Лабораторная работа 3. Работа со строками в Java

1. Основные теоретические сведения

1.1. Классы String и StringBuffer

Для хранения и обработки строк в Java имеются два класса: **String** для неизменяемых строк и **StringBuffer** – для строк, которые могут меняться. Оба класса расширяют класс Object. Они находятся в пакете **java.lang**, поэтому их не надо подключать с помощью оператора import.

Строковые литералы в Java, как и в C, заключаются в двойные апострофы, например, "*abc*" задает строковый литерал abc.

Если внутри строкового литерала необходимо задать символ апострофа, он задается с помощью символов \', например, "it\'s" задает строковый литерал it's.

1.1.1. Создание и инициализация объекта класса String

Инициализация объекта класса **String** может выполняться как с помощью оператора присваивания переменной класса **String** строковой переменной или строкового литерала, например: **String** str = "Строка 1"; либо при создании объекта с помощью оператора **new** с использованием одного из конструкторов класса **String**, представленных в табл. 1.1.

Таблица 1.1. Конструкторы класса `String`

Конструктор	Объект, который он создает
`String()`	Пустая строка.
`String (String` *строка-1*`)`	Новая строка, которая является копией *строки-1*.
`String(char[]` *массив*`)`	Строка, созданная из элементов *массива*.
`String(char[]` *массив*`, int` *начальный-индекс*`, int` *длина*`)`	Строка, создаваемая из фрагмент *массива* символов, начинающегося с позиции *начальный-индекс* и заданной *длины*.

`String(byte[]` *массив*`)`	Строка, создаваемая из *массива* байт, с использованием кодировки на данном компьютере по умолчанию (для русского языка в Windows – кодировка Windows-1251).
`String(byte[]` *массив*`, int` *начальный-индекс*`, int` *длина*`)`	Строка, создаваемая из фрагмент *массива* байт, начинающегося с позиции *начальный-индекс* и заданной *длины*.

Длина строки может быть определена с помощью метода **public int length()**.

Единственная операция, которую можно использовать для строк, является операция сцепления (конкатенация) двух или более строк – "+".

Строки класса **String** можно изменять, но при каждом изменении длины строки создается новый экземпляр строки.

1.1.2. Создание и инициализация объекта класса StringBuffer

Класс **StringBuffer** похож на класс **String**, но строки, созданные с помощью этого класса можно модифицировать, т.е. их содержимое и длина могут меняться. При изменении строки класса **StringBuffer** программа не создает новый строковый объект, а работает непосредственно с исходной строкой, т.е. все методы оперируют непосредственно с буфером, содержащим строку. Поэтому класс **StringBuffer** обычно используется, когда строку приходится часто модифицировать с изменением ее длины.

Размещение строк в объекте **StringBuffer** выполняется следующим образом. Для объекта **StringBuffer** задается размер или емкость (capacity) буферной памяти для строки. Строка символов в объекте **StringBuffer**, характеризуется своей длиной, которая может быть меньше или равна емкости буфера. Если длина строки меньше емкости буфера, то оставшаяся длина строки заполняется символом Unicode "\u0000". Если в результате модификации строки ее длина станет больше емкости буфера, емкость буфера автоматически увеличивается.

В классе **StringBuffer** имеется три конструктора, позволяющих по-разному создавать объекты типа **StringBuffer**:

StringBuffer()

StringBuffer(**int** длина)

StringBuffer(**String** строка)

Первый конструктор создает пустой объект **StringBuffer** с емкостью буферной памяти в 16 символов, второй конструктор задает буфера с емкостью заданной длины для хранения строки, а третий конструктор создает объект **StringBuffer** из строки с емкостью буфера, равной длине строки.

Методы определения и установки характеристик строки в классе **StringBuffer** приведены в табл. 1.2.

Таблица 1.2. Методы определения и установки характеристик строки в классе **`StringBuffer`**

Объявление метода	Действие
`public int length()`	Возвращает длину строки для объекта класса **`StringBuffer`**.
`public int capacity()`	Возвращает текущую емкость буферной области для объекта класса **`StringBuffer`**.
`public void ensureCapacity(int` *емкость*`)`	Устанавливает емкость буферной области для объекта класса **`StringBuffer`**.
`public void setLength(int` *новая-длина*`)`	Устанавливает *новую-длину* строки. Если новая длина больше старой, увеличиваются длины строки и буфера, при этом дополнительные символы заполняются нулями. Если новая длина меньше старой, символы в конце строки отбрасываются, а размер буфера не изменяется.
`public String toString()`	Преобразует строку **`StringBuffer`** в строку **`String`**.

1.1.3. Сравнение строк

Поскольку в Java строки являются объектами, для сравнения строк можно использовать оператор "==" и метод **public boolean equals**(**Object** объект), сравнивающий строку, для которой вызывается метод, с объектом. Результат будет **true**, только если объект является строкой и значения сравниваемых строк равны.

Использование оператора "==" для сравнения строк может привести к неверному результату, если сравниваемые строки – разные объекты, поэтому более предпочтительным является использование метода **equals**() (этот метод работает и для строк **String** и для строк **StringBuffer**).

Другие методы сравнения строк, также возвращающие булевские значения, приведены в табл. 1.3.

Таблица 1.3. Методы сравнения строк класса **`String`**

Метод	Возвращает **`true`**, когда
`equalsIgnoreCase(String` *строка-1***`)`**	Строка равняется *строке-1* независимо от регистра символов.
`startsWith(String` *префикс***`)`**	Строка начинается с *префикс*.
`startsWith(String` *префикс***`,`** **`int`** *начальный-индекс***`)`**	Подстрока строки *префикс*, начиная с позиции *начальный-индекс*, содержится в начале проверяемой строки.
`endsWith(String` *суффикс***`)`**	Строка заканчивается строкой *суффикс*.
`regionMatches(int` *начальный-индекс***`,`** **`String`** *строка-1***`,`** **`int`** *начальный-индекс-1***`,`** **`int`** *длина***`)`**	Подстрока в строке, начиная с позиции со смещением *начальный-индекс*, соответствует подстроке *строки-1*, начиная по смещению *начальный-индекс-1*, и заданной *длины*.
`regionMatches(boolean` *без-учета-регистра***`,`** **`int`** *начальный-индекс***`,`** **`String`** *строка-1***`,`** **`int`** *начальный-индекс-1***`,`** **`int`** *длина***`)`**	То же, что и предыдущий метод, но игнорирует регистр символов, когда параметр *без-учета-регистра* равен **`true`**.

Метод **int compareTo(String** anotherString) – лексикографически сравнивает две строки и возвращает 0, если строки равны по длине и имеют одинаковое, значение, меньшее 0, если в первой позиции, в которой символы строк не равны, код символа в первой строке меньше кода символа во второй строке или длина первой строки меньше длины второй строки все символы первой строки равны символам в тех же позициях второй строки. Если в первой позиции, в которой символы строк не равны код символа в первой строке больше кода символа во второй строке или длина первой строки больше длины второй строки, возвращается значение, больше 0.

1.1.4. Поиск в строках

Для поиска символов или последовательностей символов (только в строках класса **String**) используются следующие перегружаемые методы **indexOf**(), приведенные в табл. 1.4.

Таблица 1.4. Методы поиска класса `String`

Объявление метода	Возвращаемое значение
`int indexOf(int` *символ*`)`	Первая позиция в строке, в которой встречается *символ*.
`public int indexOf(int` *символ*`, int` *начальный-индекс*`)`	Первая позиция в строке, начиная с позиции *начальный-индекс*, в которой встречается *символ*.
`int indexOf(String` *строка*`)`	Первая позиция в строке, в которой встречается *строка*.
`public int indexOf(String` *строка*`, int` *начальный-индекс*`)`	Первая позиция в строке, начиная с позиции *начальный-индекс*, в которой встречается *строка*.

Для каждого метода **indexOf**() имеется соответствующий метод **lastIndexOf**(), который начинает поиск символа или строки не с начала, а с конца строки. Если символ или строка не найдены в строке, в которой производится поиск, методы **indexOf**() и **lastIndexOf**() возвращают значение -1.

1.1.5. Извлечение символов и подстрок из строки

Извлечение символов и подстрок из строки, а также создание новых строк на основе существующих строк выполняется с помощью методов, приведенных в табл. 1.5.

Таблица 1.5. Методы манипуляций со строками класса `String`

Метод	Возвращаемое значение
`char charAt(int` *индекс*`)`	Символ строки в позиции *индекс*.
`char[] toCharArray()`	Массив символов – копию строки.
`String substring(int` *начальный-индекс*`)`	Подстрока исходной строки, начинающаяся с позиции *начальный-индекс* исходной строки.
`String substring(int` *начальный-индекс*`, int` *конечный-индекс*`)`	Подстрока исходной строки, начинающаяся в позиции *начальный-индекс* и заканчивающаяся в позиции *конечный-индекс*`-1` исходной строки.
`void getChars (int` *начальный-индекс*`, int` *конечный-индекс*`,`	Возвращаемого значения нет. Копирует часть строки, начиная с символа в позиции *начальный-индекс* и

char[] *массив*, **int** *индекс-вставки*)	заканчивая символом в позиции *индекс-вставки* **+ (***конечный-индекс* **-** *начальный-индекс***)-1** в символьный *массив*, начиная с позиции *индекс-вставки*.

1.1.6. Модификация строк

Создание новых строк на основе существующих строк выполняется с помощью методов класса String, приведенных в табл. 1.6.

Таблица 1.6. Методы создания новых строк класса **String**

Метод	Возвращаемое значение
String concat(String *строка***)**	Исходная строка, в конец которой добавлена *строка*.
String toLowerCase()	Исходная строка, переведенная в нижний регистр.
String toUpperCase()	Исходная строка, переведенная в верхний регистр.
String trim()	Исходная строка, из которой исключены начальные и конечные пробельные символов.
String replace(char *символ-1*, **char** *символ-2***)**	Исходная строка, в которой все символы *символ-1* заменены на *символ-2*.
public static String valueOf(*тип имя***)**	Строка, преобразованная из примитивных типов данных. Допустимые *типы* параметра: **boolean**, **char**, **int**, **long**, **float**, **double** или **char[]**.
public static String valueOf(char[] *массив*, **int** *начальный-индекс*, **int** *длина***)**	Строка, полученная из фрагмента *массива*, начинающегося в позиции *начальный-индекс* и заданной *длины*.

Методы класса **StringBuffer**, представленные в табл. 1.7, позволяют непосредственно модифицировать строку.

Таблица 1.7. Методы модификации строк класса **StringBuffer**

Метод	Действие
public void setCharAt(int *индекс*, **char** *символ***)**	Помещает в позиции *индекс* заданный *символ*.
public void deleteCharAt(int *индекс***)**	Удаляет символ в позиции *индекс*.
public StringBuffer replace(int *начальный-индекс*, **int** *конечный-индекс*, **String** *строка***)**	Заменяет фрагмент строки, начиная с позиции *начальный-индекс* и до позиции *конечный-индекс***-1** *строкой*, а затем возвращает измененную строку.
public StringBuffer append(*тип имя***)**	Добавляет в конец строки данное заданного *типа* с заданным *именем* и возвращает измененную строку (см.

	ниже). Допустимые *типы* параметра: **Object**, **boolean**, **char**, **int**, **long**, **float**, **double**, **String**, **StringBuffer** или **char[]**.
public StringBuffer insert(int *начальный-индекс*, *тип имя***)**	Вставляет в строку в позиции *начальный-индекс* данное заданного *типа* с заданным *именем* и возвращает измененную строку. Допустимые *типы* параметра: **Object**, **boolean**, **char**, **int**, **long**, **float**, **double**, **String**, **StringBuffer** или **char[]**.
public StringBuffer append(char[] *массив*, **int** *начальный-индекс*, **int** *длина***)**	Добавляет в строку фрагмент символьного *массива*, начинающийся с позиции *начальный-индекс* заданной *длины*, и возвращает измененную строку.
public StringBuffer insert(int *начальный-индекс-строки*, **char[]** *массив*, **int** *начальный-индекс-массива*, **int** *длина***)**	Вставляет в строку в позиции *начальный-индекс-строки* фрагмент символьного *массива*, начинающийся с позиции *начальный-индекс-массива* заданной *длины*, и возвращает измененную строку.

1.2. Регулярные выражения в Java

1.2.1. Основные сведения о регулярных выражениях

Функции работы с подстроками позволяют выполнить операции поиска подстрок в строке и замены подстрок в строке. Однако при работе с данными часто приходится выполнять операции поиска и замены по довольно сложным алгоритмам, например, найти первое вхождение цифры в строке. Хотя такие операции можно выполнить, используя функции работы со строками, условные операторы и операторы цикла, в языке Java, как и в других языках программирования, существует более удобный способ – использование регулярных выражений.

Регулярные выражения используются для решения следующих задач:

- проверки данных на наличие некоторой последовательности данных, заданных с помощью определенного образца, называемого шаблоном (pattern);
- замены или удаления данных;
- извлечения некоторой последовательности из данных.

Синтаксис регулярных выражений в Java в основном такой же, как и в других языках программирования.

1.2.2. Синтаксис регулярного выражения

Регулярное выражение в языке Java является объектом класса **String**, т.е. является строкой – последовательностью символов.

Символы в строке могут быть следующих типов:

- алфавитно-цифровые символы, включая буквы кириллицы;
- символ '\\' – обратная косая черта (обратный слеш);
- символ '\0num' – восьмеричное число, где num – одна, две или три восьмеричные цифры;
- символ '\xhh' – код символа ASCII, где hh – две шестнадцатеричные цифры;
- символ '\uhhhh' – код символа Unicode, где hhhh – четыре шестнадцатеричные цифры;
- символ табуляции ('\t' или '\u0009');
- символ новой строки ('\n' или '\u000A');
- символ возврата каретки ('\r' или '\u000D');
- символ перехода к новой странице ('\f' или '\u000C');
- символ звукового сигнала ('\a' или '\u0007');
- символ Escape (Esc) ('\u001B');
- символ '\cx' – соответствует управляющему символу x (например, \cM соответствует символу Ctrl+M или символу возврата каретки).

Некоторые символы в регулярных выражениях, называемые метасимволами, имеют особое значения. Метасимволами являются следующие символы:

^ $ ( ) \ | [{ ? . + *

Именно использование метасимволов обеспечивает всю мощь и гибкость регулярных выражений.

Если в регулярном выражении необходимо рассматривать метасимвол как обычный символ, перед ним надо поставить символ '\\', например, "\\(".

1.2.3. Операция альтернации

В регулярных выражениях можно объединять несколько шаблонов, так чтобы найденная строка соответствовала хотя бы одному из них. Для решения подобной проблемы служит операция альтернации, которая в

регулярных выражениях задается символом "|", например шаблон "Internet|Интернет" означает поиск в исходной строке либо строки "Internet", либо строки "Интернет".

1.2.4. Одиночный метасимвол

Метасимвол точка "." внутри регулярного выражения точка соответствует любому одиночному символу, кроме символа перевода строки.

1.2.5. Квантификаторы

Квантификаторы – это метасимволы, используемые для указания количественных отношений между символами в шаблоне и в искомой строке. Квантификатор может быть поставлен после одиночного символа или после группы символов.

Простейшим квантификатором является метасимвол "+". Он означает, что идущий перед ним символ соответствует нескольким идущим подряд таким символам в строке поиска. Количество символов может быть любым (максимально большим в рамках соответствия шаблону), но должен присутствовать хотя бы один символ.

Действие метасимвола "*" похоже на действие метасимвола "+". Метасимвол "*" указывает, что идущий перед ним символ встречается нуль или более раз.

Метасимвол "?" указывает, что предшествующий ему символ должен встречаться либо один раз, либо не встречаться вообще.

Если необходимо указать точно количество повторений символа, можно воспользоваться конструкцией {n,m}. Здесь n – минимально допустимое количество повторений предшествующего символа, m – максимально допустимое количество повторений. Один из параметров n или m можно опустить.

Фактически квантификаторы "+", "*" и "?" являются частными случаями конструкции {n,m}: соответственно, {1,}, {0,} и {0,1}.

В регулярных выражениях часто используют сочетание метасимволов ".*". Ему соответствуют любые символы. По правилам обработки регулярных выражений находится самая длинная строка, все еще удовлетворяющая шаблону поиска.

Если необходимо ограничить поиск, следует после квантификатора (в том числе и символа "?") указать символ "?".

1.2.6. Классы символов

Для поиска в регулярных выражениях можно задавать также **классы символов**, заключенные в квадратные скобки. Во время поиска все символы в классе рассматриваются как один символ. Внутри класса можно задавать диапазон символов (когда такой диапазон имеет смысл), помещая дефис между границами диапазона. Внутри символьных классов большинство метасимволов теряют свои значения и становятся обыкновенными символами.

Если первым символом класса является знак вставки "`^`", то значение выражения инвертируется. Другими словами, такому классу соответствует любой символ, **не входящий в класс**.

Так как в классах символы "`]`", "`^`" и "`-`" имеют специальное значение, для их использования в классе существуют определенные правила:

- литерал "`^`" не должен быть первым символом класса;
- перед литералом "`]`" должен стоять символ обратной косой черты;
- для помещения в класс символа "`-`" достаточно либо поставить его на первую позицию, либо поместить перед ним символ обратной косой черты.

3.2.1.6. Специальные символы

Наиболее распространенные классы символов можно задать с помощью следующих специальных символов:

- \d – соответствует любому цифровому символу (эквивалентно [0-9]);
- \D – соответствует любому нецифровому символу (эквивалентно [^0-9]);
- \w – соответствует любой латинской букве или цифре (эквивалентно [A-Za-z0-9]);
- \W – соответствует любому небуквенному (латинскому) и нецифровому символу (эквивалентно [^A-Za-z0-9]);
- \s – соответствует любому пробельному символу (эквивалентно [\f\n\r\t\v]);
- \S – соответствует любому непробельному символу (эквивалентно [^\f\n\r\t\v]).

Следует отметить, что специальные символы \w и \W нельзя использовать для букв кириллицы, а также букв западноевропейских алфавитов, отличных от латинских букв. В этом случае необходимо напрямую задавать диапазон символов, как это делается для классов символов.

1.2.7. Анкеры

С помощью анкеров можно указать, в каком месте строки должно быть найдено соответствие с шаблоном.

В Java определены следующие анкеры:

- ^ – соответствует позиции в начале строки;
- $ – соответствует позиции в конце строки;
- \b – соответствует границе слова, т.е. границе между словом и пробельным символом;
- \B – соответствует не границе слова.

К сожалению, анкеры \b и \B действуют только для строк, состоящих из латинских букв.

1.2.8. Группировка элементов и обратные ссылки

Операция группировки элементов, т.е. заключение группы элементов в круглые скобки, позволяет рассматривать данную группу элементов как один элемент.

Если в регулярных выражениях используются скобки, части искомой строки, соответствующие фрагментам в скобках, запоминаются в специальных переменных $1 (первый фрагмент в скобках), $2 (второй фрагмент в скобках), $3 (третий фрагмент в скобках) и т.д. Такая операция называется **захватом** (capture) переменной.

Для вложенных скобок переменные $1-$9 формируются в порядке появления левой (открывающей) скобки выражения.

Следует отметить, что переменные модифицируются при каждом успешном поиске, независимо от использования в регулярном выражении скобок. Кроме того, значения этих переменных устанавливаются тогда и только тогда, когда строка полностью соответствует шаблону. В противном случае значения этих переменных равны пустой строке.

Переменные $1-$9 можно записывать и в выражении шаблона в форме \i, где i – номер переменной. Переменную называет обратной ссылкой.

Переменные $1-$9 не всегда необходимо использовать как результат поиска соответствия шаблону. В случае, когда необходимо сгруппировать какие-либо символы шаблона, но не выполнять для них операции по определению соответствующих переменных (не выполнять для них

операцию захвата), используется следующая форма группирования: (?:символы).

Другим видом группировки является группировка с «**заглядыванием вперед**». Группирование в этом случае записывается в следующем виде: шаблон(?=символы). В этом случае при поиске соответствия шаблону учитываются символы, заданные в скобках и идущие после шаблона, но в результат поиска эти символы не входят. Следующий поиск начинается с позиции, с которой начинаются символы в скобках, т.е. результат «заглядывания вперед» не учитывается.

Вторая форма группировки с «заглядыванием вперед» записывается в следующем виде: шаблон(?!символы). В отличие от первой формы символы в скобках не должны содержаться в соответствии шаблону. В результат поиска эти символы не входят и результат «заглядывания вперед» также не учитывается.

Группировка «**заглядывание назад**» записывается в следующем виде: (?<=символы) шаблон. В этом случае при поиске соответствия шаблону учитываются символы, заданные в скобках и идущие перед шаблоном, но в результат поиска эти символы не входят. Следующий поиск начинается с первой позиции после шаблона.

Вторая форма группировки с «заглядыванием назад» записывается в следующем виде: (?<!символы) шаблон. В отличие от первой формы символы в скобках не должны содержаться в соответствии шаблону. В результат поиска эти символы не входят и результат «заглядывания назад» также не учитывается.

Для работы с регулярными выражениями в Java используются классы **Pattern**, **Matcher** и **PatternSyntaxException** пакета **java**.**util**.**regex**.

1.2.8. Класс Pattern

Объект класса **Pattern** является откомпилированным представлением шаблона регулярного выражения и создается не с помощью ключевого слова **new**, а с помощью статических методов **compile**() класса **Pattern**.

Метод **public static Pattern compile**(**String** шаблон) возвращает объект класса **Pattern** для заданного в параметре шаблона.

Метод **public static Pattern compile(String** шаблон, **int** флажки) возвращает объект класса **Pattern** для заданного в параметре шаблона с заданными флажками.

Флажки (табл. 1.2.1) представлены в Java как статические поля типа **public static final int** класса **Pattern**.

Таблица 1.2.1. Флажки класса **Pattern**

Флажок	Числовое значение	Использование
CASE_INSENSITIVE	**2**	Включает поиск соответствия без учета верхнего или нижнего регистра, т.е. строки "**abc**", "**Abc**" и "**ABC**" будут считаться соответствующими регулярному выражению "**abc**" (при отключенном флажке шаблону будет соответствовать только первая строка).
UNICODE_CASE	**64**	Если этот флажок включен вместе с флажком **CASE_INSENSITIVE**, то верхний и нижний регистры букв в коде Unicode не учитываются при поиске соответствия, т.е. строки "**строка**", "**Строка**" и "**СТРОКА**" будут считаться соответствующими регулярному выражению "**строка**" (при отключенном флажке **UNICODE_CASE** и включенном флажке **CASE_INSENSITIVE** шаблону будет соответствовать только строки, содержащие латинские буквы).
UNIX_LINES	**1**	При включении этого флажка только символ "**\n**" учитывается как символ окончания строки, в которой выполняется поиск соответствия
MULTILINE	**8**	Если внутри строки, в которой выполняется поиск соответствия, есть символы "**\n**", то считается что строка состоит из нескольких строк (если флажок выключен, то считается, что поиск соответствия производится в одной строке, независимо от наличия символов "**\n**").
LITERAL	**16**	Все символы шаблона, включая метасимволы, рассматриваются как обычные символы (если флажок выключен, метасимволы в строке обрабатываются при компилировании).
DOTALL	**32**	Если в шаблоне есть метасимвол "**.**", то ему будет соответствовать любой символ, включая символ "**\n**" (если флажок выключен, метасимвол "**.**" будет соответствовать любому символу, исключая символ "**\n**").
COMMENTS	**4**	В строке шаблона, допустимы пробелы и комментарии, начинающиеся с символа "**#**" до

		конца строки (при компиляции шаблона пробелы и комментарии будут проигнорированы).
CANON_EQ	**128**	При поиске соответствия будет учитываться соответствие между кодом символа и сами символом, т.е. при включенном флажке латинская буква "**a**" будет соответствовать коду Unicode этой буквы "**\u00E5**" в шаблоне.

Флажки можно включать непосредственно в шаблоне, используя следующую синтаксическую форму: (?строка-символов), где символы в строке-символов могут иметь одно из следующих значений:

i – для флажка CASE_INSENSITIVE;

d – для флажка UNIX_LINES;

m – для флажка MULTILINE;

s – для флажка DOTALL;

u – для флажка UNICODE_CASE;

x – для флажка COMMENTS.

Методы класса **Pattern** приведены в табл. 1.2.2.

Таблица 1.2.2. Методы класса **Pattern**

Метод	Действие и возвращаемое значение
public static boolean matches(String *шаблон*, **CharSequence** *строка-поиска***)**	Проверяет соответствие *шаблона строке-поиска* и возвращает значение **true**, если строка поиска соответствует шаблону и **false** – в противном случае.
public String pattern()	Возвращает строку шаблона для объекта класса **Pattern**.
public int flags()	Возвращает числовое значения флажка для объекта класса **Pattern**; (если задано несколько флажков, возвращает сумму их числовых значений).
public static String quote(String *строка***)**	Возвращает строковый шаблон для заданной *строки* (см. поле **LITERAL**).
public String[] split(CharSequence *строка-поиска***)**	Создает из *строки-поиска* массив, разделенный на элементы по шаблону, заданному в объекте класса **Pattern**.
public String[] split(CharSequence *строка-поиска*, **int** *предел***)**	Создает из *строки-поиска* массив, разделенный на элементы по шаблону, заданному в объекте класса **Pattern**, и с заданным в параметре *предел* количеством элементов (если значение параметра больше или равно количеству элементов, либо

	меньше 0, выводятся все элементы, если меньше количества элементов – все оставшиеся соответствия выводятся в последнем элементе массива).
`public String toString()`	Возвращает строковое представление откомпилированного шаблона.

1.2.9. Класс Matcher

Класс **Matcher** обеспечивает выполнение поиска или замены соответствия заданному объектом класса **Pattern** шаблону.

Объект класса **Matcher** создается с помощью метода **public Matcher matcher(CharSequence** строка-поиска) класса **Pattern** для строки-поиска.

Строковое представление объекта класса **Matcher** можно получить с помощью метода **public String toString**().

Поиск соответствия выполняется в подстроке исходной строки, называемой **регионом** (region). По умолчанию регионом является вся вводимая последовательность символов.

Методы для операций с регионами приведены в табл. 1.2.3.

Таблица 1.2.3. Методы для операций с регионами класса **`Matcher`**

Метод	Действие и возвращаемое значение
`public Matcher region(int` *начальный-индекс*`,` `int` *конечный-индекс*`)`	Устанавливает границы региона и возвращает объект класса **`Matcher`** для подстроки, начинающейся с *начального-индекса* и заканчивающуюся индексом, на единицу меньшим, чем *конечный-индекс*.
`public int regionStart()`	Получает и возвращает индекс начала региона.
`public int regionEnd()`	Получает и возвращает индекс окончания региона.
`public Matcher useAnchoringBounds(boolean` *флажок*`)`	Если параметр *флажок* установлен в **`true`**, то для задания шаблона на границах региона можно использовать анкеры "**`^`**" и "**`$`**". В противном случае (если *флажок* установлен в **`false`**) соответствие анкерам "**`^`**" и "**`$`**" на границах региона не проверяется. По умолчанию значение *флажка* равно **`true`**.
`public Matcher useTransparentBounds(boolean` *флажок*`)`	Если параметр *флажок* установлен в **`true`**, границы региона являются прозрачными для шаблонов, содержащих граничные условия, а также «заглядывание вперед» и «заглядывание назад», в том числе и за границы региона. В противном случае (если *флажок* установлен в **`false`**) границы региона будут непрозрачными, т.е. операции с символами за границами региона

	запрещены. По умолчанию значение *флажка* равно **false**.
public boolean hasAnchoringBounds()	Проверяет, являются ли границы анкерными. При выполнении условия метод возвращает **true**, в противном случае – **false**.
public boolean hasTransparentBounds()	Проверяет, являются ли границы прозрачными. При выполнении условия метод возвращает **true**, в противном случае – **false**.

Методы поиска соответствий приведены в табл. 1.2.4.

Таблица 1.2.4. Методы поиска соответствий класса **Matcher**

Метод	Действие и возвращаемое значение
public Matcher region(int *начальный-индекс*, **int** *конечный-индекс***)**	Устанавливает границы региона и возвращает объект класса **Matcher** для подстроки, начинающейся с *начального-индекса* и заканчивающуюся индексом, на единицу меньшим, чем *конечный-индекс*.
public int regionStart()	Получает и возвращает индекс начала региона.
public boolean matches()	Выполняет для объекта класса **Matcher** поиск на соответствие всего региона, начиная с начала региона. Возвращает **true**, если соответствие найдено и **false** – в противном случае.
public boolean lookingAt()	Выполняет для объекта класса **Matcher** поиск, начиная с начала региона на наличие шаблона в регионе, но необязательно соответствия всего региона шаблону. Возвращает **true**, если соответствие найдено и **false** – в противном случае.
public boolean find()	Выполняет для объекта класса **Matcher** поиск, начиная с начала региона или, если предыдущий вызов метода был успешным, и объект класса **Matcher** не был сброшен, с первого символа после найденного предыдущего соответствия. Возвращает **true**, если соответствие найдено и **false** – в противном случае.
public boolean find(int *начальный-индекс***)**	Выполняется так же, как и метод **find()** без параметров , но поиск начинается не с начала региона, а заданного *начального-индекса*. Если необходимо найти все соответствия шаблону в строке, начиная с *начального-индекса*, то этот метод можно использовать только для поиска первого соответствия. Все остальные соответствия определяются с помощью метода **find()** без параметров.

public String group()	Возвращает предыдущее, найденное с помощью методов **matches()**,**lookingAt()** или **find()**, соответствие.
public String group(int *номер-группы***)** –	Возвращает предыдущее, найденное с помощью методов **matches()**,**lookingAt()** или **find()**, соответствие для заданного *номера-группы* (группы нумеруются слева направо, начиная с **1**; если задан **0**, метод выполняется так жк, как метод **group()** без параметров)
public int start()	Возвращает начальный индекс в строке поиска предыдущего соответствия, найденного с помощью методов **matches()**,**lookingAt()** или **find()**.
public int start(int *номер-группы***)**	Возвращает начальный индекс в строке поиска предыдущего соответствия, найденного с помощью методов **matches()**,**lookingAt()** или **find()**, для заданного *номера-группы* (группы нумеруются слева направо, начиная с **1**; если задан **0**, метод выполняется аналогично предыдущему методу **start()** без параметров).
public int end()	Возвращает конечный индекс в строке поиска предыдущего соответствия, найденного с помощью методов **matches()**,**lookingAt()** или **find()**.
public int end(int *номер-группы***)**	Возвращает конечный индекс в строке поиска предыдущего соответствия, найденного с помощью методов **matches()**,**lookingAt()** или **find()**, для заданного *номера-группы* (группы нумеруются слева направо, начиная с **1**; если задан **0**, метод выполняется аналогично предыдущему методу **start()** без параметров).
public int groupCount()	Возвращает количество групп в объекте класса **Matcher**, найденных с помощью методов **matches()**,**lookingAt()** или **find()**.
public Pattern pattern()	Возвращает шаблон, соответствующий объекту класса **Match**.
public Matcher usePattern(Pattern *новый-шаблон***)**	Заменяет шаблон для объекта класса **Match** на *новый-шаблон*.
public boolean hitEnd()	Возвращает **true**, если поиск соответствия затронул окончание строки поиска и **false** – в противном случае.
public boolean requireEnd()	Возвращает **true** и соответствие было найдено, то дальнейший поиск может привести к потере соответствия. Если метод возвращает **false** и соответствие было найдено, то дальнейший поиск не приведет привести к потере

	соответствия. Если соответствия не было найдено, метод не имеет значения.

Методы класса **Matcher** позволяют не только выполнить поиск в строке по заданному шаблону, но и заменить найденные соответствия заданными последовательностями символов – строками замены.

Методы замены приведены в табл. 1.2.5.

Таблица 1.2.5. Методы замены класса **Matcher**

Метод	Действие и возвращаемое значение
public String replaceFirst(String *строка-замены***)**	Заменяет только первое соответствие в строке поиска *строкой-замены* и возвращает измененную строку.
public String replaceAll(String *строка-замены***)**	Заменяет все соответствия в строке поиска *строкой-замены* и возвращает измененную строку.
public Matcher appendReplacement(StringBuffer *новая-строка* **,** **String** *строка-замены***)**	Формирует *новую-строку* по следующему алгоритму: • пересылает символы строки поиска в *новую-строку*, начиная с конечной позиции (append position) до символа на единицу меньшего, чем символ, определяемого методом **start()** объекта **Match**; • затем к новой строке добавляется *строка-замены*; после этого конечная позиция в новой строке становится равной позицией, определяемой методом **end()** объекта **Match**. В начале просмотра и замены значение конечной позиции равно **0**.
public StringBuffer appendTail(StringBuffer *новая-строка***)**	Пересылает символы строки поиска в *новую-строку*, начиная с конечной позиции и до конца строки поиска. Этот метод используется вместе с методом **appendReplacement()** для завершения процесса поиска и замены в строке.
public static String quoteReplacement(String *строка-замены*)	Преобразует строку замены в строковую форму, в которой метасимволы, такие как **"$"**, рассматриваются как обычные символы (действует аналогично методу **quote()** класса **Pattern**) и возвращает измененную строку.
public Matcher reset()	Сбрасывает все хранимые в объекте класса **Matcher** данные, устанавливает конечную позицию в **0**, регионом становится вся строка

	поиска (однако на состояние анкерной границы и границы прозрачности сброс не влияет) и возвращает измененный объект класса **Matcher**;
public Matcher reset(CharSequence *строка-поиска***)**	Выполняет те же действия, что и метод **reset()** без параметров, но задает для объекта новую последовательность символов поиска.

1.2.10. Класс PatternSyntaxException

Класс **PatternSyntaxException** бросает исключение, если регулярное выражение (шаблон) содержит синтаксическую ошибку. Методы этого класса приведены в табл. 1.2.6.

Таблица 1.2.6. Методы класса **PatternSyntaxException**

Метод	Возвращаемое значение
public String getPattern()	Шаблон, содержащий ошибку.
public String getDescription()	Описание ошибки.
public int getIndex()	Позиция символа ошибки в шаблоне.
public String getMessage()	Сообщение об ошибке, содержащее все перечисленные выше компоненты: описание ошибки и ее индекс, шаблон, содержащий ошибку и визуальную индикацию индекса ошибки внутри шаблона.

1.2.11. Методы класса String для работы с регулярными выражениями

Для работы с регулярными выражениями в классе String определены методы, приведенные в табл. 1.2.7.

Таблица 1.2.7. Методы для работы с регулярными выражениями класса **String**

Метод	Действие и возвращаемое значение
public boolean matches(String *шаблон***)**	Если объект класса **String** соответствует шаблону, возвращает значение **true**, в противном случае возвращает **false** (действует аналогично методу **matches()** класса **Pattern**).
public String[] split(String *шаблон***)**	Создает для объекта класса **String** массив строк, разделенный на элементы по заданному *шаблону* (действует аналогично

	соответствующему методу **split()** класса **Pattern**)
public String[] split(String *шаблон*, **int** *предел***)**	Создает для объекта класса **String** массив строк, разделенный на элементы по заданному *шаблону*, и с заданным в параметре *предел* количеством элементов (если значение параметра больше или равно количеству элементов, либо меньше 0, выводятся все элементы, если меньше количества элементов – все оставшиеся соответствия выводятся в последнем элементе массива) (действует аналогично соответствующему методу **split()** класса **Pattern**).
public String replaceFirst(String *шаблон*, **String** *строка-замены*)	Заменяет в объекте **String** первое соответствие *шаблону* на *строку-замены* и возвращает измененную строку (действует аналогично методу **replaceFirst ()** класса **Match**).
public String replaceAll(String *шаблон*, **String** *строка-замены*)	Заменяет в объекте **String** все соответствия *шаблону* на *строку-замены* и возвращает измененную строку (действует аналогично методу **replaceAll()** класса **Match**).

2. Варианты заданий

№	Задачи	№	Задачи	№	Задачи	№	Задачи
1	1,5,16,40	**7**	8,13,18,22	13	2,14,28,38	19	8,15,22,34
2	2,13,17,41	**8**	9,14,23,43	14	3,5,29,39	20	9,13,21,35
3	3,15,18,42	**9**	5,10,24,44	15	4,15,26,30	21	10,14,20,36
4	4,14,19,43	**10**	11,15,25,35	16	5,13,25,31	22	5,11,19,37
5	5,6,20,44	**11**	12,14,26,36	17	5,7,24,32	23	12,15,18,38
6	7,15,17,21	**12**	1,13,27,37	18	14,6,23,33	24	1,13,17,39

1. Вывести таблицу преобразований целых десятичных чисел в интервале от 10 до 100 с шагом 20 в 16-ное представление.
2. Вывести таблицу преобразований целых десятичных чисел в интервале от min до max с шагом step в 16-ном представлении. Параметры задачи задаются как параметры командной строки.

3. Вывести таблицу преобразований целых десятичных чисел в интервале от min до max с шагом step в 16-ном представлении. Параметры задачи вводятся пользователем в ходе диалога с программой за один раз.
4. Обеспечьте вывод команд меню в виде нумерованного списка и запрос на ввод номера, что приводит к выводу сообщения о выполнении команды.
5. Создайте программу для шифрования\расшифровки текста методом Цезаря. В нем ключом является целое число, а шифрование\расшифровка заключается в суммировании\ вычитании кодов символов открытого текста\криптотекста с ключом.
6. Создайте программу для определения кода введенного символа в стандарте Юникод. Обеспечьте цикличность выполнения программы до ввода знака равенства (что приводит к выходу из программы).
7. Создайте программу для определения кода введенного символа в 16-ном представлены в формате '\ uXXXX ". Обеспечьте цикличность выполнения программы до ввода символа пробел. Проверьте правильность работы программы, воспользовавшись ею для получения кодов символов в любом слове (например, "cat") и выведя на экран слово из полученных кодов.
8. Создайте программу, которая принимает произвольный текстовую строку, а возвращает другой, в котором символы располагаются в обратном порядке.
9. Создайте программу для проверки текста на палиндром. Регистр символов и пробелы не учитываются.
10. Создайте программу для определения количества вхождений указанного символа в заданном тексте. Работа программы прекращается после введения символа "-".
11. Создайте программу для определения позиций вхождений указанного символа в заданном тексте. Работа программы прекращается после введения символа "-".

12. Создайте программу, которая принимает текстовую строку и выдает другой, в котором буквы переставлены в случайном порядке.
13. Создайте программу, которая осуществляет шифрование / расшифровка методом простой замены. В нем каждый символ незашифрованного текста из исходного алфавита заменяется другим из алфавита, символы в каком представлены в соответствии с ключом шифрования.
14. Создайте программу, которая осуществляет шифрование / расшифровка методом гаммирования. В нем ключом является текстовая строка такой же длины, как и открытый текст, а шифрование / расшифровка заключается в суммировании / вычитании кодов символов открытого текста / криптотексте с кодами символов ключа.
15. Создайте программу, которая осуществляет шифрование / расшифровка модифицированным методом Цезаря. В нем ключом является лозунг, который многократно повторяется до тех пор, чтобы достичь длины открытого текста, а шифрование / расшифровка заключается в суммировании / вычитании кодов символов открытого текста / криптотексте с соответствующими кодами символов лозунга.
16. Анализ типов аргументов, задаваемых при запуске программы. Если аргумент является правильным целым числом (шаблон: одна или несколько цифр, первым символом может быть либо цифра, либо знак "+" или "-"), то тип аргумента "Integer", иначе "String". Программа выводит количество заданных аргументов и, для каждого аргумента, его тип и значение.
17. Анализ типов аргументов, задаваемых при запуске программы. Если аргумент является десятичным числом с целой и дробной частью (шаблон: состоит из одной или нескольких цифр, одной десятичной точки, которая может быть в начале, в середине или в конце числа, и, кроме того, первым символом числа может быть знак "+" или "-"), то тип аргумента "Decimal", иначе "String". Программа выводит количество

заданных аргументов и, для каждого аргумента, его тип и значение.

18. Анализ типов аргументов, задаваемых при запуске программы. Если аргумент является числом с плавающей точкой (шаблон: состоит из мантиссы – одна или несколько цифр, возможно со знаком "+" или "-", которая может содержать десятичную точку в начале, середине или конце, а также порядка – целого числа со знаком "+" или "-" или без знака, разделителем между мантиссой и порядком служит символ "e" или символ "E"), то тип аргумента "Float", иначе "String". Программа выводит количество заданных аргументов и, для каждого аргумента, его тип и значение.
19. Анализ типов аргументов, задаваемых при запуске программы. Если аргумент является целым восьмеричным числом (шаблон: состоит из цифр от 0 до 7, причем первой цифрой должен быть 0), то тип аргумента "Octal", иначе "String". Программа выводит количество заданных аргументов и, для каждого аргумента, его тип и значение.
20. Анализ типов аргументов, задаваемых при запуске программы. Если аргумент является целым шестнадцатеричным числом (шаблон: состоит из цифр от 0 до 9 и букв A(a), B(b), C(c), D(d), E(e),F(f), перед числом должны стоять символы "0X" или "0x"), то тип аргумента "Hexadecimal", иначе "String". Программа выводит количество заданных аргументов и, для каждого аргумента, его тип и значение.
21. Анализ типов аргументов, задаваемых при запуске программы. Если аргумент является целым двоичным числом (шаблон: одна и более цифр 0 и 1), то тип аргумента "Binary", иначе "String". Программа выводит количество заданных аргументов и, для каждого аргумента, его тип и значение.
22. Анализ типов аргументов, задаваемых при запуске программы. Если аргумент имеет вид "имя=значение", то он является ключевым параметром (тип "Keyed"), если аргумент имеет вид "-значение" или "/значение", то он является опцией (тип "Optional") и если имеет вид "значение", то является

непосредственным параметром (тип "Immediate"). Шаблон для значения: одна или несколько цифр и букв (включая буквы кириллицы). Программа выводит количество заданных аргументов и, для каждого аргумента, его тип и значение (для ключевых параметров дополнительно выводится имя параметра).

23. Анализ типов аргументов, задаваемых при запуске программы. Если аргумент имеет вид "значение", то он является одиночным параметром (тип "Single"), если аргумент имеет вид "значение, значение,… ", то он является списком (тип "List"). Программа выводит количество заданных аргументов и, для каждого аргумента, его тип и значение (для каждого параметра-списка при выводе его значение преобразуется в массив типа String, каждый элемент которого содержит элемент списка и выводится в цикле по элементам).
24. Анализ типов аргументов, задаваемых при запуске программы. Если аргумент является правильным идентификатором Java (шаблон: состоит из латинских букв, цифр и символов "$" и "_", считающихся буквами, и, кроме того, первый символ является буквой), то его тип "Identifier", если аргумент является ключевым словом Java (для примера задать несколько ключевых слов Java, "if", "for", "while", "do" и "else"), то его тип "Keyword", иначе его тип считается "Illegal". Программа выводит количество заданных аргументов и, для каждого аргумента, его тип и значение.
25. Анализ аргументов, задаваемых при запуске программы. Программа ищет наибольшую общую подстроку, содержащуюся во введенных аргументах (шаблон аргумента: строка либо латинских букв, либо букв кириллицы). Программа выводит количество заданных аргументов, их значения и наибольшую общую подстроку или сообщение о том, что общей подстроки нет.
26. Анализ аргументов, задаваемых при запуске программы. Программа определяет, какие символы содержатся во введенных аргументах (например, аргументы "abc", "cf", "bfc"

содержат символы "abcf"). Шаблон аргумента: строка либо латинских букв, либо букв кириллицы. Программа выводит количество заданных аргументов, значения аргументов и строку символов, содержащихся в аргументах.

27. Анализ аргументов, задаваемых при запуске программы. Программа удаляет из аргументов все повторяющиеся символы, кроме одного (например, аргумент "abcadc", преобразуется в "abcd"). Шаблон аргумента: строка либо латинских букв, либо букв кириллицы. Программа выводит количество заданных аргументов, их значения и преобразованные значения аргументов.
28. Анализ аргументов, задаваемых при запуске программы. Программа определяет строку символов встречающиеся только в одном из аргументов (например, для аргументов "agc", "cf", "bfc" такой строкой будет строка "gb"). Шаблон аргумента: строка либо латинских букв, либо букв кириллицы. Программа выводит количество заданных аргументов, их значения и найденную строку символов или сообщение о том, что таких символов нет.
29. Вариант 2-14
30. Анализ аргументов, задаваемых при запуске программы. Программа определяет, сколько раз символ встречается во введенных аргументах (например, для аргументов "agc", "cf", "bfc" символы "a", "g" и "b" встречаются один раз, символ "f" – два раза и символ "c" – три раза). Шаблон аргумента: строка либо латинских букв, либо букв кириллицы. Программа выводит количество заданных аргументов, их значения и для каждого символа – частоту его повторения в аргументах.
31. Анализ аргументов, задаваемых при запуске программы. Программа сортирует введенные аргументы (шаблон аргумента: строка латинских букв) по первому символу аргумент. Программа выводит количество заданных параметров, их значения и список отсортированных аргументов.

32. Анализ аргументов, задаваемых при запуске программы. Программа определяет тип аргумент – целое число или строка (шаблон: целым числом считается строка, которая содержит цифры и, кроме того, первым символом строки может быть цифра, знак "+" или "-"). Затем среди аргументов-чисел ищется максимальное число. Программа выводит количество заданных аргументов, их значения, количество аргументов-чисел и значение максимального числа.
33. Анализ аргументов, задаваемых при запуске программы. Программа определяет, имеются ли среди введенных аргументов одинаковые аргументы и число их повторения. Шаблон аргумента: строка либо цифр, либо латинских букв. Программа выводит количество заданных аргументов, значения повторяющихся аргументов и количество их повторения или сообщение о том, что повторяющихся аргументов нет.
34. Анализ аргументов, задаваемых при запуске программы. Программа переставляет введенные аргументы в порядке возрастания их длины. Шаблон аргумента: строка либо цифр, либо латинских букв, либо букв кириллицы. Программа выводит количество заданных аргументов, их значения, а также список значений аргументов в порядке возрастания их длины.
35. Анализ аргументов, задаваемых при запуске программы. Программа определяет, какие из введенных аргументов содержат строку, задаваемую в качестве первого параметра. Шаблон аргумента: строка либо цифр, либо латинских букв, либо букв кириллицы. Программа выводит количество заданных аргументов (без учета первого аргумента) и аргументы, содержащие заданную подстроку или сообщение о том, что данная строка не содержится во введенных аргументах.
36. Анализ аргументов, задаваемых при запуске программы. Программа удаляет из массива введенных аргументов все повторяющиеся аргументы, кроме одного (например, из

аргументов "ab", "cd", "ab" будут оставлены аргументы "ab" и "cd"). Шаблон аргумента: строка латинских букв. Программа выводит количество заданных аргументов, их значения, а также количество разных аргументов и их значения.

37. Анализ аргументов, задаваемых при запуске программы. Программа определяет тип аргумент – целое число или строка (шаблон: целым числом считается строка, которая содержит цифры и, кроме того, первым символом строки может быть цифра, знак "+" или "-"). Затем аргументы-числа сортируются в порядке возрастания их значения. Программа выводит количество заданных аргументов, их значения, количество аргументов-чисел и их значения в порядке возрастания.
38. Преобразование аргументов, задаваемых при запуске программы. Программа определяет тип аргумента – двоичное число без знака или строка (шаблон: двоичным числом без знака считается строка, которая содержит одну или более цифр 0 и 1). Введенные аргументы-числа преобразуются в шестнадцатеричные числа (каждые четыре цифры двоичного числа преобразуются в одно шестнадцатеричное, поэтому, при необходимости, в значение аргумента добавляются нули до длины, кратной 4). Программа выводит количество заданных аргументов, их значения, а также количество аргументов-чисел и их шестнадцатеричные значения.
39. Преобразование аргументов, задаваемых при запуске программы. Программа определяет тип аргумент – шестнадцатеричное число без знака (шаблон: шестнадцатеричным числом без знака считается строка, которая содержит цифры от 0 до 9 и буквы A(a), B(b), C(c), D(d), E(e),F(f)) или строка. Введенные аргументы-числа преобразуются в двоичные числа. Программа выводит количество заданных аргументов, их значения, а также количество аргументов-чисел и их двоичные значения.
40. Преобразование аргументов, задаваемых при запуске программы. Программа преобразует русские и латинские буквы в аргументах в верхний регистр (если они являются

строчными). Шаблон аргумента: либо строки латинских букв, либо строки букв кириллицы. Программа выводит количество заданных аргументов, их значения, а также новые значения аргументов.

41. Преобразование аргументов, задаваемых при запуске программы. Программа определяет тип аргумента – шестнадцатеричное число без знака (шаблон: шестнадцатеричным числом без знака считается строка, которая содержит цифры от 0 до 9 и буквы A(a), B(b), C(c), D(d), E(e),F(f)) или строка. Введенные аргументы-числа преобразуются в десятичные числа (каждая i-ая цифра шестнадцатеричного числа преобразуется в десятичное число Ni по формуле Ni=16n-i-1, где n – количество цифр в числе; i = 0,n – индекс цифры в числе, искомое число является суммой всех Ni). Программа выводит количество заданных аргументов, их значения, а также количество аргументов-чисел и их десятичные значения.
42. Анализ аргументов, задаваемых при запуске программы. Аргумент имеет следующий формат: имя-типа-или-имя-класса.имя-переменной-или-имя-метода и представляет собой обращение к переменной или вызов метода для объекта или класса в Java. Шаблон аргумента: латинские буквы, цифры и символы "$" и "_", считающиеся буквами, и, кроме того, первый символ является буквой. Если первый символ обращения – заглавная буква, то выводится тип "Static", если строчная буква – выводится "Object". Если имя переменной или метода начинается со строчной буквы, то если имя содержит символы "("и ")", то выводится тип обращения "Method", иначе выводится тип обращения "Variable". Если какое-либо из приведенных условий не выполняется, то выводится тип обращения "Illegal". Программа выводит количество заданных аргументов и, для каждого аргумента, его тип, тип обращения и значение.
43. Анализ типов аргументов, задаваемых при запуске программы. Если аргумент является правильным идентификатором языка

C (шаблон: состоит из латинских букв, цифр и символа и "_", считающегося буквой, и, кроме того, первый символ является буквой), то его тип "Identifier", если параметр является ключевым словом C (для примера задать несколько ключевых слов C, "if", "for", "while", "do" и "else"), то его тип "Keyword", иначе его тип считается "Illegal". Программа выводит количество заданных аргументов и, для каждого аргумента, его тип и значение

44. Анализ типов аргументов, задаваемых при запуске программы. Если аргумент является числовым литералом, т.е. начинается с цифры, то определяется его тип ("Integer" или "Real"), если аргумент заключен в одиночные апострофы и содержит один символ, то его тип – "Character", если аргумент заключен в двойные апострофы, то его тип – "String". Если не одно из условий не выполняется, то тип аргумента – "Identifier". Программа выводит количество заданных аргументов и, для каждого аргумента, его тип и значение.

3. Ход выполнения работы

1. В IntelliJ IDEA создайте проект **Maven**.
2. Создайте классы программ в соответствии с вариантом индивидуального задания.
3. Создайте unit-тесты для тестирования созданной программы.
4. Выполните тестирование программ.
5. Оформите отчет о проделанной работе.