

JAVA



ПЛАТФОРМНО-НЕЗАВИСИМИ ПРОГРАМНИ ЕЗИЦИ

Лекция 6. Символни низове

доц. д-р инж. Румен П. Миронов



1. Декларация и преобразуване на низове



Символният низ е последователност от символи, записана на даден адрес в паметта.

Всеки символ в Java има пореден номер в Unicode таблицата (16-битова кодова таблица за символи, т.е. $2^{16} = 65536$ символа).

Класът **java.lang.String** позволява обработката на символните низове в Java. Работата със **String** улеснява манипулацията на текстови данни: построяване на текстове, търсене в текст и много други.

Декларация на символен низ:

```
String greeting = "Hello, Java";
```

Вътрешното представяне на класа е масив от символи. По принцип може да се избегне използването на класа, като се декларира променлива от тип `char[]` и се запълнят елементите на масива символ по символ.

1. Декларация и преобразуване на низове



Недостатъци на това представяне:

1. Запълването на масива става символ по символ, а не наведнъж.
2. Трябва да се знае дължитата на текста, за да се побере в заделеното място за масива.
3. Обработката на текстовото съдържание става ръчно.

Използването на **String** не е идеално и универсално решение – понякога е уместно използването и на други символни структури. Класът `String` има обаче важна особеност – последователностите от символи, записани в променлива от класа, са неизменими (immutable). Веднъж записано, съдържанието на променливата не се променя директно - ако опитаме да променим стойността, тя ще бъде записана на ново място в динамичната памет, а променливата ще започне да сочи към него.

1. Декларация и преобразуване на низове



Типът **String** е по-особен от останалите типове данни. При него се спазват принципите на ООП: стойностите се записват в динамичната памет, а променливите пазят препратка към паметта (референция към обект в динамичната памет). От друга страна, **String** променливите са неизменими – т.е. ако няколко променливи сочат към една и съща област в паметта с дадена стойност, тази стойност не може да бъде директно променена.

Промяната ще се отрази само на променливата, чрез която е редактирана стойността, тъй като това ще създаде нова стойност в динамичната памет и ще насочи въпросната променлива към нея, докато останалите променливи ще сочат на старото място.

1. Декларация и преобразуване на низове



Пример за използване:

```
String msg = "Stand up, stand up, Balkan superman.";
System.out.printf("msg = \"%s\\n\"", msg);
System.out.printf("msg.length() = %d\\n", msg.length());
for (int i = 0; i < msg.length(); i++) {
    System.out.printf("msg[%d] = %c\\n", i, msg.charAt(i));
}
```

Иход от програмата:

```
msg = "Stand up, stand up, Balkan superman."
msg.length() = 36
msg[0] = S
msg[1] = t
msg[2] = a
msg[3] = n
msg[4] = d
.....
```

2. Описание на символни низове



Escaping при символни низове.

Когато е необходимо използване на кавички в съдържанието, тогава трябва да се постави наклонена черта преди тях за указание на компилатора.

Book's title is "Intro to Java"

```
String quote = "Book's title is \"Intro to Java\"";
```

Наклонената черта се използва за символи, които играят специална роля в текста (кавичките) или за дефиниране на действие, което не може да се изрази със символ. Пример за втория случай са обозначаването на символа за нов ред (`\n`), табулация (`\t`), избор на символ по неговия Unicode (`\uXXXX`, където с `X` се обозначава кодът) и др.

3. Създаване и инициализиране на символни низове



Инициализацията може да се извърши по 3 начина:

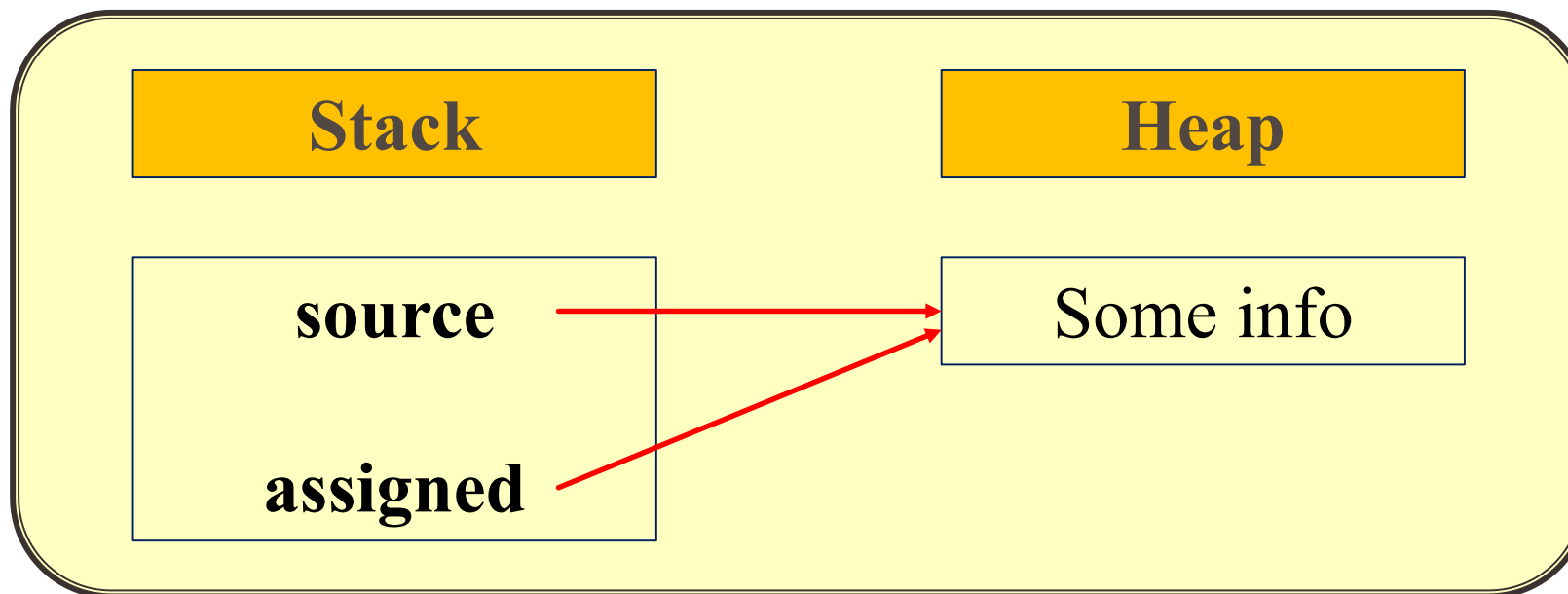
- ❑ *Чрез задаване на символна константа (литерал)* - означава предаване на предефинирано текстово съдържание на променлива от тип String. Използва се когато се знае стойността, която трябва да се съхрани в променливата.

```
String text = "Stand up, stand up, Balkan superman.";
```

- ❑ *Чрез присвояване стойността на друг символен низ* - означава насочване на String променлива към друга променлива от същия тип.

```
String source = "Some info";  
String assigned = source;
```

3. Създаване и инициализиране на символни низове



Промяната на коя да е от променливите ще се отрази само и единствено на нея, поради неизменността на типа **String**.

Това не се отнася за останалите референтни типове, които не са неизменни (*immutable*), защото при тях промените се нанасят на адреса в паметта и всички референции сочат към променения обект.

3. Създаване и инициализиране на символни низове



- *Чрез предаване стойността на операция, връщаща символен низ*
– това може да бъде резултат от метод, който валидира данни;
събиране на стойностите на няколко константи и променливи,
преобразуване на съществуваща променлива и др..

```
String email = "some@email.bg";  
String info = "My mail is: " + email + ".";  
// The result is: some@email.bg.
```

Променливата `info` е създадена от съединяването (concatenation) на литерали и променлива.

4. Операции със символни низове



Четене на символни низове – чрез класа **java.util.Scanner** :

```
Scanner input = new Scanner(System.in);  
String name = input.nextLine();
```

Отпечатване на символни низове:

```
System.out.println("Your name is: " + name);  
System.out.printf("Hello, %s, have a nice reading!", name);
```

Сравняване на низове по азбучен ред. Сравнение за еднаквост:

```
String word1 = "Java";  
String word2 = "JAVA";  
System.out.println(word1.equals(word2)); // false  
System.out.println(word1.equalsIgnoreCase(word2)); // true
```

4. Операции със символни низове



Сравнение на низове по азбучен ред – чрез използване на методите **compareTo(...)** и **compareToIgnoreCase(...)**. Методът **compareTo(...)** връща положително число, отрицателно число или 0 в зависимост от лексикографската подредба.

```
String score = "sCore"; String scary = "scary";  
System.out.println(score.compareToIgnoreCase(scary)); // 14  
System.out.println(scary.compareToIgnoreCase(score)); // -14  
System.out.println(scary.compareTo(score));           // 32
```

В примера се изважда кода на параметъра от кода на променливата, за която е извикан методът. Крайният резултат е 14 (кодът на 'o' е 111, кодът на 'a' е 97; 111-97 = 14). Следващият ред се смята на обратно, защото тогава отправната точка е низът `scary` и кодовете се изваждат в обратен ред. В третия случай разликата е още във втория символ, тъй като се прави разлика между големи и малки букви – в `scary` символът "с" има код 99, а в `score` главното "С" е 67 и връщаният резултат е 32.

4. Операции със символни низове



Сравняването на низове с оператора `==` в Java е груба грешка, защото този оператор сравнява адресите на низовете, не техните стойности! За сравняване на низове използвайте методите `equals()` / `equalsIgnorecase()` и `compareTo()` / `compareToIgnoreCase()` и проверявайте за изрично за `null`, защото извикването на `equals()` върху `null` стойност предизвиква `NullPointerException`.

Оптимизация на паметта при символни низове.

```
String hello = "Hello";  
String same = "Hello";  
System.out.println(same == hello); // true
```

Съществува оптимизация във виртуалната машина, която спестява създаването на дублиращи символни низове в паметта. Тази оптимизация се нарича `strings interning` (интерниране на низовете) и благодарение на нея двете променливи в паметта се записват да сочат към едно и също място в паметта.

4. Операции със символни низове



Операции за манипулация на символни низове.

Символните низове са неизменими! Всяка промяна на променлива от тип **String** създава нов низ, в който се записва резултатът. Така че операциите, които се прилагат на символните низове, връщат като резултат препратка към получения резултат.

Долепване на низове (конкатенация). Долепването на символни низове и получаването на нов, резултатен низ, се нарича конкатенация. То може да бъде извършено по 2 начина: чрез метода `concat(...)` или с оператора `+`, или `+=`.

```
String greet = "Hello, "; String name = "reader!";
```

```
String result = greet.concat(name);
```

```
String result = greet + name;
```


```
result = result + " How are you?"; // използване на + и +=
```

4. Операции със символни низове



Търсене на низ в друг низ. Java платформата предоставя 2 метода за търсене на низове: `indexOf(...)` и `lastIndexOf(...)`, които претърсват съдържанието на текстова последователност, но в различна посока.

```
String book = "Introduction to Java book";  
int index = book.indexOf("Java");  
System.out.println(index);           // index = 16
```



Всички срещания на дадена дума. Пример:

```
String quote = "The main subject in the \"Intro Java\" +  
               \" book is Java for Java newbies.\"";  
int index = quote.indexOf("Java");  
while(index != -1) {  
    System.out.println("Java found on index: " + index);  
    index = quote.indexOf("Java", index + 1);  
}
```

Important!

4. Операции със символни низове



Извличане на част от низ. Използвайки `substring(...)` метода, можем да извлечем даден подниз по зададени начална и крайна позиция в текста.

Извикването на метода `substring(index1, index2)` извлича подниз на дадена променлива, който се намира между `index1` и `(index2 - 1)` включително. Символът на посочената позиция `<index2>` не се взима предвид! Например, ако посочим `substring(5, 10)`, ще бъдат извлечени символите между индекс 5 и 9 включително, а не между 5 и 10!

Разцепване на низ по разделител. Методът `split(...)` дава възможност за разцепване на един стринг по разделител или група разделители.

```
String listOfBeers = "Amstel, Zagorka, Tuborg, Becks.";
```

```
String[] beersArr = listOfBeers.split("[ ,.]"); // , и . са разделители
```

Премахване на празните елементи “ “ с +:

```
String[] beersArr = listOfBeers.split("[ ,.]+");
```

4. Операции със символни низове



Регулярни изрази (regular expression) –това са символни низове, представящи множества или подмножества. Например:

- . (символът точка) – обхваща всички възможни символи (може да прихваща или не обозначенията за нов ред)
- \d – обхваща всички цифри (еквивалентно на [0-9])
- \D – обхваща всички символи, които не са цифри (еквивалентно на [^0-9])
- \s – знак за интервали: [\t\n\r\f]
- \S – всички знаци, освен тези за интервали: [^\s]
- \w – всички символи, считани за дума: [a-zA-Z_0-9]
- \W – еквивалентно на [^\w]

```
String small = "[a-z]";
```

```
String allLetters= "[a-zA-Za-яА-Я]";
```

```
String nan = "[^0-9]";
```


4. Операции със символни низове



Замяна на един подниз с друг. Използване на метода **replace(...)**:

```
String doc = "Hello, some@mail.bg, " +  
    "you have been using some@mail.bg in your registration.";  
String fixedDoc = doc.replace("some@mail.bg", "fixed@some.bg");  
System.out.println(fixedDoc);
```

Когато се налага да работим с по-обща информация, може да се използва метода **replaceAll(...)**:

```
String doc = "Smith's number: 0892880022 \n"+  
    "Franky can be found at 0853445566 \n" +  
    "so as Steven - 0811654321";  
replacedDoc = doc.replaceAll("(08)[0-9]{8}", "$1*****");  
System.out.println(replacedDoc);
```

Изход: Smith's number: 08*****

4. Операции със символни низове



Преминаване към главни и малки букви. Двата метода, които биха свършили работата в случая, са **toLowerCase()** и **toUpperCase()**.

```
String text = "All Kind OF LeTTeRs";  
System.out.println(text.toLowerCase()); // === all kind of letters
```

Премахване на празно пространство в края на низ. Методът **trim()** се грижи за премахването на паразитните празни места в текста.

```
String fileData = "  \n\n  Mario Peshev  ";  
String reduced = fileData.trim(); // Mario Peshev
```

4. Операции със символни низове



Изграждане на символни низове. Клас **StringBuilder**.

Символните низове в Java са неизменими. Това означава, че всички корекции, приложени върху съществуващ низ, връщат като резултат нов символен низ. Това има много предимства, но в някои случаи може да създаде проблеми с производителността на приложенията, ако не се знае тази съществена особеност.

Долепяне на низове в цикъл: никога да не се прави!

```
String collector = "Numbers: ";  
for (int idx = 1; idx <= 5000; idx++) {  
    collector += idx; // итерацията на цикъла отнема 2-4 s  
}
```

Обработка на символни низове в паметта.

4. Операции със символни низове



Построяване и промяна на низове със **StringBuilder**. Класът `java.lang.StringBuilder` служи за построяване и промяна на символни низове. Той преодолява проблемите с бързодействието, които възникват при конкатениране на низове от тип `String`. Класът е изграден под формата на масив от символи и това, което трябва да знаем за него, е че информацията в него не е неизменима – промените, които се налагат в променливите от тип `StringBuilder`, се извършват в една и съща област от паметта (буфер), което спестява време и ресурси. За промяната на съдържанието не се създава нов обект, а просто се променя текущият.

4. Операции със символни низове



```
import java.util.Date;

public class NumbersConcatenator {
    public static void main(String[] args) {
        System.out.println(new Date());
        String collector = "Numbers: ";
        for(int idx = 1; idx <= 50000; idx++) {
            collector += idx;
        }
        System.out.println(collector.substring(0, 1024));
        System.out.println(new Date());
    }
}
```

4. Операции със символни низове



```
import java.util.Date;

public class NumbersConcatenatorEllegant {
    public static void main(String[] args) {
        System.out.println(new Date());
        StringBuilder sb = new StringBuilder();
        sb.append("Numbers: ");
        for(int idx = 1; idx <= 50000; idx++) {
            sb.append(idx);
        }
        System.out.println(sb.substring(0, 1024));
        System.out.println(new Date());
    }
}
```

4. Операции със символни низове



Класът `StringBuilder` предоставя набор от методи, които помагат за по-лесна и ефективна работа с променливите. Някои от тях са:

- `StringBuilder(int capacity)` – конструктор с параметър начален капацитет. Чрез него може предварително да зададем размера на буфера, ако имаме приблизителна информация за броя итерации и слепвания. Така спестяваме излишни заделяния на динамична памет.
- `capacity()` – връща размера на целия буфер (общия брой заети и свободни символи)
- `length()` – връща дължината на записания низ в променливата
- `charAt(int index)` – връща символа на указаната позиция
- `append(...)` – слепва низ, число или друга стойност след последния записан символ в буфера.

4. Операции със символни низове



- `delete(int start, int end)` – премахва низ по зададена начална и крайна позиция
- `insert(int offset, String str)` – вмъква даден стринг на дадена позиция
- `replace(int start, int end, String str)` – замества записания низ между началната и крайната позиция със стойността на променливата `str`
- `toString()` – връща записаната информация в обекта на `StringBuilder` като резултат от тип `String`, който можем да запишем в променлива на `String`.

Форматиране на низове.

Класът `java.util.Formatter` дава възможност за извеждане на форматиращи символни низове. Сред възможностите на класа са подравняването на текста и различни методи за форматиране на текст, символи, дати и специфичен изход в зависимост от местоположението.

Всеки метод, който връща форматиран изход, изисква форматиращ стринг и списък от аргументи. Форматиращият низ е `String` обект, който съдържа фиксиран текст и един или повече вложени форматиращи спецификатори (`format specifiers`). Основните спецификатори за символни и числови типове имат следния синтаксис:

`%[индекс_на_аргумента$][флагове][ширина][.точност]формат`

Основните параметри на израза са:

-
- индекс_на_аргумента — незадължителен спецификатор; десетично число, указващо позицията на аргумента. Първият аргумент има индекс "1\$", вторият — "2\$", и т.н.
 - флагове — незадължителен спецификатор; списък от символи, модифициращи начина на извеждане на низа. Зависи пряко от формата.
 - ширина — незадължителен спецификатор; неотрицателно десетично число, посочващо минималния брой от символи, които да бъдат изведени на изхода. Удобен за таблично форматиране.
 - точност — незадължителен спецификатор; неотрицателно десетично число, ограничаващ броя символи. Зависи от типа формат, широко използван при десетични числа.
 - формат (conversion) — символ, указващ как да бъде форматиран аргументът. Зависи от типа на подадения аргумент.

Служебният метод `toString()`.

Един от основните помощници за представянето на обектите като символни низове е методът `toString()`. Той е заложен в дефиницията на класа **Object** – базовият клас, който наследяват пряко или не всички референтни типове в езика. По този начин дефиницията на методи се появява във всеки един клас, като се има възможност да се изведе под някаква форма съдържанието на един обект като текст.

```
Date currentDate = new Date();  
System.out.println(currentDate);  
System.out.println(currentDate.toString());
```

Използване на `String.format()`.

`String.format()` е статичен метод, чрез който може да създаваме форматиращи стрингове, на които да подаваме параметри. Той е удобен при създаването на шаблони – често срещани текстове с променливи параметри. С негова помощ можем да използваме низове с деклариращи параметри и всеки път да променяме единствено параметрите.

5. Преобразуване на типове



Преобразуване към числови типове.

За преобразуване на символен низ към число може да използваме обвиващите класове (wrapper classes) на примитивните типове. Всички примитивни типове имат прилежащите им класове, които служат за представянето на примитивна стойност като обект и предоставят често използвани методи, които може да се ползват наготово.

Обвиващите класове, без Character, предлагат методи за преобразуването на текстова променлива към променлива от примитивния тип, с който е обвързан с обвиващия клас. Методите имат формата `parseXXX(String)`, като на мястото на `XXX` е името на типа (например `parseInt(...)`, `parseBoolean(...)` и др.).

```
String text = "53";
```

```
int intValue = Integer.parseInt(text); // 53
```

5. Преобразуване на типове



Преобразуване на дати.

```
String text = "27.10.2008";  
String [] dateElements = text.split("[.]");  
String dayString = dateElements[0];  
String monthString = dateElements[1];  
String yearString = dateElements[2];  
int day = Integer.parseInt(dayString);  
int month = Integer.parseInt(monthString);  
int year = Integer.parseInt(yearString);  
  
Calendar cal = new GregorianCalendar(year, month - 1, day);  
Date date = cal.getTime(); // Mon Oct 27 00:00:00 EET 2008
```

Класът `java.text.SimpleDateFormat` съдържа функционалност, чрез която се постига по-елегантно преобразуване на типовете. Той дава възможност за преобразуване на текстово съдържание към дата, както и обратното.

```
SimpleDateFormat sdf = new SimpleDateFormat("dd.MM.yyyy");  
Date date = sdf.parse("27.10.2008");  
System.out.println(date); // Mon Oct 27 00:00:00 EET 2008
```



Въпроси ?