**Строки**

Для работы с текстом служат два встроенных Java-класса: String и StringBuffer. Как и в языке С#, объекты класса String изменять нельзя, а объекты класса StringBuffer – можно. В некоторых случаях можно использовать массив символов char [].

Классы String и StringBuffer определены в базовом пакете java.lang, который доступен по умолчанию, поэтому для создания объекта класса String или StringBuffer импорт пакетов выполнять не нужно. Оба класса определены как неизменяемые (final), то есть они не могут быть суперклассами для наследования.

## **Объекты класса String**

**Конструкторы класса String:**

* Конструктор создания пустой строки - конструктор без параметров. Пример команды создания объекта класса String со значением в виде пустой строки:

String s=new String();

* Конструктор создания текстовой строки на основе символьного массива. В этом случае аргументом конструктору передается имя массива символов. Результатом является текст, составленный из всех символов массива в порядке их размещения в массиве. Пример создания текстовой строки на основе символьного массива:

char symbols[]={'a','b','c'};

String s=new String(symbols);

В этом конструкторе, помимо имени массива, можно указать индекс элемента массива, начиная с которого будет извлекаться строка, а также длину строки в символах. Например, так:

char symbols={'a','b','c','d','e','f'};

String s=new String(symbols,2,3); // s="cde"

* Конструктор копирования объекта. Параметром конструктора является переменная текстового типа, ссылающаяся на уже существующий текстовый объект или текстовый литерал. В результате создается новый объект с таким же текстовым значением, как и исходный. Например:

String obj=new String("Текстовая строка");

String s=new String(obj);

Некоторые методы класса String

|  |  |
| --- | --- |
| **Название метода** | **Описание** |
| length()  compareTo(String anotherString)  compareToIgnoreCase(String str)  regionMatches(boolean ignoreCase, int ind1, String other, int ind2, int len)  regionMatches(int ind1, String other, int ind2, int len)  concat(String str)  contains(CharSequence s)  endsWith(String suffix)  startsWith(String prefix)    startsWith(String prefix, int toffset)    equals(Object anObject)  equalsIgnoreCase()  getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)  indexOf(int ch)  indexOf(int ch, int fromIndex)  indexOf(String str)  indexOf(String str, int fromIndex)  lastIndexOf(int ch)  lastIndexOf(int ch, int fromIndex)  lastIndexOf(String str)  lastIndexOf(String str, int fromIndex)  replace(char oldChar, char newChar)  replace(CharSequence target, CharSequence replacement)  substring(int beginIndex, int endIndex)  toLowerCase()  toUpperCase()  trim()  valueOf(a) | возвращает длину строки  лексиграфическое сравнение строк; лексиграфическое сравнение строк без учета регистра символов;  тест на идентичность участков строк, можно указать учет регистра символов;  Здесь ind1 – индекс начала подстроки данной строки, ind2 – индекс начала подстроки другой строки str. Результат false получается в следующих случаях:   * хотя бы один из индексов ind1 или ind2 отрицателен; * хотя бы одно из ind1 + len или ind2 + len больше длины соответствующей строки; * хотя бы одна пара символов не совпадает.   тест на идентичность участков строк;  возвращает соединение двух строк;  проверяет, входит ли указанная последовательность символов в строку;  проверяет завершается ли строка указанным суффиксом;  проверяет, начинается ли строка с указанного префикса;  проверяет, начинается ли строка в указанной позиции с указанного префикса;  проверяет идентична ли строка указанному объекту;  возвращает символьное представление участка строки;  поиск первого вхождения символа в строке;  поиск первого вхождения символа в строке с указанной позиции;  поиск первого вхождения указанной подстроки;  поиск первого вхождения указанной подстроки с указанной позиции;  поиск последнего вхождения символа;  поиск последнего вхождения символа с указанной позиции;  поиск последнего вхождения строки;  поиск последнего вхождения строки с указанной позиции;  замена в строке одного символа на другой;  замена одной подстроки другой;  возвращает заданную подстроку исходной строки;  преобразовать строку в нижний регистр;  преобразовать строку в верхний регистр;  удаляет начальные и конечные пробелы;  статические методы преобразования различных типов в строку. |

## Класс StringBuffer

У класса StringBuffer несколько конструкторов:

* конструктор без аргументов StringBuffer()

При использовании конструктора без аргумента создается объект класса StringBuffer со значением в виде пустой текстовой строки, а также автоматически резервируется память еще для 16-ти символов (буфер памяти).

* конструктор с числовым аргументом

Чтобы в явном виде указать размер буфера памяти при создании объекта класса StringBuffer, используют конструктор с числовым аргументом.

* конструктор с текстовым аргументом (типа String или StringBuffer).

Для создания копии уже существующего текстового объекта применяют конструктор с текстовым аргументом.

Некоторые методы для работы с объектами класса StringBuffer

|  |  |
| --- | --- |
| **Название метода** | **Описание** |
| length()  capacity()  ensureCapacity()  setLength()  charAt()  setCharAt()  getChars()  append()  insert()  reverse()  delete()  deleteCharAt()  replace() | возвращает текущую длину текстовой строки  возвращает выделенный для данной текстовой переменной объем памяти (в символах, то есть количество символов, которые можно записать в текстовую строку)  выделяет память для уже созданного объекта. Размер выделяемой памяти указывается аргументом метода  устанавливает длину текстовой строки  возвращает символ в строке с указанным индексом  У метода два аргумента: индекс символа в строке и символьное значение. Символ строки с заданным первым аргументом индексом заменяется символом, указанным вторым аргументом метода. Изменяется исходная строка  Копирование строки в символьный массив. Аргументы метода:  начальный индекс подстроки и индекс первого не входящего в подстроку символа, массив, в который выполняется копирование, а также индекс элемента в этом массиве, начиная с  которого в массив производится посимвольное копирование  подстроки  добавляет заданный текст в конец исходной строки  выполняет вставку текста, указанного вторым аргументом метода в строку вызова. Первым аргументом метода указывается индекс начала вставки подстроки  меняет порядок следования символов в строке вызова.  удаляет подстроку из строки вызова. Первым аргументом метода указывается индекс начала удаляемой подстроки, вторым – индекс первого после удаляемой подстроки символа  удаляет символ с заданным индексом из строки вызова  удаляет подстроку из строки вызова и на ее место вставляет другой текст. Первым аргументом метода указывается индекс начала удаляемой подстроки, вторым – индекс первого после удаляемой подстроки символа. Третий аргумент метода – текст, вставляемый вместо удаленной подстроки. |

## Форматирование строк

В центре системы поддержки форматированного вывода находится класс Formatter. Он предлагает преобразования формата, позволяющие отображать числа, строки, время и даты практически в любом виде по вашему желанию.

Для форматирования вывода нужно создать объект Formatter.

Методы класса Formatter

|  |  |
| --- | --- |
| **Название метода** | **Описание** |
| format(String fтtString, Object ... args) | Форматирует apгументы, переданные в args, в соответствие со спецификаторами формата, содержащимися в fтtString. Возвращает вызываемый объект. |

**Спецификаторы формата:**

**%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: **%е** или**%f**

**%o –** Восьмеричное целое значение аргумента

**%n –** Вставка символа новой строки (перевод строки)

**%s –** Строковое представление аргумента

**%t –** Время и дата

**%x –** Шестнадцатеричное целое значение аргумента

**%% –** Вставка знака **%**

Пример:

Formatter fmt = new Formatter();

fmt.fоrmаt("Форматировать %s очень просто: %d %f", "с помощью Java", 10, 98.6);

­ Приведенная кодовая последовательность создает объект Formatter, содержащий следующую строку:

*­Форматировать с помощью Java очень просто: 10 98.600000*

В этом примере спецификаторы формата %s, %d и %f замещаются аргументами, следующими за строкой формата. То есть %s заменяется на "с помощью Java", %d ­ на 10, а %f ­ на 98.6. Все остальные символы используются, как есть. Спецификатор параметра %s указывает строку, %d ­ целое число, а %f, означает число с плавающей точкой.

**Указание минимальной ширины поля**

Целое число, помещенное между символом % и кодом преобразования формата, выступает в качестве спецификатора минимальной ширины. Он дополняет вывод пробелами, чтобы обеспечивать заданную минимальную длину. Если строка или число получаются длиннее, чем этот заданный минимум, они будут напечатаны полностью. По умолчанию дополнение осуществляется пробелами. Если вы хотите дополнять нулями, поместите 0 перед спецификатором ширины поля. Например, %05d дополнит число, состоящее из менее чем 5 разрядов, нулями таким образом, чтобы его общая ширина была равна пяти. Спецификатор ширины поля может применяться вместе со всеми спецификаторами формата, кроме %n.

Пример:

Formatter fmt ­ new Formatter();

fmt.format("|%fl%nl%12fl%nl%012f|", 10.12345, 10.12345, 10.12345);

System.out.println(fmt);

Эта программа генерирует следующий вывод:

|10.123450|

| 10.123450|

|00010.123450|

Первая строка отображает число 10,12345 с шириной по умолчанию. Вторая отображает это значение в 12-ти символьном поле. Третья строка отображает значение в 12-ти символьном поле, дополняя его ведущими нулями.

Минимальный модификатор ширины поля часто используется для генерации таблиц, состоящих из строк и столбцов.

**Указание точности**

Спецификатор точности может быть применим к спецификаторам формата %f, %е, %g и %s. Он следует за спецификатором минимальной ширины поля (если таковой имеется) и состоит из точки с последующим целым числом. Его конкретное значение зависит от типа данных, к которому он применяется.

Когда вы применяете спецификатор точности к данным с плавающей точкой с применением спецификаторов преобразования %f или %е, то он определяет количество отображаемых десятичных разрядов. Например, %10.4 f отображает число, по меньшей мере, в 10 символов шириной с четырьмя разрядами после запятой. При использовании %g точность определяет количество значащих десятичных разрядов. Точность по умолчанию составляет 6 знаков после запятой.

В применении к строкам спецификатор точности задает максимальную ширину поля. Например, %5. 7s отображает строку длиной минимум в пять символов и не превышающую семь символов. Если строка длиннее максимальной ширины, конечные символы будут усечены.

**Использование индекса аргументов**

Formatter включает очень удобное средство, позволяющее указать аргумент, к которому должен применяться конкретный спецификатор формата. Обычно порядок аргументов и спецификаторов формата совпадает слева направо. То есть первый спецификатор формата относится к первому аргументу, второй – ко второму и т.д. Однако, используя индекс аргумента, вы можете явно управлять тем, к какому из аргументов относится спецификатор формата.

Индекс аргумента следует сразу за % в спецификаторе формата. Он имеет следующий вид:

n$

Здесь n – индекс нужного аргумента, начиная с 1. Рассмотрим следующий пример.

fmt.format("%3$d%1$d%2$d", 10, 20, 30);

Этот код порождает строку: 30 10 20

В этом примере первый спецификатор формата соответствует 30, второй – 10, а третий – 20.

Одной из выгод индексирования аргументов является то, что оно позволяет повторно использовать аргумент, не указывая его дважды. Например, рассмотрим следующую строку:

fmt.format("%d в шестнадцатеричном формате равно %l$х", 255);

Она порождает следующий результат: *255 в шестнадцатеричном формате равно ff* .

Как видите, аргумент 255 используется с обоими спецификаторами формата.

## Задания для самостоятельного выполнения:

1. Реализуйте метод transliteration принимающий в качестве аргумента исходную строку и строку-разделитель (divider по умолчанию " ") и возвращающий преобразованную строку из латинских символов. Создайте словарь символов соответствия алфавитов: "а": "a", "б": "b", "в": "v", "г": "g", "д": "d", "е": "e", "ё": "e", "ж": "zh", "з": "z", "и": "i", "й": "i", "к": "k", "л": "l", "м": "m", "н": "n", "о": "o", "п": "p", "р": "r", "с": "s", "т": "t", "у": "u", "ф": "f", "х": "h", "ц": "c", "ч": "ch", "ш": "sh", "щ": "sh'", "ъ": "", "ы": "i", "ь": "", "э": "e", "ю": "yu", "я": "ya".

Пример:

transliteration("Иван Иванов") //Ivan Ivanov

transliteration("Иванов John","\_") //Ivanov\_John.

1. Реализуй метод truncate, усекающий исходную строку до указанного числа символов (по умолчанию 16) и возвращающий усеченную строку с заполнителем "..." (если строка была усечена). Если последний символ усеченной строки является пробелом – удалить его и добавить заполнитель

Пример:

truncate("39 новых фич, которые будут доступны в Java 12") //39 новых фич, ко...

truncate("39 новых фич, которые будут доступны в Java 12", 9) //39 новых…

truncate("A ", 3) //A

1. Пользователь вводит некоторый текст и выбирает один из способов изменения регистра введенного текста:
   1. Как в предложениях
   2. все строчные
   3. ВСЕ ПРОПИСНЫЕ
   4. Начинать С Прописных
   5. иЗМЕНИТЬ РЕГИСТР

Для реализации использовать шаблон Стратегия.