наибольшего количества слов.

2. Найти наибольшее количество предложений текста, в которых есть одинаковые слова.
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. 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.
7. 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.
8. На основании правила кодирования, описанного в предыдущем примере, расшифровать заданный набор символов.
9. Напечатать слова русского текста в алфавитном порядке по первой букве. Слова, начинающиеся с новой буквы, печатать с красной строки.
10. Рассортировать слова русского текста по возрастанию доли гласных букв (отношение количества гласных к общему количеству букв в слове).
11. Слова английского текста, начинающиеся с гласных букв, рассортировать в алфавитном порядке по первой согласной букве слова.
12. Все слова английского текста рассортировать по возрастанию количества заданной буквы в слове. Слова с одинаковым количеством расположить в алфавитном порядке.
13. Ввести текст и список слов. Для каждого слова из заданного списка найти, сколько раз оно встречается в тексте, и рассортировать слова по убыванию количества вхождений.
14. Все слова текста рассортировать в порядке убывания их длин, при этом все слова одинаковой длины рассортировать в порядке возрастания в них количества гласных букв.
15. В тексте исключить подстроку максимальной длины, начинающуюся и заканчивающуюся заданными символами.
16. Заменить все одинаковые рядом стоящие символы в тексте одним символом.
17. Вывести в заданном тексте все слова, расположив их в алфавитном порядке.
18. Подсчитать, сколько слов в заданном тексте начинается с прописной буквы.
19. Подсчитать, сколько раз заданное слово входит в текст.
20. Преобразовать каждое слово в тексте, удалив из него все последующие (предыдущие) вхождения первой (последней) буквы этого слова.
21. Вычеркнуть из текста минимальное количество предложений, так чтобы у любых двух оставшихся предложений было хотя бы одно общее слово.

22. 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 $90^0$ 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.
23. На основании правила кодирования, описанного в предыдущем примере, расшифровать заданный набор символов.
24. Исключить из текста подстроку максимальной длины, начинающуюся и заканчивающуюся одним и тем же символом.
25. 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.
26. Распаковать текст, сжатый по правилу из предыдущего задания.
27. Определить, удовлетворяет ли имя файла маске. Маска может содержать символы ‘?’ (произвольный символ) и ‘*’ (произвольное количество произвольных символов).
28. Отсортировать слова в тексте по убыванию количества вхождений заданного символа, а в случае равенства – по алфавиту. Словом считать максимальную группу подряд стоящих не пробельных символов.
29. 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.
30. В заданном тексте найти подстроку максимальной длины, являющуюся палиндромом, т.е. читающуюся слева направо и справа налево одинаково.
31. 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.
32. Изменить программу из предыдущего примера так, чтобы она осуществляла форматирование с выравниванием по обоим краям. Для этого добавить дополнительные пробелы между словами.
33. 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.
34. Пусть массив содержит миллион символов и необходимо сформировать из них строку путем конкатенации. Определить время работы кода. Как можно ускорить процесс, используя класс StringBuffer?

35. 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.
36. Восстановить слово по его преобразованию Барроуза – Уиллера. К примеру, получив на вход VJAA, в результате работы программа должна выдать слово JAVA.

### *Тестовые задания к главе 7*

***Вопрос 7.1.***

Дан код:

```
public class Quest1 {
   public static void main(String[] args) {
      String str = new String("java");
      int i=1;
      char j=3;
      System.out.println(str.substring(i,j));
}}
```

В результате при компиляции и запуске будет выведено:

1) ja;
2) av;
3) ava;
4) jav;
5) ошибка компиляции: заданы некорректные параметры для метода `substring()`.

***Вопрос 7.2.***

Какую инструкцию следует использовать, чтобы обнаружить позицию буквы **v** в строке **`str = "Java"?`**

1) `charAt(2,str);`
2) `str.charAt(2);`
3) `str.indexOf('v');`
4) `indexOf(str,'v');`

***Вопрос 7.3.***

Какие из следующих операций корректны при объявлении?

```
String s = new String("Java");
String t = new String();
String r = null;
```

1) `r = s + t + r;`
2) `r = s + t + 'r';`

3) `r = s & t & r;`
4) `r = s && t && r;`

***Вопрос 7.4.***

Дан код:

```
public class Quest4 {
   public static void main(String[] args) {
      String str="ava";
      char ch=0x74; // 74 - это код символа 'J'
      str=ch+str;
      System.out.print(str);
}}
```

В результате при компиляции и запуске будет выведено:

1) 74ava;
2) Java;
3) 0H74ava;
4) ошибка компиляции: недопустимая операция `ch+str`;
5) ошибка компиляции: недопустимое объявление `char ch=0x74`;
6) нет правильного ответа.

***Вопрос 7.5.***

Что будет результатом компиляции и выполнения следующего кода?

```
public class Quest5 {
   public static void main(String[] args) {
     StringBuffer s = new StringBuffer("you java");
      s.insert(2, "like ");
      System.out.print(s);
}}
```

1) `yolike u java;`
2) `you like java;`
3) `ylike ou java;`
4) `you java like;`
5) ошибка компиляции;
6) нет правильного ответа.