РАБОТА СО СТРОКОВЫМИ ДАННЫМИ

Обработка строк

- Как в большинстве других языков программирования, *строка* в Java — это последовательность символов.
- Но, в отличие от многих языков, которые реализуют строки как *символьные массивы,* в Java строки реализуются как *объекты* типа string.
- Реализация строк в виде встроенных объектов обеспечивает полный комплект свойств, которые делают обработку строк очень удобной.
- После создания string-объекта символы, входящие в строку, нельзя изменять.
- Над этим объектом можно выполнять все типы строковых операций.
- Для того чтобы изменить версию существующей строки, нужно создать новый объект типа string, который содержит необходимую модификацию.

Обработка строк

- Для тех случаев, когда желательна изменяемая строка, существует компаньон класса string с именем stringBuffer, чьи объекты содержат строки, которые могут изменяться после их создания.
- Классы string и stringBuffer определены в пакете java.iang. Таким образом, они доступны всем программам автоматически.
- Оба объявлены как final, что означает, что ни один из этих классов не может иметь подклассов.
- Когда говорят, что строки в объектах типа string являются неизменяемыми, это означает, что содержимое string-объекта не может быть модифицировано после того, как он был создан. Однако переменную, объявленную как string-ссылка, можно в любое время изменить так, чтобы она указывала на другой string-объект.

String-конструкторы

- Класс string поддерживает несколько конструкторов. Чтобы создать пустой объект типа string, нужно вызвать умалчиваемый конструктор. Например, следующий оператор

String s = new String();

создает экземпляр класса string, не содержащий символов (т. е. пустую строку).

- Чтобы создать string-объект, инициализированный массивом символов, используйте следующий конструктор:

String(char *chars*[ ])

- Например:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

Этот конструктор инициализирует (объектную) переменную s строкой "abc".

String-конструкторы

- В качестве инициализатора можно указать *поддиапазон* символьного массива, для чего используется следующий конструктор:

String (char *chars[ ]* , int *startIndex,* int *numChars)*

- где *startindex* определяет индекс (*индекс* — это просто порядковый номер символа в строке, причем нумерация выполняется как в массиве — с нуля), с которого начинается поддиапазон; **numChars** определяет число символов в диапазоне. Например:

char chars[] = { 'a', 'b', 'c', 'd' 'e‘,'f' }; String s = new String (chars, *2,* 3);

что инициализирует строчный объект s символами cde.

- С помощью конструктора

String(String strObj)

- можно создать string-объект, который содержит такую же символьную последовательность, как другой string-объект. Здесь *strObj* — объект типа string.

String-конструкторы

- Тип char в Java использует 16-разрядное представление символов (из набора Unicode), тогда как в Internet для представления строчных символов используется 8-разрядный набор символов ASCII. Поскольку 8-разрядные строки ASCII используются достаточно широко, в классе string имеются конструкторы, которые инициализируют строку 8-разрядными byte-массивами. Формы этих конструкторов таковы:

String(byte *asciiChars*[ ])

String(byte *asciiChars*[ ], int *startIndex*, int *numChars*)

- где *asciiChars* указывает байтовый массив. Вторая форма позволяет указать поддиапазон. В каждом из этих конструкторов преобразование байтов в символы использует кодовый набор символов платформы, заданный по умолчанию.

Длина строки

- Длина строки определяется количеством содержащихся в ней символов. Для получения этого значения вызовите метод length() в форме:

int length()

- Следующий фрагмент выведет число 3, т. к. в строке s имеется три символа:

```
char chars[] ={ 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Специальные строковые операции

- Поскольку работа со строками — обычная и очень важная часть программирования, в синтаксис языка Java добавлена поддержка для некоторых специальных строковых операций.
- К этим операциям относятся автоматическое создание новых string-объектов из строковых литералов, конкатенация множественных string-объектов при помощи операции + и преобразование других типов данных в строковое представление.
- Существуют явные методы для реализации всех этих функций, но Java выполняет их автоматически как для удобства программиста, так и для того, чтобы сделать запись программы более ясной.

Строковые литералы

- Для каждого строкового литерала в программе Java автоматически создает string-объект. Например, следующий кодовый фрагмент создает две эквивалентные строки:

```
char chars[] = { 'a', 'b‘,'c' };
String s1 = new String(chars);
String s2 = "abc";      // использование строкового литерала
```

- Поскольку объект типа string создается для каждого строкового литерала, то этот литерал можно применять в любом месте, где указывается string-объект.

```
System.out.println("abc".length());
```

- Здесь строчный литерал ("abc") указан на месте, где должна бы была стоять объектная ссылка s2 из предыдущего фрагмента. Аргумент "abc".length o вызывает метод length() прямо для строки "abc" (вместо того, чтобы вызвать его для объекта s2).

Конкатенация строк

- Вообще, Java не разрешает применять операции к string-объектам. Однако в этом правиле есть одно исключение. Это операция +, которая связывает две строки, строя в результате string-объект с объединенной последовательностью символов. Можно также организовать цепочку из нескольких + операций. Например, следующий фрагмент связывает три строки:

```
String age = "9";
String s = "Ему " + age + " лет.";
System.out.println(s);
```

- Здесь происходит конкатенация (сцепление) трех строк, в результате которой на экран выводится строка "Ему 9 лет.".
- Еще одно практическое использование конкатенации — это создание очень длинных строк. Вместо ввода длинных последовательностей символов в исходный код можно разбить их на меньшие части и использовать цепочку + операций для их сцепления. Например:

```
// Использование конкатенации для создания длинных строк.
class ConCat {
public static void main(String args[]) {
String longStr = "Это была бы очень длинная строка, " + "не удобная ни для ввода, ни для
    вывода. " + "Но конкатенация строк " + "устраняет этот недостаток.";
System.out.println(longStr); ) }
```

Конкатенация других типов данных

- Операцию конкатенации строк можно использовать с другими типами данных. Например, рассмотрим следующую, немного измененную, версию предыдущего примера:

```
int age= 9;
String s = "Ему " + age + " лет.";
System.out.println(s);
```

- В этом случае переменная age — имеет int-тип, а не string, как в предыдущем фрагменте, но вывод — такой же, как прежде. Это потому, что int-значение переменной age автоматически преобразуется в ее строчное представление внутри string-объекта и затем сцепляется аналогичным способом.
- Однако будьте внимательны, когда смешиваете другие типы операций в выражениях конкатенации строк. Вы можете получить неожиданные результаты. Рассмотрим следующий фрагмент:

```
String s = "четыре: " + 2 + 2; System.out.println(s);
```

- Этот фрагмент выводит на экран:

```
четыре:  22
```

- а не четыре: 4. С учетом старшинства операций сначала выполняется конкатенация первого операнда ("четыре:") со строчным эквивалентом второго ("2"). Этот результат затем сцепляется со строчным эквивалентом третьего операнда (тоже "2"). Чтобы выполнить сначала целочисленное сложение, нужно использовать круглые скобки:

```
String s =  "четыре:   "  +   (2+2); Теперь s содержит строку "четыре:  4".
```

Преобразование строк и метод toString()

- Каждый класс реализует toString(), поскольку данный метод определен в классе Object.
- Однако реализации toString(), заданной по умолчанию, редко достаточно. Метод toString () имеет следующую общую форму:

String toString()

- Реализация toString() просто возвращает объект типа string, который содержит удобочитаемую строку, описывающую объект вашего класса.
- Переопределяя toString() для создаваемых вами классов, вы получаете строчные представления объектов, полностью интегрированные в среду программирования Java.
- Например, они могут использоваться в операторах print() и printin() в выражениях конкатенации.

Преобразование строк и метод toString()

```
// Переопределение toString() для Box-классов.
class Box {
double width;
double height;
double depth;
Box(double w, double h, double d) {
width = w;
height = h;
depth = d; }

public String toString ()
{
return "Размеры Box-объекта: " + width + " x " +
      depth + " x " + height + "."; } }
class toStringDemo {
public static void main(String args[]) ( Box b = new
      Box(10, 12, 14);
String s = "Box b: " + b;
// конкатенация Box-объекта
System.out.println(b);
// преобразование Box-объекта в строку
//System.out.println(s);
}
 }
```

- Вывод этой программы:

Размеры Box-объекта: 10 x 14 x 12.
Box b: Размеры Box-объекта: 10 x 14 x 12.

- Обратите внимание, что метод toString() вызывается автоматически, когда Box-объект используется в выражении конкатенации или в обращении к println ().

Извлечение символов

- Класс String предоставляет несколько способов извлечения символов из объекта типа String.
- Хотя символы, которые составляют строку String-объекта, не могут быть индексированы, как в символьном массиве, многие из String-методов используют индекс (порядковый номер или позицию) символа в строке для выполнения своих операций.
- Подобно массивам, индекс строки начинается с нуля.

Метод *charAt()*

- Для извлечения одиночного символа из String-объекта вы можете прямо сослаться на индивидуальный символ через метод charAt().

char charAt(int *where)*

- где параметр where — индекс (номер) символа, который вы хотите получить. Значение *where* должно определять позицию искомого символа в строке и не может быть отрицательным. charAt() возвращает символ, находящийся в указанной позиции строки. Например, фрагмент:

```
char ch;
ch =  "abc".charAt(1);
```

назначает символьное значение «b" переменной ch.

Метод *getChars()*

- Если нужно извлечь больше одного символа, то можно использовать метод getChars ().

void getChars(int sourceStart, int *sourceEnd,* char *target[ ]*, int targetStart)

Здесь *sourceStart* указывает индекс начала подстроки;

sourceEnd указывает индекс, который на 1 больше индекса конца желательной подстроки.

- Таким образом, подстрока содержит символы в позициях от SourceStart до sourceEnd-1.
- Массив, который будет принимать символы, указывается параметром target[].
- Позиция в *target,* начиная с которой будет скопирована подстрока, передается через параметр targetstart.

```
class getCharsDemo {
public static void main(String args[]) {
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0); System.out.println(buf); } }
```

- Вывод этой программы:

demo

Метод *getBytes()*

- Имеется альтернатива getChars (), которая сохраняет символы в массиве байтов.
- Этот метод называется getBytes (). Он выполняет преобразование символов в байты заданное по умолчанию на используемой платформе.

Byte[ ] getBytes()

- Имеются и другие формы getBytes().
- getBytes() наиболее полезен, когда вы экспортируете string-значение в среду, которая не поддерживает 16-разрядные символы Unicode.
- Например, большинство протоколов Internet и форматов текстовых файлов использует 8-разрядную кодировку ASCII для всего текстового обмена.

Метод *toCharArray()*

- Если вы хотите преобразовать все символы в объекте типа String в символьный массив, самый простой способ состоит в вызове метода toCharArray ().
- Он возвращает массив символов всей строки и имеет следующую общую форму:

char[ ] toCharArray()

- Эта функция обеспечивает определенные удобства, так как достичь того же результата можно и с помощью метода getChars ().

Сравнение строк

- Класс String включает несколько методов, которые сравнивают строки или подстроки внутри строк.

Методы *equals()* и *equalsIgnoreCase()*

- Чтобы сравнивать две строки на равенство, нужно использовать метод equals (). Он имеет следующую общую форму:

boolean equals(Object *str)*

где ***str*** — string-объект, который сравнивается с вызывающим string-объектом. Метод возвращает значение true, если строки содержат одни и те же символы в одинаковом порядке, иначе возвращается false. Сравнение чувствительно к регистру.

- Чтобы выполнить сравнение, которое игнорирует различия в регистре, вызывается метод equalsIgnoreCase(). При сравнении двух строк он предполагает, что символы A—Z и a—z не различаются. Общий формат этого метода:

boolean equalsIgnoreCase(String str)

где ***str*** — string-объект, который сравнивается с вызывающим string-объектом. Он тоже возвращает true, если строки содержат одни и те же символы в одном и том же порядке, иначе возвращает false.

Методы *equals()* и *equalsIgnoreCase()*

```
// Демонстрирует equals O и equalsIgnoreCase ().
class equalsDemo {
public static void main(String args[]) (
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Good-bye";
String s4 = "HELLO";
System.out.println(s1 + " равно " + s2 + " -> " + si.equals(s2));
System.out.println(s1+ " равно " + s3 + " -> " + si.equals(s3));
System.out.println(s1 + " равно " + s4 + " -> " + si.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " + s1.equalsIgnoreCase
    (s4)) ; } }
```

- Вывод этой программы:

```
Hello      равно Hello -> true
Hello      равно equals Good-bye -> false
Hello      равно equals HELLO -> false
Hello      equalsIgnoreCase HELLO -> true
```

Метод *regionMatches()*

- Метод regionMatches () сравнивает некоторую область внутри строчного объекта с другой некоторой областью в другом строчном объекте.

boolean regionMatches(int *startIndex,* String *str2,* int *str2StartIndex,* int numChars)

boolean regionMatches (boolean *ignoreCase,* int *startIndex,* String *str2,* int *str2StartIndex,* int *numChars)*

- Для обеих версий startIndex определяет индекс, с которого область начинается в вызывающем string-объекте.
- Сравниваемый string-объект указывается параметром *str2.*
- Индекс, в котором сравнение начнется внутри *str2,* определяется параметром *str2startIndex.*
- Длина сравниваемой подстроки пересылается через *numChars.* Во второй версии, если *gnoreCase* — true, регистр символов игнорируется. Иначе, регистр учитывается.

Методы *startsWith()* и *endsWith()*

- В классе string определены две подпрограммы, которые являются специализированными формами метода regionMatches ().
- Метод startsWith() определяет, начинается ли данный String-объект с указанной строки. Наоборот, метод endsWith() определяет, заканчивается ли string-объект указанной строкой. Они имеют следующие общие формы:

boolean startsWith(String *str*)

boolean endsWith(String *str*)

где ***str*** — проверяемый string-объект. Если строки согласованы, возвращается true, иначе — false.

- Например,

"Foobar".endsWith("bar")

- И

"Foobar".startsWith("Foo")

- оба возвращают true.

Методы *startsWith()* и *endsWith()*

- Вторая форма startsWith() с помощью своего второго параметра *(startindex)* позволяет определить начальную точку области сравнения (в вызывающем объекте):

boolean startsWith(String str, int *startindex)*

где *startindex* определяет индекс символа в вызывающей строке, с которого начинается поиск символов для операции сравнения.

- Например:

"Foobar".startsWith("bar", 3)

- возвращает true (потому что строка первого аргумента вызова точно совпадает с подстрокой "Foobar", начинающейся с четвертой[1] позиции в исходной строке).

Сравнение *equals()* и операции ==

- Важно понять, что метод equals () и оператор == выполняют две различных операции.
- Метод equals о сравнивает символы внутри string-объекта, а оператор == — две объектные ссылки, чтобы видеть, обращаются ли они к одному и тому же экземпляру (объекту).

```
// equals() в сравнении с ==
class EqualsNotEqualTo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = new String(sl);
System.out.println(s1 + " равен " + s2 + " -> " +s1.equals(s2));
System.out.println(si1+ " == " + s2 + " -> " + (s1 == s2)); } }
```

- Переменная s1 ссылается на string-экземпляр, созданный строкой "Hello". Объект, на который указывает s2, создается с объектом s1 в качестве инианализатора. Таким образом, содержимое двух string-объектов идентично, но это — разные объекты. Это означает, что s1 и s2 не ссылаются на один и тот же объект и. Поэтому, при сравнении с помощью операции == оказываются не равными, как показывает вывод предыдущего примера:

```
Hello равен Hello -> true
Hello == Hello -> false
```

Метод *compareTo()*

- Часто, не достаточно просто знать, идентичны ли две строки. Для приложений сортировки нужно знать, какая из них меньше, равна, или больше чем другая.
- Одна строка считается меньше чем другая, если она расположена *перед* другой в словарном (упорядоченном по алфавиту) списке.
- Строка считается больше чем другая, если она расположена *после* другой в словарном списке.

int compareTo (String *str)*

- Здесь *str* — string-объект, сравниваемый с вызывающим string-объектом. Результат сравнения возвращается (в вызывающую программу) и интерпретируется так:
  - Меньше нуля: строка вызова — меньше, чем *str.*
  - Больше нуля: строка вызова — больше, чем *str.*
  - Нуль: две строки равны.

Метод *compareTo()*

```
// Пузырьковая сортировка строк,
class SortString {
static String arr[] = {
"Now", "is", "the", "time", "for", "all",
    "good", "men", "to", "come", "to",
    "the", "aid", "of", "their", "country" I ;
public static void main(String args[])
{ for(int j =0; j < arr.length; j++)
{
for(int i = j +1; i < arr.length; i++)
 { if(arr[i].compareTo(arr[j]) < 0)
{ String t = arr[j];
arrfj] = arr[i];
arr[i] = t; } }
System.out.println(arr[j]); } } }
```

- Вывод этой программы :

```
Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to
```

- Как вы видите, compareTo() принимает во внимание символы нижнего и верхнего регистра.
- Если вы хотите игнорировать различия в регистре при сравнении двух строк, используйте метод

int compareToignoreCase(String *str*)

- Этот метод был добавлен в Java 2.

Поиск строк

- Класс string предоставляет два метода, которые позволяют выполнять поиск указанного символа или подстроки внутри строки:
  - Indexof (). Поиск первого вхождения символа или подстроки.
  - lastIndexOf(). Поиск последнего вхождения символа или подстроки.
- При неудачном поиске возвращается — 1. Для поиска *первого* вхождения символа используйте

int indexOf(int ch)

- Для поиска *последнего* вхождения символа используйте

int lastIndexOf(int *ch)*

Здесь *ch* — разыскиваемый символ.

- Для поиска первого или последнего вхождения подстроки используйте

int indexOf(String *str)* int lastIndexOf(String *str}*

Здесь *str* определяет подстроку.

- Можно определить начальную точку поиска, применяя следующие формы:

int indexOf(int *ch,* int *startIndex)*

int lastIndexOf(int *ch,* int *startIndex)*

int indexOf(String *str,* int *startIndex)*

int lastIndexOf (String str, int *startIndex)*

Здесь *startIndex* указывает индекс (номер) символа, с которого начинается поиск для indexOf () поиск выполняется от символа с индексом **startIndex** до конца строки. Для lastIndexOf о поиск выполняется от символа с индексом startIndexOf() до нуля.

Поиск строк

```
// Демонстрирует indexOf{) к lastIndexOf().
class indexOfDemo {
public static void main(String args[]) {
String s = "Now is the time for all good men " +
"to come to the aid of their country.";
System.out.println(s);
System.out.println("indexOf(t) = " +
      s.indexOf('t'));
System.out.println("lastIndexOf(t) =        "        +
      s.lastIndexOf('t'));
System.out.println("indexOf(the) = "        +
      s.indexOf("the"));
System.out.println("lastIndexOf(the)        =        " +
      s.lastIndexOf("the"));
System.out.println("indexOf(t, 10) =        "        +
      s.indexOf('t', 10));
System.out.println("lastIndexOf(t, 60)      = " +
      s.lastIndexOf('t', 60));
System.out.println("indexOf(the, 10)        =        " +
      s.indexOf("the", 10));
System.out.println("lastIndexOf(the, 60) = " +
s.lastIndexOf("the", 60)); } }
```

- Вывод этой программы:

```
Now is the time for all good men to come to the aid
      of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) =11
lastIndexOf(t, 60) =55
indexOf(the, 10) =44
lastIndexOf (the, 60) =55
```

Изменение строки

- Поскольку String-объекты неизменяемы, всякий раз, когда вы хотите изменить String-объект, нужно или копировать его в stringBuffer, или использовать один из следующих String-методов, которые создадут новую копию строки с вашими модификациями.

Метод *substring()*

- Вы можете извлечь подстроку с помощью метода substring (). Он имеет две формы. Первая:

String substring(int *startIndex)*

- Здесь *startIndex* специфицирует индекс символа, с которого начнется подстрока. Эта форма возвращает копию подстроки, которая начинается с номера ***startIndex*** и простирается до конца строки вызова.
- Вторая форма substring!) позволяет указывать как начальный, так и конечный индексы подстроки:

String substring(int *startIndex,* int *endIndex)*

- Здесь ***startIndex*** указывает начальный индекс; ***endindex*** определяет индекс последнего символа подстроки. Возвращаемая строка содержит все символы от начального до конечного индекса (но не включая символ с конечным индексом).

Метод *substring()*

```
// Замена подстроки,
class StringReplace {
public static void main(String args[]) {
String org = "This is a test. This is, too.";
String search = "is";
String sub = "was";
String result = "";
int i ;
do {
// заменить все совпавшие подстроки
System.out.println(org) ;
i = org.indexOf(search) ;
if(i != -1) {
result = org.substring(0, i);
result = result + sub;
result = result + org.substring(i + search.length());
org = result; } }
while(i != -1);
} }
```

- Вывод этой программы:

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```

Метод *concat()*

- Можно сцеплять две строки, используя метод concat о , с такой сигнатурой:

String concat(String str)

- Данный метод создает новый объект, включающий строку вызова с содержимым объекта *str,* добавленным в конец этой строки, concat () выполняет ту же функцию, что и операция конкатенации +.
- Например, фрагмент

```
String s1 = "one";
String s2 = s1.concat("two");
```

- Помещает строку "onetwo" в s2. Он генерирует тот же результат, что следующая последовательность:

```
String s1  =  "one";
String  s2  =  s1   +   "two";
```

Метод *replace()*

- Метод replace () заменяет все вхождения одного символа в строке вызова другим символом.

String replace(char *original,* char replacement)

- Здесь *original* определяет символ, который будет заменен символом, указанным в *replacement.* Строка, полученная в результате замены, возвращается в вызывающую программу. Например,

String s = "Hello".replace(‘l', 'w');

- помешает в s строку "Hewwo".

Метод *trim()*

- Метод trim() возвращает копию строки вызова, из которой удалены любые ведущие и завершающие пробелы.

String trim()

- Пример:

String s = " Hello World ".trim();

- Этот оператор помещает в строку s "Hello World".
- Метод trim о весьма полезен, когда вы обрабатываете команды пользователя.
- Например, следующая программа запрашивает у пользователя название штата и затем отображает столицу этого штата. Она использует trim о для удаления любых ведущих и завершающих пробелов, которые, возможно, по неосторожности были введены пользователем.

Метод *trim()*

```
// Использование trim() для обработки
    команд,
import java.io.*;
class UseTrim (
public static void main(String args [ ])
throws IOException {
// создать BufferedReader, использующий
    System.in
BufferedReader br = new
BufferedReader(new
    InputStreamReader(System.in));
String str;
System.out.println("Enter 'stop1 to quit.");
 System.out.println("Enter State: ");
do {
str = br.readLine();
str = str.trimO;       // удалить пробелы
if(str.equals("Illinois"))
System.out.println("Capital is Springfield.");
else if(str.equals("Missouri"))
System.out.println("Capital is Jefferson
    City.");
else if(str.equals("California"))
System.out.println("Capital is Sacramento.");
else if(str.equals("Washington"))
System.out.println("Capital is Olympia.");
// ... }
while{!str.equals("stop"));
} }
```

Преобразование данных, использующее метод valueOf()

- Метод vaiueOf () преобразует данные из их внутреннего формата в удобную для чтения форму.
- Это статический метод, который перегружен в классе string для всех встроенных типов Java так, что каждый тип можно преобразовать в строку,
- **static String valueOf(double *num)***
- **static String valueOf (long *num)***
- **static String valueOf (Object *ob)***
- **static String valueOf(char *chars[ ]*)**
- vaiueOf o вызывается, когда необходимо строчное представление некоторого другого типа данных — например, во время операций конкатенации.
- Вы можете вызывать этот метод прямо, с любым типом данных в аргументе, и получать разумное строчное представление.
- Все простые типы преобразуются к их обычному string-представлению.
- Любой объект, который вы передаете в метод valueOf (), возвращает результат обращения к методу toString().
- Для большинства массивов valueOf o возвращает строку, которая указывает, что это — массив некоторого типа. Для массивов типа char, однако, создается string-объект, который содержит символы этого массива. Существует специальная версия vaiueOf (), которая позволяет указывать подмножество char-массива. Она имеет следующую общую форму:

static String valueOf(char chars[ ], int *staxrtIndex,* int *numChars)*

- Здесь **chars** — массив, который содержит символы; ***startindex*** — индекс в массиве символов, в котором начинается желательная подстрока; ***numChars*** указывает длину подстроки.

Изменение регистра символов в строке

- Метод toLowerCase () преобразует вер символы в строке с верхнего регистра на нижний. Метод toUpperCase() преобразует все символы в строке с нижнего регистра на верхний. Неалфавитные символы, типа цифр, остаются незатронутыми.

String toLowerCase()

String toUpperCase()

- Оба метода возвращают объект типа string, который содержит верхне- или нижнерегистровый эквивалент вызывающего String-объекта.

```
// Демонстрирует toUpperCaseO и toLowerCase().
class ChangeCase {
public static void main(String args [ ]) {
String s = "Это тест.";
System.out.println("Оригинал: " + s);
String upper = s.toUpperCase(); String lower = s.toLowerCase();
System.out.println("Uppercase: " + upper);
System.out.println("Lowercase: " + lower); }
```

- Вывод, выполненный этой программой:

```
Оригинал: Это тест.
Uppercase: ЭТО ТЕСТ.
Lowercase: это тест.
```

Класс *StringBuffer*

- StringBuffer — это класс, равный по положению классу String.
- Он обеспечивает много функциональных возможностей для строк.
- String представляет неизменяемые символьные последовательности фиксированной длины.
- StringBuffer представляет возрастающие и перезаписываемые символьные последовательности. StringBuffer может вставлять символы и подстроки в середину строки или добавлять их в конец строки.
- StringBuffer растет автоматически, чтобы создать место для таких добавлений, и часто имеет больше предварительно выделенной памяти, чем фактически необходимо для роста.

Конструкторы *StringBuffer*

StringBuffer()
StringBuffer(int *size)*
StringBuffer(String *str)*

- Заданный по умолчанию конструктор (без параметров) резервирует участок памяти для шестнадцати дополнительных символов, не участвующих в распределении.
- Вторая версия принимает целочисленный аргумент, который явно устанавливает размер буфера.
- Третья версия принимает string-аргумент, который устанавливает начальное содержимое объекта типа StringBuffer и резервирует участок памяти для еще шестнадцати дополнительных символов.

Методы *length()* и *capacity()*

- Текущую длину объекта типа StringBuffer можно найти с помощью метода length(), а общий распределенный объем — с помощью метода capacity(). Они имеют следующие общие формы:

int length()

int capacity()

- Пример:

```
// StringBuffer length() в сравнении с capacity() .
class StringBufferDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out-println{"buffer = " + sb);
System.out.printing"length = " + sb.length O);
System.out.println("capacity = " + sb.capacity 0); } }
```

- Вывод этой программы, который показывает, как StringBuffer резервирует дополнительное пространство для возможных манипуляций:

```
buffer = Hello
length = 5
capacity = 21
```

- Объект sb инициализируется

Метод *ensureCapacity()*

- Если вы хотите предварительно выделить участок памяти для некоторого числа символов после того, как StringBuffer был создан, то для установки размера буфера можно использовать метод ensureCapacity(). Это полезно, если вы знаете заранее, что будете добавлять (в конец буфера StringBuffer) большое количество маленьких строк, ensurecapacity о имеет следующую общую форму:

void ensureCapacity(int *capacity)*

где capacity определяет общий размер (емкость) буфера.

Метод *setLength()*

- Чтобы устанавливать длину буфера в пределах объекта типа StringBuffer, используйте метод setLength (). Его общая форма:

void setLength(int *len)*

где ***len*** определяет длину буфера. Это значение должно быть неотрицательным.

- Когда вы увеличиваете размер буфера, к концу существующего буфера добавляются нулевые символы. Если вы вызываете setLength() со значением меньше чем текущее значение, возвращенное методом length (), то символы, хранящиеся вне новой длины, будут потеряны.

Методы *charAt()* и *setCharAt()*

- Значение одиночного символа можно получить из StringBuffer с помощью метода charAt(). Устанавливать значение символа в StringBuffer может метод setcharAt (). Их общие форматы:

char charAt(int *where)*

void setcharAt(int *where,* char *ch)*

- **Параметр *where*** в charAto указывает индекс получаемого символа. В setcharAt о ***where*** указывает индекс устанавливаемого символа, а ch — новое значение этого символа. Для обоих методов параметр ***where*** должен быть неотрицательным и не должен определять позицию вне конца буфера.

```
// Демонстрирует charAt() и setCharAt().
class setCharAtDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer before = " + sb;
System.out.println("charAt(1) before = " + sb.charAt(i)); sb.setCharAt(1, 'i');
sb.setLength(2);
System.out.println("buffer after = " + sb);
System.out.println("charAt(1) after = " + sb.charAt(1)); } }
```

- Вывод, сгенерированный этой программой:

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi charAt(1) after = i
```

Метод *getChars()*

- Для копирования подстроки StringBuffer в массив можно использовать метод getChars (). Его общая форма:

void getchars (int *sourceStart,* int *sourceEnd,* char *target* [ ], int *targetStart)*

где ***sovircestart*** указывает индекс начала подстроки; **sourceEnd** определяет индекс, на 1 больший, чем индекс конца подстроки. Это означает, что подстрока содержит символы от ***sourceStart*** до ***sourceEnd*-1.** массив символов указывается параметром ***target.*** Индекс (номер) позиции в ***target,*** с которой будет скопирована подстрока, передается в *targetStart.* Необходимо позаботиться о том, чтобы массив *target* был достаточно большим для размещения всех символов указанной подстроки.

Метод *append()*

- Метод append() добавляет строчные представления любого другого типа данных в конец вызывающего объекта типа stringBuffer. Он имеет перегруженные версии для всех встроенных типов и для типа object. Имеется несколько его форм:

StringBuffer append(String *str)* StringBuffer append(int num) StringBuffer append(Object *obj)*

- Чтобы получить строчное представление каждого параметра, вызывается метод string, vaiueOf() . Результат добавляется в конец текущего stringBuffer-объекта. Сам буфер возвращается каждой версией append о. Это позволяет соединять в цепочку все последовательные вызовы append о, как показано в следующем примере:

```
// Демонстрирует append(). class appendDemo {
public static void main(String args [ ]) (
String s;
int a = 42;
StringBuffer sb = new StringBuffer(40);
s = sb.append("a = ").append(a).append("!").toString (); System.out.println(s); j }
```

- Вывод этого примера:

```
a  =  42!
```

Метод *insert()*

- Метод insert о вставляет одну строку в другую.

StringBuffer insert(int *index.* String *str)

StringBuffer insert(int index, char *ch)

StringBuffer insert(int *index,* Object *obj)

- Параметр *index* указывает индекс (номер позиции) (в вызывающем stringBuffer-объекте), с которого будет вставлена строка, символ или объект, указанные во втором параметре.
- Следующая программа вставляет "нравится" между "Мне" и "Java":

```
// Демонстрирует insert().
class insertDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Мне Java!");
sb.insert(2, "нравится ");
System.out.println(sb);
}
```

- Вывод этого примера:

```
Мне нравится  Java!
```

Метод *reverse()*

- **Можно** изменить порядок символов в объекте типа stringBuffer, используя метод reverse () с форматом:

StringBuffer reverse ()

- Данный метод возвращает реверсированный (с обратным расположением символов) объект вызова. Следующая программа демонстрирует reverse ():

```
// Использование reverse() для реверса
StringBuffer-объекта. class ReverseDemo {
public static void main(String args[]) {
StringBuffer s = new StringBuffer("abcdef");
System.out.println(s);
s.reverse();
System.out.println (s); }
}
```

- Вывод этой программы:

```
abcdef
fedcba
```

Методы *delete()* и *deleteCharAt()*

StringBuffer delete(int *startIndex,* int *endIndex)

StringBuffer deleteCharAt(int *loc)*

- Метод delete () удаляет последовательность символов из объекта вызова. Параметр ***startindex*** указывает индекс первого удаляемого символа; ***endIndex*** определяет индекс, на 1 больший, чем у последнего удаляемого. Таким образом, удаляемая подстрока простирается от ***startindex*** до ***endIndex—*** 1. Возвращается stringBuffer-объект, полученный в результате удаления.
- Метод deieteCharAt() удаляет символ с индексом, указанным в *loc,* и возвращает результирующий StringBuffer-объект.
- Программа, которая демонстрирует методы delete о и deieteCharAt о:

```
// Демонстрирует delete О и deieteCharAt()
class deleteDemo {
public static void main(String args[]) (
StringBuffer sb = new StringBuffer("Это проверка.");
System.out.println("До delete: " + sb) ;
sb.delete (4, 7);
System.out.println("После delete: " + sb) ;
sb.deieteCharAt(0);
System. out.println("После deieteCharAt: " + sb); } }
```

- Вывод этой программы:

```
До delete: Это проверка.
После delete: Это верка.
После deleteCharAt: то верка.
```

Метод *replace()*

StringBuffer replace(int *startIndex,* int *endIndex,* String *str)

- Заменяемая подстрока указывается индексами ***startIndex*** и **endIndex.** Таким образом, заменяемая подстрока занимает позиции от ***startIndex*** до ***endIndex*** -1. Заменяющая строка передается через параметр ***str.*** Модифицированный таким образом stringBuffer-объект возвращается в вызывающую программу.

```
// Демонстрирует replace()
class replaceDemo (
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Это есть тест.");
sb.replace(4, 8, "был");
System.out.println("После replace: " + sb) ; } }
```

- Вывод этой программы:

После replace: Это был тест.

Метод *substring()*

String substring(int *startIndex*)

String substring(int *startIndex*, int *endIndex*)

- Первая форма возвращает подстроку, которая начинается в позиции **startIndex** и простирается до конца вызывающего stringBuffer-объекта.
- Вторая форма возвращает подстроку, которая начинаетеся в позиции **startIndex** и простирается до позиции **endIndex**-1.
- Перечисленные методы работают точно так же, как аналогичные методы, определенные в классе string, которые были описаны ранее.

Спасибо за внимание!