Лекция 8

Работа со строками в Java

1. Работа со строками (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. 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.

Таблица 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.

1.2. 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. 2.

Таблица 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.

1.3. Сравнение строк

Поскольку в Java строки являются объектами, для сравнения строк можно использовать оператор "==" и метод

`public boolean equals(Object объект),`

сравнивающей 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. 3.

Таблица 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.

1.4. Поиск в строках

Для поиска символов или последовательностей символов (только в строках класса **String**) используются следующие перегружаемые методы **`indexOf()`**, приведенные в табл. 4.

Таблица 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.

1.5. 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. 5.

Таблица 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, char[] массив, int индекс-вставки)	Возвращаемого значения нет. Копирует часть строки, начиная с символа в позиции начальный-индекс и заканчивая символом в позиции индекс-вставки + (конечный-индекс - начальный-индекс)-1 в символьный массив, начиная с позиции индекс-вставки.

1.6. Модификация строк

Создание новых строк на основе существующих строк выполняется с помощью методов класса String, приведенных в табл. 6.

Таблица 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. 7, позволяют непосредственно модифицировать строку.

Таблица 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.

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.

```
/* 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.**