Строки и регулярные выражения

Строки

**Реализация строк представлена тремя основными классами: String, StringBuffer, StringBuilder**

**String** – это класс который позволяет работать со строками. Он является финальным, не может иметь подклассом или быть изменен.

**Строки** не требуют явного конструктора, и если при копировании строки он не требуется, то лучше его не использовать.

**Строки** можно создавать от массива символов или байт.

**Методы:**

**startsWith(“Hello”)** - проверяет начинается ли заданный объект типа String с указанной символьной строки

**endsWith(“World”)** - проверяет завершается ли объект типа String заданной подстрокой

**contains(“ ”)** - позволяющий проверить, содержит ли String другую подстроку или нет

**equalsIgnoreCase(“HeLLo wOrlD!”)** - сравнить без учета регистра

**indexOf(“oo”)** – поиск

**lastIndexOf(“o”, 10)** - поиск идет от конца к началу, начиная с полученного индекса

**replace(“Hello”,”Goodbye”)** – замена все совпадения

**replaceFirst(“^Hello”,”Goodbye”)** – замена первой подстроки данной строки.

**replaceAll(“\\w”,”x”)** - замена по регулярки всех совпадений

**trim( )** - отсечь пробелы

**substring(6, 11)** - получить под строку

**split(“ “)** - разделить на массив

**concat()** – сливает 2 строки и выдает новую.

**format()** – позволяет создавать форматированные строки.

**String.valueOf()** – переводит число в строку

**Integer.toString()** - переводит число в строку

**StringBuffer** – это класс который позволяет более эффективно модифицировать строки. Он в свою очередь уже является изменяемый, но при этом его можно использовать многопоточно, так-как все требуемые методы синхронны. В основном он используется для многократного выполнения методов добавления, удаления и вставки.

**Методы(основные):**

**length()** - Метод позволяет получить текущую длину объекта.

**capacity()** - Метод позволяет получить текущий объём выделенной памяти.

**ensureCapacity()** - Можно предварительно выделить место для определённого количества символов, если собираетесь добавлять большое количество маленьких строк.

**setLength(int length)** - Устанавливает длину строки. Значение должно быть неотрицательным.

**charAt(int index) -** извлекает значение отдельного символа.

**setCharAt(int index, char ch)** – установить новое значение символа.

**getChars()** - Позволяет скопировать подстроку из объекта класса StringBuffer в массив. Необходимо позаботиться, чтобы массив был достаточного размера для приёма нужного количества символов указанной подстроки.

**append()** - Метод соединяет представление любого другого типа данных. Есть несколько перегруженных версий.

**insert()** - Вставляет одну строку в другую. Также можно вставлять значения других типов, которые будут автоматически преобразованы в строки. Вам надо указать индекс позиции, куда будет вставляться строка.

**reverse()** - Позволяет изменить порядок символов на обратный.

**delete()** - удаляет последовательность символов, вам надо задать индекс первого символа, который надо удалить, а также индекс символа, следующего за последним из удаляемых.

**deleteCharAt()** - удаляет один символ из указанной позиции.

**replace()** - Позволяет заменить один набор символов на другой. Нужно указать начальный и конечный индекс и строку замены.

**substring()** - Позволяет получить часть содержимого. Есть две формы метода. В первом случае нужно указать индекс начала позиции, с которой нужно получить подстроку. Во втором варианте указывается начальный индекс и конечный индекс, если нужно получить текст из середины строки.

**StringBuilder** – это класс который идентичен классу StringBuffer, но он гораздо быстрей и не синхронизирован. Из-за чего его он эффективнее, но его нужно использовать только когда строка коммуницирует с одним потоком.

Регулярные выражения

**Регулярные выражения – это средство для обработки текста.**

**Таблицы поддерживаемых символьных классов:**

**[abc] -** a, b, или c (простой класс)

**[^abc] -** Любой символ, кроме a, b и c (отрицание)

**[a-zA-Z] -** от a до z, или от A до Z, включительно (диапазон)

**[a-d[m-p]] -** от a до d, или от m до p: [a-dm-p] (объединение)

**[a-z&&[def]] -** d, e, или f (пересечение)

**[a-z&&[^bc]] -** от a до z, исключая b и c: [ad-z] (вычитание)

**[a-z&&[^m-p]] -** от a до z, и не от m до p: [a-lq-z] (вычитание)

**Предопределённые классы символов**

**. -** любой символ (может совпадать и с символами конца строки)

**\d -** цифра: [0-9].

**\D -** не цифра: [^0-9].

**\s -** пробельный символ: [ \t\n\x0B\f\r].

**\S -** не пробельный символ: [^\s].

**\w -** символ слова (английская буква, подчёркивание или цифра): [a-zA-Z\_0-9].

**\W -** не словарный символ: [^\w].

**Квантификаторы** позволяют вам указать количество вхождений, с которыми будет совпадать шаблон.

|  |  |  |  |
| --- | --- | --- | --- |
| **Жадный** | **Ленивый** | **Собственнический** | **Значение** |
| X? | X?? | X?+ | X, один раз или ни разу |
| X\* | X\*? | X\*+ | X, ноль или более раз |
| X+ | X+? | X++ | X, один или более раз |
| X{n} | X{n}? | X{n}+ | X, точно *n* раз |
| X{n,} | X{n,}? | X{n,}+ | X, хотя бы *n* раз |
| X{n, m} | X{n, m}? | X{n, m}+ | X, хотя бы *n*, но не более *m* раз |

**Жадные квантификаторы** считаются «жадными», потому что они пытаются сначала прочитать (или съесть) всю строку для первого сопоставления. Если первое сопоставление (вся входная строка) не удалось, то происходит сдвиг на один символ назад по входной строке, и снова происходит попытка сопоставления и т. д., пока больше не останется символов. В зависимости от используемого квантификатора последняя попытка будет сопоставлять с 1 или 0 символов.

**Ленивые квантификаторы** начинают с начала строки и съедают по одному символу в попытке найти соответствие. И лишь в самую последнюю очередь они сравнивают со всей строкой.

**Собственнические квантификаторы** всегда съедают всю входную строку и всегда производят только одну попытку сопоставления. В отличие от жадных квантификаторов собственнические квантификаторы никогда на сдвигаются назад, даже если это привело бы к успешному сопоставлению.

**Захват групп —** это способ расценивать несколько символов как единое целое. Порция входной строки, которая совпадает с захватываемой группой, будет сохранена в памяти для последующий обращений.

**Захватываемые группы нумеруются подсчётом скобок слева направо. В выражении ((A)(B(C))) будут такие группы:**

1. ((A)(B(C)))
2. (A)
3. (B(C))
4. (C)

**groupCount -** возвращает int, показывающий количество захватываемых групп в шаблоне.Существует также специальная группа 0, которая всегда представляет из себя полное выражение. Эта группа не включается в число, сообщаемое groupCount. Обратиться к такой группе внутри регулярного выражения можно с помощью обратной косой черты с последующей цифрой, обозначающей номер группы.

**Границы совпадений**

**^ -** В начале строки

**$ -** В конце строки

**\b -** На границе слова

**\B -** Не на границе слова

**\A -** В начале входной строки.

**\G -** Конец предыдущего совпадения

**\Z -** Конец ввода для завершающего символа, если есть.

**\z -** Конец ввода.

**Pattern** – это класс который представляет собой скомпилированное представление регулярного выражения. У него нет публичных конструкторов, поэтому для создания объекта необходимо вызвать статический метод compile и передать в качестве аргумента строку с регулярным выражением. Также можно передать в качестве второго аргумента флаг в виде статической константы.

**Флаги:**

**Pattern.CANON\_EQ**- два символа будут считаться совпадающими тогда и только тогда, когда их полные канонические разложения совпадают. Указание этого флага может привести к снижению производительности.

**Pattern.CASE\_INSENSITIVE**– Включает сопоставление без учета регистра. Сопоставление без учета регистра также может быть включено с помощью встроенного выражения флага (?i). Указание этого флага может привести к небольшому снижению производительности.

**Pattern.COMMENTS -** Разрешает пробелы и комментарии в шаблоне. В этом режиме пробелы игнорируются, а встроенные комментарии, начинающиеся с #, игнорируются до конца строки. Режим комментариев также может быть включен с помощью встроенного выражения флага (?x).

**Pattern.DOTALL -** Включает режим dotall. В режиме dotall выражение . соответствует любому символу, включая символ окончания строки. По умолчанию это выражение не соответствует разделителям строк. Режим Dotall также может быть включен с помощью встроенного выражения флага (?s).

**Pattern.LITERAL -** Позволяет выполнять буквальный синтаксический анализ шаблона. Если указан этот флаг, то входная строка, определяющая шаблон, обрабатывается как последовательность буквенных символов.

**Pattern.MULTILINE -** Включает многострочный режим. В многострочном режиме выражения ^ и $ совпадают сразу после или непосредственно перед, соответственно, завершителем строки или концом входной последовательности.

**Pattern.UNICODE\_CASE -** Позволяет сгибать регистры в соответствии с Юникодом. По умолчанию сопоставление без учета регистра предполагает, что сопоставляются только символы в кодировке US-ASCII. Сгибание регистров с поддержкой Unicode также может быть включено с помощью встроенного выражения флага (?u). Указание этого флага может привести к снижению производительности.

**Pattern.UNIX\_LINES** - Включает режим строк в UNIX. В этом режиме только '\n' ограничитель строки распознается в поведении ., ^ и $. Режим строк в UNIX также может быть включен с помощью встроенного выражения флага (?d).

**Методы:**

**matches** - статический метод который позволяет быстро сравнить регулярное выражение со строкой.

**Matcher –** это класс который осуществляет сопоставление строки символов с шаблоном в Pattern. У него нет публичных конструкторов, для создания нужно использовать метод matcher  класса Pattern.

**Методы:**

**region(int start, int end)** – позволяет задать регион строки с которой должен работать класс. Изначально он работает со всей строкой.

**start() -** возвращает индекс начала предыдущего совпадения.

**start(int group)** - возвращает начальный индекс схваченной группы group в предыдущем совпадении.

**end()** - возвращает индекс после последнего совпавшего символа.

**end(int group)** - возвращает индекс после последнего символа схваченной группы group в предыдущем совпадении.

**lookingAt() -** пытается сопоставить строку с самого начала региона с шаблоном. Возвращает true, если начало региона входной строки совпадает с шаблоном.

**find()** - пытается найти следующую последовательность во входном регионе, совпадающую с шаблоном.

**find(int start) -** очищает состояние Matcher и пытается найти следующее совпадение с индекса start.

**matches() -** пытается сопоставить весь регион с шаблоном.

**replaceAll(String replacement) -** заменяет все совпадения с шаблонов во входной строке на replacement.

**replaceFirst(String replacement)** - заменяет первое совпадение с шаблоном во входной строке на replacement.

**PatternSyntaxException** — непроверяемое исключение, которое возникает при наличии синтаксической ошибки в регулярном выражении.

**Методы:**

**getDescription()** - Возвращает описание ошибки.

**getIndex() -** возвращает приблизительный индекс, где находится ошибка, либо -1.

**getPattern() -** возвращает регулярное выражение.

**getMessage() -** возвращает многострочное сообщение об ошибке с описанием и индексом.