Шестая лекция
String

Строки

Класс **String** представляет строковый набор символов. Строки используются практически по всех Java-приложениях, и есть несколько фактов, которые мы должны знать о классе String.

В Java поддерживаются *2 вида строк* :

неизменяемые – объекты класса **String**;

изменяемые – объекты класса **StringBuffer, StringBuilder**

Основные свойства «строковых» классов:
оба класса принадлежат пакету ***java.lang***;
оба класса — **final** (следовательно от их нельзя унаследоваться).

```
public final class String {...}
```

Конструкторы класса String

Строка в Java является объектом, поэтому ее можно создать, как и любой другой объект, при помощи оператора **new**.

Конструктор - это специальный метод, который вызывается при создании нового объекта.

Конструктор без параметров создает пустую строку:

```
String s = new String();
```

Конструкторы, создающие строки из массива символов:

```
char chars[] = {'a', 'b', 'c', 'd', 'e', 'f'};
String s1 = new String(chars); // s1 = "abcdef"
String s2 = new String(chars, 2, 3); // s2 = "cde”
```

Можно также создать массив строк.

```
String[] name = {"Маша", "Петя", "Вася", "Коля"};
```

Строковые константы

Также строку можно создать при помощи литерала (фразы заключенной в кавычки).

Строковый литерал — последовательность символов заключенных в двойные кавычки.

Важно понимать, что всегда когда вы используете строковой литерал компилятор создает объект со значением этого литерала:

Каждая такая константа представляет собой полноценный объект класса String, можно вызывать методы этого объекта.

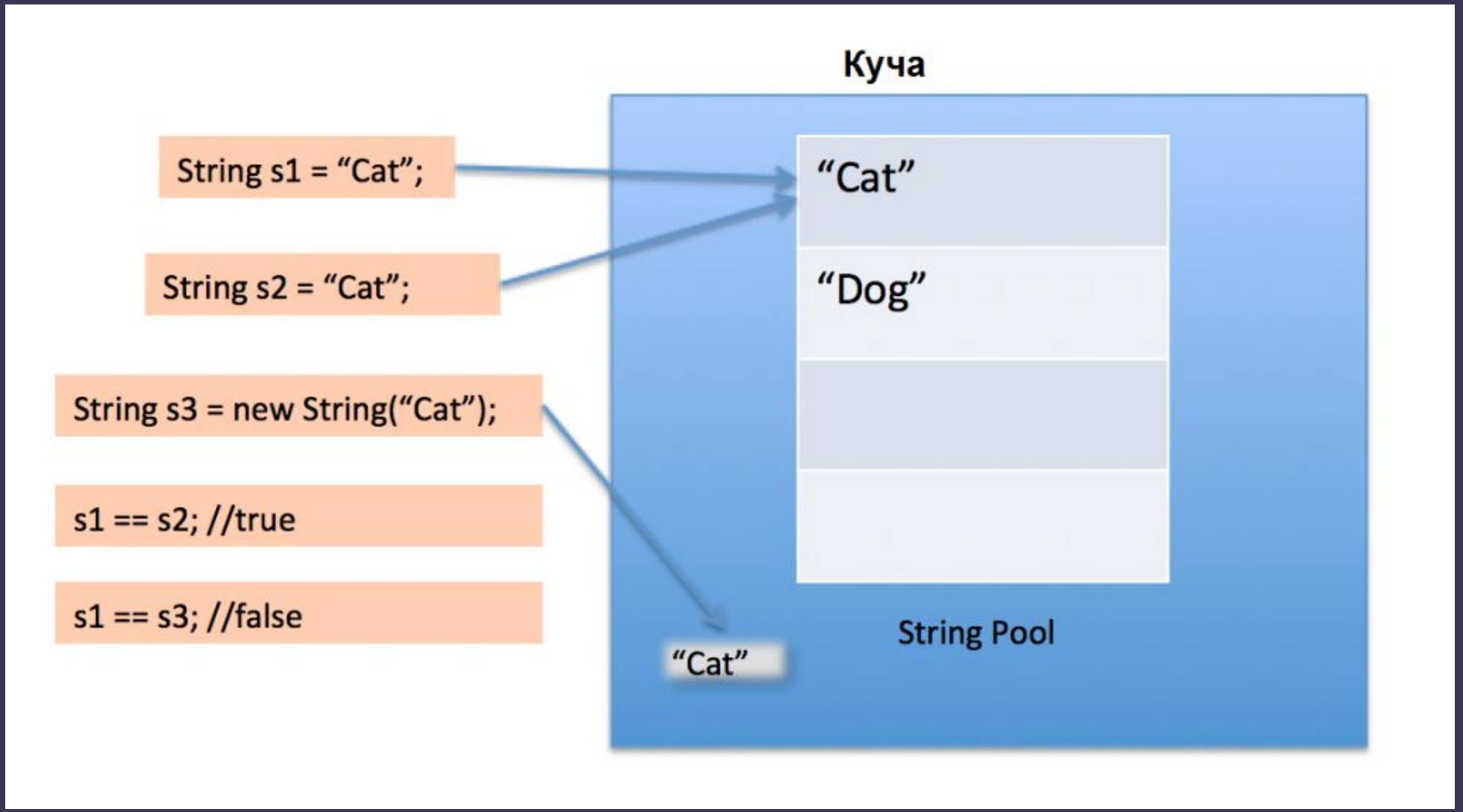
Независимо от способа создания строки являются объектами — экземплярами класса String.

```
String s = "abc";
String s2 = new String("abc");
```

Все ссылки на String объекты хранится в String Pool и перед созданием строки с помощью литерала проверяется нет ли эквивалентной строки в пуле, если нет — добавляется, иначе просто получаем ссылку на уже В случаи с new, новый объект создаться в любом случаи, независимо от пула.

String Pool

Пул строк (**String Pool**) — это множество строк в кучи (Java Heap Memory). Мы знаем, что String — особый класс в java, с помощью которого мы можем создавать строковые объекты.



Ввод строки с клавиатуры

Для чтения данных из консоли используется стандартный поток ввода

System.in

Чтение данных осуществляется посредством вызова метода **nextLine()** ;

```
Scanner sc = new Scanner(System.in);
    String s1;
    String s2;
    s1 = sc.nextLine();
    s2 = sc.nextLine();
System.out.println(s1);
    System.out.println(s2);
```

Форматируем вывод чисел в Java
Создание DecimalFormat

Вы можете использовать класс **DecimalFormat** для контроля вывода нулей в десятичных числах. Пример:

```
String pattern = "###,###.###";
DecimalFormat myFormatter = new DecimalFormat(pattern);
String output = myFormatter.format(123456789.123);
System.out.println(output);
```

```
123456.789  ###,###.###  123,456.789
123456.789  ###.##  123456.79
123.78  000000.000  000123.780
12345.67  $###,###.###  $12,345.67
```

```
String pattern = "###,###.###";
DecimalFormatSymbols otherSymbols = new DecimalFormatSymbols();
otherSymbols.setDecimalSeparator('.');
 otherSymbols.setGroupingSeparator('`');
 DecimalFormat myFormatter = new DecimalFormat(pattern, otherSymbols);
 String output = myFormatter.format(123456789.12);
 System.out.println(output);


 Locale locale  = new Locale("en", "UK");
 DecimalFormat myFormatter = (DecimalFormat)
NumberFormat.getNumberInstance(locale);
 String pattern = "00000000000000.000";
 String pattern = "###,###.###";
 myFormatter.applyPattern(pattern);
 String output = myFormatter.format(123456789.12);
 System.out.println(output);
```

Форматируем вывод даты в Java Создание SimpleDateFormat

Для форматирования дат предназначен класс SimpleDateFormat

Перечислим основные поля (полный список полей можно найти в документации): d - день месяца, M - месяц, y - год, H - часы (24-х часовая шкала), m - минуты, s - секунды.

```
SimpleDateFormat sdf = new SimpleDateFormat("d MMMM yyyy");
System.out.println(sdf.format(new Date()));
```

Класс String предоставляет так же возможность создания форматированных строк. За это отвечает статический метод **format**, например:

```
String formatString = String.format("We are printing double variable %f, string '%s' and integer variable %d.", 2.3, "java", 10);
System.out.println(formatString);
// We are printing double variable (2.300000), string ('java') and integer variable (10).
```

Каждый спецификатор формата, начинающийся с символа %, заменяется соответствующим параметром.

Основные символы преобразования:

%s используется для вставки строки в отформатированную строку

```
String str = String.format("Hello %s", "Raj");
```

%tD используется для вывода даты в формате мм/дд/ гг

```
String str = String.format("Today is %tD", new Date());
Date today = new Date();
System.out.printf("Today is %te %tB, %tY год", today,today,today);
```

%d используется для вывода целых чисел

```
str = String.format("%d", 12.3);
System.out.println(str);
```

%f - float

```
System.out.printf("%8.3f %n", Math.E);
```

Преобразование строк

Когда какое-то значение должно быть преобразовано в строку, вызывается статический метод **String.valueOf(…)**.

Этот метод перегружен ***для всех простых типов***.

```
String pi = String.valueOf(3.14159);
String cond = String.valueOf(true);
```

А также для **Object**. В этом случае вызывается метод **toString()** объекта.

```
String.valueOf(someObj) // Эквивалентно someObj.toString()
```

Также для примитивов есть свой метод valueOf(String s), который возвращает преобразованное численное значение из строки. При этом форматы строки и принимаемого типа должны совпадать. Например:

```
String x = "523.5";
Double xd = Double.valueOf(x);
```

Метод parseInt() в Java используется для получения примитивного типа данных определенной строки. parseXxx()

```
String x = "523.5";
double xd = Double.parseDouble(x);
```

Извлечение символов

char charAt (int index) - возвращает символ строки, стоящий в позиции **index**

```
String s = "Strings are immutable";
char result = s.charAt(8);
System.out.println(result);
```

void getChars(int sourceStart, int sourceEnd, char target[], int targetStart) существующий массив target заполняется частью строки, начиная с позиции **sourceStart** включительно и кончая позицией **sourceEnd** исключительно. Параметр **targetStart** задает место в массиве, с которого начнется его заполнение.

```
String Str1 = new String("Welcome to Sumy");
char[] Str2 = new char[7];
Str1.getChars(2, 9, Str2, 0);
System.out.print("Copied Value ="  + Str2 );
```

Если мы хотим работать со строкой, как с массивом символов, можем конвертировать строку в массив при помощи метода **toCharArray. char[] toCharArray** создается массив символов и заполняется символами строки.

```
char[] Str2 = Str1.toCharArray() ;
```

Извлечение байт

byte[] getBytes() - создается массив байт и заполняется символами строки.

Символы предварительно преобразуются в байты, поэтому количество байт массива будет равно количеству символов строки.

```
String Str1 = new String("Welcome to Sumy");
byte[] Str2 = Str1.getBytes();
```

byte[] getBytes(String encoding) - то же, но с изменением кодировки

```
byte[] Str2 = Str1.getBytes("UTF-8" );
```

Методы класса String

Благодаря множеству методов предоставляется возможность манипулирования строкой и ее символами.

int length() - возвращает длину строки (количество символов в строке).

String trim() - удаляет ведущие и завершающие пробельные символы.

```
String s = "a    ";
System.out.println(s.trim() + "b");//ab
```

String replace(char original, char replacement) - заменяет все вхождения символа **original** символом **replacement**.

```
String sb = "AABCC";
String s = sb.replace("C", "**");
```

String toLowerCase() - изменяет регистр символов в строке, делая все буквы строчными.

String toUpperCase() - изменяет регистр символов в строке, делая все буквы прописными

String replaceAll(String regex, String replacement)) – замещает все вхождения regex на replacement. В качестве regex может быть регулярное выражение

```
String Str = new String("Welcome to Sumy);
System.out.println(Str.replaceAll("Sumy", "Kyiv" ));
```

String replaceFirst(String regex, String replacement) – то же, только замещает первое вхождение

Конкатенация строк

Для склеивания строк в Java используется оператор “+”

```
String s1 = "lang" + "uage" ; // s1 = language
```

Если строка соединяется не со строковым значением простого или ссылочного типа, то последнее преобразуется в строку

```
String s2 = 3 + " frends" ; // s2 = "3 frends”
String s3 = "four = " + 2 + 2;  // s3 = " four = 22 “
String s4 = "four = " + (2 + 2); // s4 = “four = 4”
```

Добавить к одной строке другую можно с помощью оператора “+=“

```
String str = "ABC";
str += "DEF";
```

String **concat**(String s) - присоединяет строку s к строке (Конкатенация). Использование этого метода положительно влияет на производительность и скорость программы.

```
String str2 = "one".concat("two").concat("three");
```

Сравнение строк

boolean **equals**(Object other) – производит посимвольное сравнение строки this со строкой other с учетом регистра символов

boolean equalsIgnoreCase(String other) – то же, но без учета регистра символов

boolean **regionMatches**(int startIndex, String other, int otherStartIndex, int numChars) - сравнивает между собой два участка строк this и other.

```
String Str1 = new String("Welcome to Tutorialspoint.com");
String Str2 = new String("Tutorials");
System.out.println(Str1.regionMatches(11, Str2, 0, 9));
```

boolean **regionMatches** (boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int nimChars) – без учета регистра символов

Сравнение строк

int compareTo(String other) - позволяет узнать, какая строка больше и возвращает отрицательное число, если строка this меньше, чем other, ноль, если строки совпадают, и положительное число, если строка this больше, чем строка other.

```
String str1 = "tutorials", str2 = "point";
int retval = str1.compareTo(str2);
```

boolean startsWith(String substr) - проверяет, начинается ли строка this с подстроки substr.

```
String Str = new String("Welcome to Sumy");
System.out.println(Str.startsWith("Welcome") );
```

boolean endsWith(String substr) - проверяет, заканчивается ли строка this подстрокой substr.

```
String Str = new String("Welcome to Sumy");
System.out.println(Str.endsWith("Sumy") );
```

Поиск в строках

Задача поиска состоит в том, чтобы определить, в каком месте строки в нее входит символ или другая строка. Это позволяет серия методов indexOf():

int indexOf(int ch) - возвращает место первого вхождения в строку this символа ch;

```
String Str = new String("Welcome to Sumy");
System.out.println(Str.indexOf( 'S' ));
```

int lastIndexOf(int ch) - возвращает место последнего вхождения в строку this символа ch;

int indexOf (String substr) - возвращает место первого вхождения в строку this подстроки substr;

int lastIndexOf(String substr) - возвращает место последнего вхождения в строку this подстроки substr;

int indexOf(int ch , fromIndex) — индекс начала поиска

Модификация строк

Как известно, изменить объект класса String нельзя. Можно только создать другую строку, равную части исходной строки

String **substring**(int beginIndex) - создает новую строку, равную части строки **this**, начинающейся с позиции **beginIndex.**

String substring(int beginIndex, int endIndex) – создает новую строку, равную участку строки **this**, начиная с позиции **beginIndex** включительно и заканчивая позицией **endIndex** исключительно

```
String Str = new String("Welcome to Sumy");
System.out.println(Str.substring(10, 15) );
```

Разделения строки

Метод **split(String regex)** для разделения строки на массив строк, используя в качестве разделителя регулярное выражение.

Метод **split(String regex, int numOfStrings)** является перегруженным методом для разделения строки на заданное количество строк.

```
String line = "I am a java developer";
String[] words = line.split(" ");
String[] twoWords = line.split(" ", 2);

System.out.println(Arrays.toString(words));
//String split with delimiter: [I, am, a, java, developer]
System.out.println(Arrays.toString(twoWords));
//String split into two: [I, am a java developer]

String wordsWithNumbers = "I|am|a|java|developer";
String[] numbers = wordsWithNumbers.split("\\|");
System.out.println("String split with special character: "+
Arrays.toString(numbers));//String split with special character: [I, am, a, java,
developer]
```

Изменяемые строки

Изменяемые строки реализованы в Java с помощью класса **StringBuffer**.

У этого класса нет многого, из того, что имеет класс String, зато есть несколько версий методов:

append() - добавления,
delete() - удаления,
insert() - вставки подстроки в строку this.

Все остальные методы для работы с StringBuffer можно посмотреть в документации.

StringBuilder — класс имеет полностью идентичный API с StringBuffer.

Единственное отличие — StringBuilder не синхронизирован. Это означает, что его использование в многопоточных средах есть нежелательным в целом StringBuilder, работает намного быстрее.

Пример работы с изменяемыми строками

Пример. Удалить из строки s1 все вхождения подстроки s2

```
String s1 = "qwe12qwe345qwe678qwe90qwe";

String s2 = "qwe";

StringBuffer s = new StringBuffer(s1);
int p = -1;
while ( (p = s.toString().indexOf(s2)) >= 0 ) {
    s.delete(p, p + s2.length());
}
s1 = s.toString();
```

Регулярные выражения

Для работы с регулярными выражениями в Java представлен пакет java.util.regex. Пакет был добавлен в версии 1.4 и уже тогда содержал мощный и современный прикладной интерфейс для работы с регулярными выражениями.
Все функциональные возможности представлены двумя классами

Pattern

Класс Pattern представляет собой скомпилированное представление РВ. Класс не имеет публичных конструкторов, поэтому для создания объекта данного класса необходимо вызвать статический метод compile и передать в качестве первого аргумента строку с РВ:

```
Pattern pattern = Pattern.compile("^<([a-z]+)([^>]+)*(?:>(.*)<\\/\\1>|\\s+\\/>)$");
```

Matcher и MatchResult

Matcher — класс, который хранит результаты согласования. Не имеет публичных конструкторов, поэтому для создания объекта этого класса нужно использовать метод matcher класса Pattern:

```
Matcher matcher = pattern.matcher(text);
```

Но результатов у нас еще нет. Чтобы их получить нужно воспользоваться методом **find,** который пытается найти подстроку, которая удовлетворяет РВ и вернет true если найдена подстрока соответствует заданному РВ.
метод **group()** - возвращает найденную строку.