

Disciplina: Linguagem de Programação II
Prof. Lucas Gonçalves Nadalete

Aula 11 – Strings, StringBuffer e StringBuilder

String

- String é um objeto **imutável**
- **Qualquer alteração** em um objeto String, um **novo objeto** String é criado para armazenar as alterações
- Exemplo:
 - `String ab = "minha string";`
 - `// esse objeto ficará perdido na memória após a execução da linha abaixo`
 - `ab += "12";`
 - `// novo objeto string é criado com o valor: "minha string12"`

Strings

- `String str = new String("minha String");`
- `String str2 = "minha String";`
- Lembre-se, toda vez que se utilizar **new**, um novo objeto será criado.
- Na primeira linha, dois objetos foram criados e uma referência, enquanto na segunda, apenas um objeto e uma referência.

Concatenação de Strings

- Exemplo clássico:
 - operador de concatenação é muito lento
- A JVM cria um novo objeto String para cada concatenação
- Isso **acumula memória e estressa** o *Garbage Collector* facilmente

```
view plain print ?  
01. String temp = "";  
02. for (int i = 0; i < 10; i++) {  
03.     temp += "string";  
04. }
```

Métodos da Classe String

- `charAt(...)`:
 - Obtém o caractere localizado na posição especificada
- `trim()`:
 - Obtém o texto equivalente sem espaços laterais
- `equals(...)`:
 - Compara conteúdo, caractere-a-caractere
- `equalsIgnoreCase(...)`:
 - Compara conteúdo, caractere-a-caractere ignorando o aspecto maiúsculo/minúsculo

Métodos da Classe String

- `length()` :
 - Obtém a quantidade de caracteres
- `contains()` :
 - Verifica se o conteúdo tem uma determinada cadeia de string específica
- `toUpperCase()` :
 - Transforma a cadeia de String para letras em maiúsculo
- `toLowerCase()`
 - Transforma a cadeia de String para letras em minúsculo
- `replace(caracter a ser substituído, caracter novo)`
- `replaceAll(caracter velho, caractere novo)`

Métodos da Classe String

- `indexOf(...)` :
 - Obtém a posição da primeira ocorrência do argumento;
- `lastIndexOf(...)` :
 - Obtém a posição da última ocorrência do argumento;
- `substring(int begin, int end)` :
 - retorna uma nova string que é uma substring da string corrente, iniciando em *begin* e terminando em *end*.

Funcionamento do *substring*

L	I	N	G	U	A	G	E	M		J	A	V	A	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

`String x = frase.substring(10);` // retorna "JAVA"

`String x = frase.substring(3, 9);` // retorna "GUAGEM"

Método *split()*

- O método `split()` quebra uma `String` em várias substrings a partir de um caractere definido
- Exemplo:

```
String texto = "banana,maçã,laranja";  
String frutas[] = texto.split(",");  
System.out.println(frutas[0]); //imprime banana  
System.out.println(frutas[1]); //imprime maçã  
System.out.println(frutas[2]); //imprime laranja
```

Método *concat*

```
String str1 = "teste";  
str1.concat("imutável");  
str1.toUpperCase();  
System.out.println(str1);
```

- O que será impresso na tela?
- O que acontece com a string "imutável"?
- O que acontece com a string "IMUTÁVEL"?
- Qual é a diferença do método concat para o operador + no contexto da Classe String?

Pool Interno de Strings

- Lugar privativo da JVM que possui referências para algumas Strings já utilizadas pelo programa
- Pode ser visto como um cache de Strings
- É utilizado para otimizar o uso de Strings
- Com isso, a JVM economiza memória

Exemplo que utiliza o Pool de Strings

```
String str1 = "minha String ";  
String str2 = str1.concat("imutável ");  
str1.concat(" teste");  
str2.concat("imutável ");  
str2 = "ok";  
str2.toUpperCase();
```

StringBuilder e StringBuffer

- Criados para tratar o problema da imutabilidade da classe String a fim de otimizar o desempenho
- Altamente recomendável para programas que se faz uso excessivo de concatenações de Strings

StringBuilder e StringBuffer

- O operador += de concatenação é muito lento, pois cria uma nova string para cada concatenação
- Isso acumula memória e estressa facilmente o *Gargabe Collector*
- Solução: StringBuilder e StringBuffer:
 - Não se cria um novo objeto para cada concatenação, ganhando tanto em desempenho quanto em consumo de memória.

StringBuilder e StringBuffer

- O exemplo abaixo é clássico
- Este código é muito mais rápido que o código apresentado anteriormente que faz uso da classe String

```
StringBuffer buffer = new StringBuffer();  
for (int i = 0; i < 10; i++) {  
    buffer.append("string");  
}  
String temp = buffer.toString();
```

StringBuilder vs StringBuffer

- Ambos são análogos, porém a classe StringBuffer suporta *multi threading* (programação concorrente)
- StringBuffer:
 - Possui sincronismo e, portanto, é segura e suporta *threads*
- StringBuilder:
 - Não suporta threads
 - Por esse motivo, é sensivelmente mais rápida

Comparando velocidade StringBuilder vs StringBuffer

```
package teste;

public class TesteSB {
    public static void main(String[] args) {
        long tm = System.currentTimeMillis();
        testeBuffer();
        System.out.println(System.currentTimeMillis()-tm);
        tm = System.currentTimeMillis();
        testeBuilder();
        System.out.println(System.currentTimeMillis()-tm);
    }

    private static void testeBuffer() {
        StringBuffer sb = new StringBuffer();
        for(int i=0; i < 100000000; i++) {
            sb.append(i);
        }
    }

    private static void testeBuilder() {
        StringBuilder sb = new StringBuilder();
        for(int i=0; i < 100000000; i++) {
            sb.append(i);
        }
    }
}
```