

Aula 13

Caracteres, Strings e Expressões Regulares

Introdução

- Para validar entradas no programa, exibir informações para usuário e outras tipos de processamento envolvendo textos é necessário saber como manipular **caracteres**, **strings** e/ou aplicar **expressões regulares**
- **Literais de caracteres**
 - Um literal de caractere é um valor inteiro representado como caractere entre aspas simples
 - Ex: 'z' representa o valor inteiro de z e '\n' representa o valor inteiro de nova linha
 - O valor de um literal de caractere é o valor inteiro do caractere no **conjunto de caracteres Unicode**
(<https://unicode-table.com/pt/#0118>)

Introdução

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         int valorCar1 = 'a';  
15         int valorCar2 = '\n';  
16  
17         System.out.println("Valor do caracter a: " + valorCar1);  
18         System.out.println("Valor do caracter \n: " + valorCar2);  
19     }  
20 }  
21  
22  
23
```

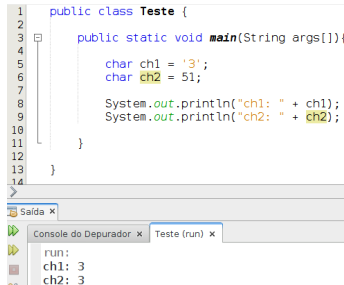
teste.Teste > main >

Saída - Teste (run) x

run:
Valor do caracter a: 97
Valor do caracter \n: 10

Introdução

- Podemos trabalhar com caracteres tanto utilizando o literal qual o valor inteiro do caractere



```
1 public class Teste {  
2  
3     public static void main(String args[]){  
4  
5         char ch1 = '3';  
6         char ch2 = 51;  
7  
8         System.out.println("ch1: " + ch1);  
9         System.out.println("ch2: " + ch2);  
10  
11     }  
12  
13 }  
14
```

Salida x

Console do Depurador x Teste (run) x

```
run:  
ch1: 3  
ch2: 3
```

- OBSERVAÇÃO:** internamente o literal será convertido para o valor inteiro

Introdução

- Uma *string* pode incluir letras, dígitos e vários caracteres especiais, como +, -, *, / e \$
- Uma *string* é um objeto da classe String
- Os **literais de string** (armazenados na memória como objetos String) são escritos como uma sequência de caracteres entre aspas duplas
- **Ex:** “Rafael Geraldeli Rossi”, “Programação Orientada a Objetos” e “Tirar 10 na prova!!”

Construtores

- 1 A classe **String** fornece diferentes construtores para inicializar objetos **String**
- 2 Todos os construtores podem ser vistos na documentação da classe **String** (<http://docs.oracle.com/javase/8/docs/api/java/lang/String.html>)
- 3 O construtor mais comum é o **String(String valor)**
- 4 **OBSERVAÇÃO:** lembrando que a classe **String** tem uma facilidade de inicialização pertencente aos *Wrappers* na qual podemos colocar diretamente o conteúdo na variável sem precisar alocar memória e chamar o construtor

Construtores

```
1 public class Teste {
2
3     public static void main(String args[]){
4
5         String string1 = "Programação";
6         char[] string2 = {'0','b','j','e','t','o','s'};
7
8         String s1 = new String();
9         String s2 = new String(string1);
10        String s3 = new String("Orientada");
11        String s4 = new String(string2,2,3);
12
13        System.out.println("s1: " + s1);
14        System.out.println("s2: " + s2);
15        System.out.println("s3: " + s3);
16        System.out.println("s4: " + s4);
17    }
18 }
19
20 }
```

Teste > main >

Saída x

Console do Depurador x Teste (run) x

```
run:
s1:
s2: Programação
s3: Orientada
s4: jet
```

Métodos length, charAt e getChars

- Método length: retorna o tamanho de uma *string*
- Método charAt: retorna o caractere em uma localização específica em uma *string*
- Método toCharArray: recuperam um conjunto de caracteres de uma String como um array do tipo *char*

Métodos length, charAt e getChars

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         String teste = "Programação Orientada a Objetos";  
15  
16         System.out.println("Tamanho da string: " + teste.length());  
17         System.out.println("Caractere na posição 1: " + teste.charAt(1));  
18         System.out.println("Caractere na posição 10: " + teste.charAt(10));  
19  
20         char[] array = teste.toCharArray();  
21         System.out.print("Imprimindo o array de caracteres: ");  
22         for(int i=0;i<array.length;i++){  
23             System.out.print(array[i] + " ");  
24         }  
25     }  
26 }  
27  
28 }
```

Saída - Teste (run) x

```
run:  
Tamanho da string: 31  
Caractere na posição 1: r  
Caractere na posição 10: o  
Imprimindo o array de caracteres: P r o g r a m a ç ã o   O r i e n t a d a   a   O b j e t o s
```

Comparando Strings

- Frequentemente *strings* são comparadas para verificar **igualdade**, a **ordem relativa** (qual *string* deve aparecer antes da outra em uma ordenação), ou ainda **se uma string está contida em outra**
- Todos os caracteres são representados no computador por valores numéricos
- Quando o computador compara *strings*, ele na verdade está comparando os valores numéricos de cada caractere que compões a *strings*

Comparando Strings: equals

- Como já vimos anteriormente, podemos comparar *strings* utilizando o método `equals`
- O método `equals` utiliza uma **comparação lexicográfica** → compara os valores inteiros Unicode que representam cada caractere em cada *string*
- Portanto se a *string* ‘‘Hello’’ é comparada com a *string* ‘‘HELLO’’ o resultado é `false`, pois a representação de inteiro de uma letra minúscula é diferente da representação de inteiro da letra maiúscula

Comparando Strings: equals

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         String s1 = new String("Java");  
15         String s2 = new String("Java");  
16         String s3 = new String("Linguagem");  
17  
18         if(s1.equals(s2)){  
19             System.out.println("s1 e s2 são iguais");  
20         }else{  
21             System.out.println("s1 e s2 são diferentes");  
22         }  
23  
24         if(s1.equals(s3)){  
25             System.out.println("s1 e s3 são iguais");  
26         }else{  
27             System.out.println("s1 e s3 são diferentes");  
28         }  
29     }  
30 }  
31  
32 }  
33 }  
34 }  
35 }  
36 }  
37 }  
38 }  
39 }  
40 }  
41 }  
42 }  
43 }  
44 }  
45 }  
46 }  
47 }  
48 }  
49 }  
50 }  
51 }  
52 }  
53 }  
54 }  
55 }  
56 }  
57 }  
58 }  
59 }  
60 }  
61 }  
62 }  
63 }  
64 }  
65 }  
66 }  
67 }  
68 }  
69 }  
70 }  
71 }  
72 }  
73 }  
74 }  
75 }  
76 }  
77 }  
78 }  
79 }  
80 }  
81 }  
82 }  
83 }  
84 }  
85 }  
86 }  
87 }  
88 }  
89 }  
90 }  
91 }  
92 }  
93 }  
94 }  
95 }  
96 }  
97 }  
98 }  
99 }  
100 }
```

teste.Teste > main >

Saída - Teste (run) x

```
run:  
s1 e s2 são iguais  
s1 e s3 são diferentes
```

Comparando Strings: equalsIgnoreCase

- Para comparar strings ignorando as caixas dos caracteres pode-se utilizar o método `equalsIgnoreCase`

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         String s1 = new String("Java");  
15         String s2 = new String("java");  
16  
17         System.out.println("Utilizando o método equals");  
18         if(s1.equals(s2)){  
19             System.out.println("s1 e s2 são iguais");  
20         }else{  
21             System.out.println("s1 e s2 são diferentes");  
22         }  
23  
24         System.out.println("Utilizando o método equalsIgnoreCase");  
25         if(s1.equalsIgnoreCase(s2)){  
26             System.out.println("s1 e s2 são iguais");  
27         }else{  
28             System.out.println("s1 e s2 são diferentes");  
29         }  
30     }  
31 }  
32  
33 }
```

Saída - Teste (run) x

```
run:  
Utilizando o método equals  
s1 e s2 são diferentes  
Utilizando o método equalsIgnoreCase  
s1 e s2 são iguais
```

Comparando Strings: `compareTo`

- O método `compareTo` é declarado na interface `Comparable` e implementado na classe `String`
- O método `compareTo` retorna:
 - **0** se as *strings* forem iguais
 - **Um número negativo** se a *string* que invoca `compareTo` for menor que a *string* que é passada como argumento
 - **Um número positivo** se a *string* que invoca `compareTo` for maior que a *string* que é passada como argumento
- Também pode ser aplicado o método `compareToIgnoreCase(...)` para ignorar as caixas das *strings*

Comparando Strings: compareTo

```
10 public class Teste {
11
12     public static void main(String[] args) {
13
14         String s1 = new String("Java");
15         String s2 = new String("java");
16         String s3 = new String("Rafael");
17         String s4 = new String("Abacate");
18
19         System.out.println("Comparando s1 com s2? " + s1.compareTo(s1));
20         System.out.println("Comparando s1 com s2? " + s1.compareTo(s2));
21         System.out.println("Comparando s1 com s3? " + s1.compareTo(s3));
22         System.out.println("Comparando s1 com s4? " + s1.compareTo(s4));
23
24         if(s1.compareTo(s4) < 0){
25             System.out.println("S1 é menor que s4");
26         }else{
27             System.out.println("S1 é maior que s4");
28         }
29     }
30 }
31
32
```

teste.Teste > main > if (s1.compareTo(s4) < 0) else >

Saída - Teste (run) x

```
run:
Comparando s1 com s2? 0
Comparando s1 com s2? -32
Comparando s1 com s3? -8
Comparando s1 com s4? 9
S1 é maior que s4
```

Comparando Strings: contains

- O método `contains` retorna `true` se uma *string* está contida dentro da outra e `false` caso contrário

```
12 public class Teste {  
13  
14     public static void main(String[] args) {  
15  
16         String s1 = new String("Sistemas de Informação");  
17  
18         System.out.println("\"Info\" está contido em s1? " + s1.contains("Info"));  
19         System.out.println("\"Abacate\" está contido em s1? " + s1.contains("Abacate"));  
20  
21     }  
22  
23 }  
24
```

Resultados da Pesquisa Saída x

Console do Depurador x Teste (run) x

run:
"Info" está contido em s1? true
"Abacate" está contido em s1? false

Comparando Strings: startsWith

- O método `startsWith(String arg)` retorna `true` se uma *string* começa com o argumento `arg` e `false` caso contrário

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         String s1 = new String("started");  
15         String s2 = new String("starting");  
16         String s3 = new String("ended");  
17         String s4 = new String("ending");  
18  
19         System.out.println("String 1 começar com \"start\"? " + s1.startsWith("start"));  
20         System.out.println("String 2 começar com \"start\"? " + s2.startsWith("start"));  
21         System.out.println("String 3 começar com \"ends\"? " + s3.startsWith("ends"));  
22         System.out.println("String 2 começar com \"endi\"? " + s1.startsWith("endi"));  
23  
24     }  
25  
26 }
```

Saída - Teste (run) x

```
run:  
String 1 começar com "start"? true  
String 2 começar com "start"? true  
String 3 começar com "ends"? false  
String 2 começar com "endi"? false
```

Comparando Strings: endsWith

- O método `endsWith(String arg)` retorna `true` se uma *string* termina com o argumento `arg` e `false` caso contrário

```
10 public class Teste {
11
12     public static void main(String[] args) {
13
14         String s1 = new String("started");
15         String s2 = new String("starting");
16         String s3 = new String("ended");
17         String s4 = new String("ending");
18
19         System.out.println("String 1 começar com \"ed\"? " + s1.endsWith("ed"));
20         System.out.println("String 2 começar com \"ing\"? " + s2.endsWith("ing"));
21         System.out.println("String 3 começar com \"ing\"? " + s3.endsWith("ing"));
22         System.out.println("String 2 começar com \"ed\"? " + s1.endsWith("ed"));
23
24     }
25
26 }
```

Saída - Teste (run) x

```
run:
String 1 começar com "ed"? true
String 2 começar com "ing"? true
String 3 começar com "ing"? false
String 2 começar com "ed"? true
```

Localizando Caracteres e *Substrings* em *Strings*

- Costuma ser útil pesquisar uma *string* para um caractere ou conjuntos de caracteres
- **Exs:**
 - Capacidade de realizar buscas em documentos
 - Segmentar textos → nome de arquivo em um diretório
 - Verificar se o usuário forneceu algum tipo de informações necessárias ou caracteres específicos
 - ...

Localizando caracteres e *Substrings* em *Strings*: `indexOf`

- O método `indexOf` retorna o índice (inicial) da posição de um caractere ou de uma *substring* em uma *string*
- Variantes do método `indexOf` permite procurar por um caractere ou *string* a partir de um índice especificado
- Caso o caractere ou *string* não seja encontrado, o retorno do método `indexOf` é `-1`

Localizando caracteres e *Substrings* em *Strings*: indexOf

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         String s1 = new String("Bafael Geraldeli Rossi");  
15  
16         System.out.println("Posição da primeira ocorrência da letra 'a' na String 1? " + s1.indexOf('a'));  
17         System.out.println("Posição da ocorrência da letra 'a' após o 5º caractere na String 1: " + s1.indexOf('a',5));  
18         System.out.println("Posição da string 'Geraldeli' na String 1: " + s1.indexOf("Geraldeli"));  
19         System.out.println("Posição da string 'Abacate' na String 1: " + s1.indexOf("Abacate"));  
20     }  
21 }  
22  
23  
24
```

teste.Teste > main >

Saída - Teste (run) x

```
run:  
Posição da primeira ocorrência da letra "a" na String 1? 1  
Posição da ocorrência da letra "a" após o 5º caractere na String 1: 10  
Posição da string "Geraldeli" na String 1: 7  
Posição da string "Abacate" na String 1: -1
```

Localizando Caracteres e *Substrings* em *Strings*: `lastIndexOf`

- O método `lastIndexOf` tem um funcionamento semelhante ao método `indexOf` mas retorna o último índice de um determinado caractere ou ocorrência
- Pode-se também especificar um índice de referência → o método `lastIndexOf` irá retornar a primeira ocorrência para trás do índice especificado

Localizando Caracteres e Substrings em Strings: lastIndexOf

```
10 public class Teste {
11
12     public static void main(String[] args) {
13
14         String s1 = new String("/home/rafael/Disciplinas/POO_1_2016/Aulas/Aula_13_-_String,_Caracteres_e_Expressões_Regulares");
15
16         System.out.println("Posição da última ocorrência do caractere / na String 1? " + s1.lastIndexOf('/'));
17         System.out.println("Posição da ocorrência da letra 'a' após o 5º caractere na String 1: " + s1.lastIndexOf('/', 8));
18         System.out.println("Posição da string \"Aulas\" na String 1: " + s1.indexOf("Aulas"));
19         System.out.println("Posição da string \"Abacate\" na String 1: " + s1.indexOf("Abacate"));
20
21     }
22
23 }
24
```

Saída - Teste (run) x

```
run:
Posição da última ocorrência do caractere / na String 1? 41
Posição da ocorrência da letra "a" após o 5º caractere na String 1: 5
Posição da string "Aulas" na String 1: 36
Posição da string "Abacate" na String 1: -1
```

Extraindo *Substring* de *Strings*

- Para extrair *substrings* de *strings* pode-se utilizar duas versões do método substring: `substring(int posicaoInicial)` e `substring(int posicaoInicial, int posicaoFinal)`

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         String s1 = new String("Programação Orientada a Objetos");  
15  
16         System.out.println("Substring da s1 indo de 0 a 15: " + s1.substring(15));  
17         System.out.println("Substring da s1 indo de 0 a 21: " + s1.substring(0,21));  
18  
19     }  
20  
21 }  
22
```

Saída - Teste (run) x

```
run:  
Substring da s1 indo de 0 a 15: entada a Objetos  
Substring da s1 indo de 0 a 21: Programação Orientada
```


Concatenando *Strings*: concat

- Já vimos na disciplina que podemos usar o operador + para concatenar *strings*
- Para tal finalidade, pode-se também utilizar o método `concat(String str)`

```
10 public class Teste {
11
12     public static void main(String[] args) {
13
14         String s1 = new String("Programação");
15         String s2 = new String(" Orientada a Objetos");
16
17         System.out.println("Concatenando s1 com s2 utilizando o operador +: " + s1 + s2);
18         System.out.println("Concatenando s1 com s2 utilizando o operador concat: " + s1.concat(s2));
19
20     }
21
22 }
23
```

Saída - Teste (run) X

run:
Concatenando s1 com s2 utilizando o operador +: Programação Orientada a Objetos
Concatenando s1 com s2 utilizando o operador concat: Programação Orientada a Objetos

Métodos String Diversos

- `replace`: retorna um novo objeto *string* em que cada ocorrência de uma determinado caractere ou *string* informado pelo usuário é substituído por outro caractere/*string* também informado pelo usuário
- `replaceAll`: mesma coisa que o método `replace` mas pode-se usar expressões regulares como argumentos
- `replaceFirst`: mesma coisa que o método `replaceAll` mas substitui apenas a primeira ocorrência da expressão regular fornecida pelo usuário

Métodos String Diversos

- **toUpperCase**: converte todos os caracteres de uma *string* para caixa alta
- **toLowerCase**: converte todos os caracteres de uma *string* para caixa baixa
- **trim**: remove todos os caracteres de espaço em branco que aparecem no início ou no fim da *string* em que é invocado

Métodos String Diversos

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         String s1 = new String("ProGraMação");  
15         String s2 = new String("    Orientada a Objetos");  
16  
17         System.out.println("String s1 com caixa baixa: " + s1.toLowerCase());  
18         System.out.println("String s2 com caixa alta: " + s2.toUpperCase());  
19         System.out.println("String s2 sem espaços em branco: " + s2.trim());  
20         System.out.println("Substituindo 'a' por 'e' na string s1: " + s1.replace('a', 'e'));  
21  
22     }  
23  
24 }  
25
```

Saída - Teste (run) x

```
run:  
String s1 com caixa baixa: programação  
String s2 com caixa alta: ORIENTADA A OBJETOS  
String s2 sem espaços em branco: Orientada a Objetos  
Substituindo 'a' por 'e' na string s1: ProGreMeção
```

Convertendo Tipos Primitivos e Por Referência em Strings: `valueOf`

- `valueOf` é um método estático da classe `String` que recebe tanto tipos primitivos e argumentos como referência e converte o argumento em uma *string*
- No caso de tipos por referência, é invocado o método `toString()` dos objetos para realizar a conversão

Convertendo Tipos Primitivos e Por Referência em Strings: `valueOf`

```
10 public class Teste {
11
12     public static void main(String[] args) {
13
14         int num1 = 50;
15         double num2 = 560.70;
16         boolean boolean1 = false;
17         Pessoa pessoal = new Pessoa("Rafael", 30);
18
19         String s1 = String.valueOf(num1);
20         String s2 = String.valueOf(num2);
21         String s3 = String.valueOf(boolean1);
22         String s4 = String.valueOf(pessoal);
23
24         System.out.println("- s1: " + s1);
25         System.out.println("- s2: " + s2);
26         System.out.println("- s3: " + s3);
27         System.out.println("- s4: " + s4);
28     }
29 }
30
31 }
```

Saída - Teste (run) x

```
run:
- s1: 50
- s2: 560.7
- s3: false
- s4: nome=Rafael, idade=30
```

Classe StringBuilder

Strings são imutáveis!!!



**Mas se são imutáveis, porque eu consigo, por exemplo,
concatenar strings????**



Classe StringBuilder

- A “concatenação”, na verdade, é uma criação de uma nova região de memória para acomodar a união das outras duas *strings* que já estavam em memória, as quais serão copiadas para a nova região da memória
- **Consequência:** ao concatenar várias *strings* o processo pode se tornar muito lento
- Para lidar com tratamentos dinâmicos de *strings* (inserções, remoções e concatenações), o Java fornece classes apropriadas para isso: **StringBuilder** e **StringBuffer**

Classe StringBuilder

- A classe `StringBuilder` é utilizada para criar e manipular informações de *string* de maneira dinâmica
- Cada `Stringbuilder` é capaz de armazenar um número de caracteres especificado pela sua capacidade
- Se a capacidade de um `StringBuilder` for excedida, a capacidade se expande para acomodar os caracteres adicionais

Classe StringBuilder

- **OBSERVAÇÃO 1:** em programas que realizam a concatenação de *strings*, ou outras modificações de *strings*, em geral, é mais eficiente implementar as modificações com a classe `StringBuilder`
- **OBSERVAÇÃO 2:** `StringBuilders` não são seguros para *threads*. Se múltiplas *threads* exigirem acesso às mesmas informações de *string* dinâmicas utilize a classe `StringBuffer`. As classes `StringBuilder` e `StringBuffer` fornecem capacidades idênticas, mas a classe `StringBuffer` é segura para *threads*

Construtores da classe StringBuilder

- A classe `StringBuilder` fornece 4 construtores
 - `StringBuilder()`: constrói um `StringBuilder` sem caracteres e com capacidade inicial de 16 caracteres
 - `StringBuilder(CharSequence seq)` ou `StringBuilder(String str)`: constrói um `StringBuilder` que contém os caracteres especificado no argumento
 - `StringBuilder(int capacidade)`: constrói um `StringBuilder` sem caracteres e com capacidade inicial conforme especificado no argumento

Construtores da classe StringBuilder

```
12 public class Teste {  
13  
14     public static void main(String[] args) {  
15  
16         StringBuilder sb1 = new StringBuilder();  
17         StringBuilder sb2 = new StringBuilder("Teste");  
18         StringBuilder sb3 = new StringBuilder(50);  
19  
20         System.out.println("-sb1: " + sb1.toString());  
21         System.out.println("-sb2: " + sb2.toString());  
22         System.out.println("-sb3: " + sb3.toString());  
23  
24     }  
25  
26 }  
27
```

Resultados da Pesquisa Saída x

Console do Depurador x Teste (run) x

```
run:  
-sb1:  
-sb2: Teste  
-sb3:
```

Métodos `StringBuilder` `length`, `capacity`, `setLength` e `ensureCapacity`

- `length`: retorna o número de caracteres atualmente em um `StringBuilder`
- `capacity`: retorna o número de caracteres que pode ser armazenado em um `StringBuilder` sem alocar mais memória
- `ensureCapacity`: garante que um `StringBuilder` tenha pelo menos a capacidade especificada
- `setLength`: aumenta ou diminui o comprimento de uma `StringBuilder`

Métodos `length`, `capacity`, `setLength` e `ensureCapacity`

```
10 public class Teste {
11
12     public static void main(String[] args) {
13
14         StringBuilder sb = new StringBuilder("Programação Orientada a Objetos");
15         System.out.println("Conteúdo: " + sb.toString());
16         System.out.println("Tamanho: " + sb.length());
17         System.out.println("Capacidade: " + sb.capacity());
18
19         sb.ensureCapacity(75);
20         System.out.println("Nova Capacidade: " + sb.capacity());
21
22         sb.setLength(10);
23         System.out.println("Novo Tamanho: " + sb.length());
24         System.out.println("Novo Conteúdo: " + sb.toString());
25     }
26 }
27
28
29
```

Saída - Teste (run) x

```
run:
Conteúdo: Programação Orientada a Objetos
Tamanho: 31
Capacidade: 47
Nova Capacidade: 96
Novo Tamanho: 10
Novo Conteúdo: Programaçã
```

Métodos `charAt`, `setCharAt`, `getChars` e `reverse`

- `charAt`: aceita um argumento inteiro e retorna o caractere no índice correspondente ao argumento
- `getChars`: recebe como argumentos posição inicial e final dos caracteres a serem copiados, o array no qual os caracteres serão copiados, e um índice inicial no qual os *arrays* serão copiados no array de destino
- `reverse`: inverte o conteúdo do `StringBuilder`
- `setCharAt`: aceita um argumento inteiro e um argumento caractere e seta o caractere na posição especificada

Métodos charAt, setCharAt, getChars e reverse

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         StringBuilder sb = new StringBuilder("Programação Orientada a Objetos");  
15         System.out.println("String completa: " + sb.toString());  
16         System.out.println("Caractere na posição 1: " + sb.charAt(1));  
17         System.out.println("Caractere na posição 1: " + sb.charAt(5));  
18  
19         char[] arrayChars = new char[5];  
20         sb.getChars(5, 10, arrayChars, 0);  
21         for(char c : arrayChars){  
22             System.out.print(c);  
23         }  
24         System.out.println();  
25  
26         sb.setCharAt(0, 'B');  
27         sb.setCharAt(1, 'l');  
28         System.out.println("Nova string completa: " + sb.toString());  
29         System.out.println("Nova string reversa: " + sb.reverse().toString());  
30  
31     }  
32  
33 }  
34
```

teste.Teste > main >

Salida - Teste (run) x

```
run:  
String completa: Programação Orientada a Objetos  
Caractere na posição 1: r  
Caractere na posição 1: a  
amaçã  
Nova string completa: Blogramação Orientada a Objetos  
Nova string reversa: sotejb0 a adatneir0 oãçamargolB
```


Método `append`

- A classe `StringBuilder` fornece métodos `append` sobrecarregados para permitir que valores de vários tipos sejam acrescentados no fim de um `StringBuilder`
- São fornecidas versões para cada um dos tipos primitivos, para arrays de caractere, *strings* e objetos

Método append

```
12 public class Teste {  
13  
14     public static void main(String[] args) {  
15  
16         StringBuilder sb = new StringBuilder();  
17         sb.append("Programação ");  
18         sb.append("Orientada ");  
19         sb.append("a ");  
20         sb.append("Objetos");  
21  
22         System.out.println(sb.toString());  
23     }  
24  
25 }
```

Resultados da Pesquisa

Saída x



Console do Depurador x

Teste (run) x

run:

Programação Orientada a Objetos

Método append

```
10 public class Teste {
11
12     public static void main(String[] args) {
13
14         StringBuilder sb = new StringBuilder("Teste");
15         String string = " o caramba! ";
16         char[] arrayChars = {'$', '@', '&', '8', '!', ' '};
17         boolean bool = false;
18         char ch = 'P';
19         int valInt = 10;
20         double valDouble = 100.53;
21
22         StringBuilder novoSb = new StringBuilder();
23         novoSb.append(sb);
24         novoSb.append(string);
25         novoSb.append(arrayChars);
26         novoSb.append(bool);
27         novoSb.append(ch);
28         novoSb.append(valInt);
29         novoSb.append(valDouble);
30
31         System.out.println("StringBuilder com as contatenações: " + novoSb.toString());
32     }
33 }
34
35 }
36
```

teste.Teste > main >

Saída - Teste (run) x

run:
StringBuilder com as contatenações: Teste o caramba! \$@&8! falseP10100.53

Comparação entre a concatenação de *strings* e o método append da classe StringBuilder

```
10 public class Teste {
11
12     public static void main(String[] args) {
13
14         long inicioString = System.currentTimeMillis();
15         String string = "";
16         for(int i=0; i<100000; i++){
17             string += "*****";
18         }
19         long fimString = System.currentTimeMillis();
20         long totalString = (fimString - inicioString)/1000;
21         System.out.println("Tempo (s) para concatenar 1000 strings: " + totalString);
22
23         long inicioSb = System.currentTimeMillis();
24         StringBuilder sb = new StringBuilder("");
25         for(int i=0; i<100000; i++){
26             sb.append("*****");
27         }
28         long fimSb = System.currentTimeMillis();
29
30         long totalSb = (fimSb - inicioSb)/1000;
31         System.out.println("Tempo (s) para concatenar 1000 strings com StringBuilder: " + totalSb);
32     }
33 }
34
35 }
36 }
```

Saída - Teste (run) x

```
run:
Tempo (s) para concatenar 1000 strings: 42
Tempo (s) para concatenar 1000 strings com StringBuilder: 0
```

Métodos de Inserção e Exclusão de StringBuilder

- A classe `StringBuilder` fornece métodos `insert` sobrecarregados para inserir valores de vários tipos em qualquer posição em um `StringBuilder`
- Cada método aceita seu segundo argumento, converte-o em uma `String` e o insere no índice especificado pelo primeiro argumento
- Se o primeiro argumento for menor que 0 ou maior que o comprimento do conteúdo da `StringBuilder`, uma `StringIndexOutOfBoundsException` ocorre

Métodos de Inserção e Exclusão de StringBuilder

- A classe `StringBuilder` também fornece métodos `delete` e `deleteCharAt` para excluir caracteres em qualquer posição em um `StringBuilder`
- O método `delete` recebe dois argumentos: o índice inicial e o índice um além do fim dos caracteres a excluir → todos os caracteres que começam no índice inicial, mas não incluindo o índice final, são excluídos
- O método `deleteCharAt` aceita um argumento: o índice do caractere a excluir
- Índices inválidos fazem com que ambos os métodos lancem uma `StringIndexOutOfBoundsException`

Métodos de Inserção e Exclusão de StringBuilder

```
13 public class Teste {  
14  
15     public static void main(String[] args) {  
16  
17         StringBuilder sb = new StringBuilder("Programação Objetos");  
18         System.out.println(sb.toString());  
19  
20         sb.insert(12, "Orientada a ");  
21         System.out.println(sb.toString());  
22  
23         sb.delete(21, 23);  
24         System.out.println(sb.toString());  
25     }  
26 }  
27  
28 }
```

Resultados da Pesquisa

Saída x



Console do Depurador x

Teste (run) x



```
run:  
Programação Objetos  
Programação Orientada a Objetos  
Programação Orientada a Objetos
```

Classe Character

- A maioria dos métodos da classe `Character` são métodos `static` projetados por uma questão de conveniência no processamento de valores `char` individuais
- Esses métodos aceitam pelo menos um argumento caractere e realizam um teste ou uma manipulação do caractere
- A classe `Character` também contém um construtor que recebe um argumento `char` para inicializar um objeto `Character`

Classe Character

- Alguns métodos static da classe Character
 - `isDefined`: retorna true se um caractere está definido no conjunto de caracteres Unicode e false caso contrário
 - `isDigit`: retorna true se um caractere é um dígito e false caso contrário
 - `isJavaIdentifierStart`: retorna true determina se o caractere pode ser o primeiro caractere de um identificador em Java e false caso contrário
 - `isJavaIdentifierPart`: retorna true se um caractere pode ser usado em um identificador em Java e false caso contrário
 - `isLetter`: retorna true se o caractere é uma letra e false caso contrário

Classe Character

- Alguns métodos static da classe Character
 - `isLetterOrDigit`: retorna true se o caractere é uma letra ou um dígito ou false caso contrário
 - `isLowerCase`: retorna true se o caractere é uma letra minúscula e false caso contrário
 - `isUpperCase`: retorna true se o caractere é uma letra minúscula e false caso contrário
 - `toLowerCase`: converte um caractere para caixa baixa
 - `toUpperCase`: converte um caractere para caixa alta

Classe Character

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         char c1 = 'A';  
15         char c2 = 'b';  
16  
17         System.out.println("Caractere c1 é definido? " + Character.isDefined(c1));  
18         System.out.println("Caractere c2 é um dígito? " + Character.isDigit(c2));  
19         System.out.println("Caractere c2 é um caractere inicial de identificadores em Java? " + Character.isDigit(c2));  
20         System.out.println("Caractere c1 é um caractere de identificadores em Java? " + Character.isDigit(c1));  
21         System.out.println("Caractere c1 é uma letra? " + Character.isLetter(c1));  
22         System.out.println("Caractere c1 é caixa baixa? " + Character.isLowerCase(c1));  
23         System.out.println("Caractere c1 é caixa alta? " + Character.isUpperCase(c1));  
24         System.out.println("Convertendo caractere c1 para caixa baixa: " + Character.toLowerCase(c1));  
25         System.out.println("Convertendo caractere c2 para caixa alta: " + Character.toUpperCase(c2));  
26  
27     }  
28  
29 }  
30
```

Saída - Teste (run) x

```
run:  
Caractere c1 é definido? true  
Caractere c2 é um dígito? false  
Caractere c2 é um caractere inicial de identificadores em Java? false  
Caractere c1 é um caractere de identificadores em Java? false  
Caractere c1 é uma letra? true  
Caractere c1 é caixa baixa? false  
Caractere c1 é caixa alta? false  
Convertendo caractere c1 para caixa baixa: a  
Convertendo caractere c2 para caixa alta: B
```

Tokenização de *Strings*

- A classe `String` contém o método `split`, que divide um *string* em seus *tokens* (“pedaços” ou componente)
- Os *tokens* são separados entre si por **delimitadores** (em geral caracteres de espaçamento como espaço, tabulação, nova linha e retorno de carro)
- Outros tipos de caracteres, *strings* ou expressões regulares podem ser utilizados como delimitadores para separar *tokens*
- O retorno do método `split` é um array de `Strings`, sendo que cada elemento desse array é um *token*

Tokenização de *Strings*

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         String teste = new String("Programação Orientada a Objetos");  
15  
16         String[] partes = teste.split(" ");  
17  
18         for(int i=0; i<partes.length; i++){  
19             System.out.println(i + "- " + partes[i]);  
20         }  
21     }  
22 }  
23  
24 }
```

Saída - Teste (run) x

```
run:  
0- Programação  
1- Orientada  
2- a  
3- Objetos
```

Tokenização de *Strings*

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         String teste = new String("Programação Orientada a Objetos");  
15  
16         String[] partes = teste.split("Orientada");  
17  
18         for(int i=0; i<partes.length; i++){  
19             System.out.println(i + "- " + partes[i]);  
20         }  
21  
22     }  
23  
24 }  
25
```

Saída - Teste (run) x

```
run:  
0- Programação  
1- a Objetos
```

Expressões Regulares

- Uma **expressão regular** é uma String especialmente formatada que **descreve um padrão de pesquisa**
- São úteis para **validar dados de entrada, assegurar que os dados estão em um determinado formato** e fazer **buscas não definidas** (ex: uma data qualquer, um ano qualquer, um endereço qualquer, ...)
- **Ex:** um CEP deve consistir em cinco dígitos e um sobrenome deve conter somente letras, espaços, apóstrofos e hífen

Expressões Regulares

- **OBSERVAÇÃO:** compiladores utilizam expressões regulares para validar a sintaxe de um programa
- A classe String fornece vários métodos para realizar operações envolvendo expressões regulares
- A mais simples é a operação de correspondência → método `matches`
- O método `matches` retorna `true` se uma *string* possui um conteúdo que corresponde (ou **casa**) com a expressão regular

Expressões Regulares

● IMPORTANTE

- Há alguns caracteres (metacaracteres) que são utilizados para determinar padrões nas expressões regulares
- Esses caracteres (os mais básicos) são: ., ?, *, +, ^, \$, |, [,], {, }, (,), \
- Portanto pense bem quando for formar uma expressão com esses caracteres

Expressões Regulares

- Uma expressão regular consiste em caracteres literais e símbolos especiais
- **Classes de caractere predefinidas** podem ser utilizadas em expressões regulares
- Uma classe de caractere é uma **sequência de escape que representa um grupo de caracteres**

Caractere	Correspondências	Caractere	Correspondências
<code>\d</code>	Qualquer dígito	<code>\D</code>	Qualquer não dígito
<code>\w</code>	Qualquer caractere de palavra	<code>\W</code>	Qualquer caractere não palavra
<code>\s</code>	Qualquer caractere de espaço em branco	<code>\S</code>	Qualquer caractere não espaço em branco

Expressões Regulares

- Um **caractere de palavra** é qualquer letra (em letras maiúsculas ou minúsculas), qualquer dígito ou caractere sublinhado
- Um **caractere de espaço** em branco é um espaço, uma tabulação, um retorno de carro ou um caractere de nova linha
- Cada **classe de caracteres localiza um único caractere** na *String* que estamos tentando localizar com a expressão regular

Expressões Regulares

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         String st1 = new String("Teste");  
15         String st2 = new String("1985");  
16         String st3 = new String("Orientação a Objetos");  
17         String st4 = new String("20008");  
18  
19         System.out.println("Verificando se st1 contém letras: " + st1.matches("\\w\\w\\w\\w\\w\\w\\w"));  
20         System.out.println("Verificando se st1 contém números: " + st2.matches("\\d\\d\\d\\d\\d"));  
21         System.out.println("Verificando se st2 contém letras: " + st3.matches("\\w\\w\\w\\w\\w\\w\\w\\w\\w\\w"));  
22         System.out.println("Verificando se st2 contém dígitos: " + st4.matches("\\d\\d\\d\\d\\d\\d"));  
23  
24     }  
25  
26 }
```

Saída - Teste (run) x

```
run:  
Verificando se st1 contém letras: true  
Verificando se st1 contém números: true  
Verificando se st2 contém letras: false  
Verificando se st2 contém dígitos: false
```

Expressões Regulares

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         String st1 = new String("13660-000");  
15         String st2 = new String("13660000");  
16         String st3 = new String("10-3581-5252");  
17         String st4 = new String("(35815252)");  
18  
19         System.out.println("Verificando se s1 é um CEP: " + st1.matches("\\d\\d\\d\\d\\d\\d\\d\\d"));  
20         System.out.println("Verificando se s2 é um CEP: " + st2.matches("\\d\\d\\d\\d\\d\\d\\d\\d"));  
21         System.out.println("Verificando se s3 é um telefone: " + st3.matches("\\d\\d\\d\\d\\s\\d\\d\\d\\d\\d\\d\\d\\d\\d"));  
22         System.out.println("Verificando se s4 é um telefone: " + st4.matches("\\d\\d\\d\\d\\s\\d\\d\\d\\d\\d\\d\\d\\d\\d"));  
23     }  
24 }  
25  
26  
27
```

Saída - Teste (run) x

```
run:  
Verificando se s1 é um CEP: true  
Verificando se s2 é um CEP: false  
Verificando se s3 é um telefone: true  
Verificando se s4 é um telefone: false
```

Expressões Regulares

- Para localizar um conjunto de caracteres que não tem uma classe de caracteres predefinidas, pode-se inserir os caracteres dentro de colchetes []
- **Ex:** '[aeiou]' localiza um único caractere que é uma vogal
- Os intervalos de caracteres são representados colocando um hífen (-) entre dois caracteres
- **Ex:** "[A-Z]" identifica uma única letra maiúscula
- Se o primeiro caractere entre colchetes for '^', a expressão aceitará qualquer caractere diferentes dos indicados entre colchetes

Expressões Regulares

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         String st1 = new String("Ab1");  
15         String st2 = new String("zD3");  
16         String st3 = new String("ABC45");  
17  
18         System.out.println("Verificando s1 é composta por duas letras e um dígito: " + st1.matches("[A-Za-z][A-Za-z][0-9]"));  
19         System.out.println("Verificando s2 é composta por duas letras e um dígito: " + st2.matches("[A-Za-z][A-Za-z][0-9]"));  
20         System.out.println("Verificando s3 é composta por duas letras e um dígito: " + st3.matches("[A-Za-z][A-Za-z][0-9]"));  
21  
22     }  
23  
24 }
```

Saída - Teste (run) x

```
run:  
Verificando s1 é composta por duas letras e um dígito: true  
Verificando s2 é composta por duas letras e um dígito: true  
Verificando s3 é composta por duas letras e um dígito: false
```

Expressões Regulares

```
1 public class Teste {  
2  
3     public static void main(String args[]){  
4  
5         String string1 = "A8";  
6         String string2 = "a8";  
7         String string3 = "78";  
8  
9         System.out.println(string1.matches("[^A-Z][0-9]"));  
10        System.out.println(string2.matches("[^A-Z][0-9]"));  
11        System.out.println(string3.matches("[^A-Z][0-9]"));  
12    }  
13 }  
14  
15 }  
16
```

Saída x Matcher.java x

Console do Depurador x Teste (run) x

run:
false
true
true

Expressões Regulares

- O caractere “.” é usado como caractere coringa, isto é, pode representar qualquer tipo de caractere
- **OBSERVAÇÃO:** se quisermos utilizar o caractere “.” como caractere literal, temos que utilizar “\\.”

Expressões Regulares

```
1 public class Teste {  
2  
3     public static void main(String args[]){  
4  
5         String string1 = "A87";  
6         String string2 = "IaC";  
7         String string3 = "E0 ";  
8         String string4 = "Fa";  
9  
10        System.out.println(string1.matches("[A-Z].."));  
11        System.out.println(string2.matches("[A-Z].."));  
12        System.out.println(string3.matches("[A-Z].."));  
13        System.out.println(string4.matches("[A-Z].."));  
14    }  
15  
16 }
```

Saída x Matcher.java x

Console do Depurador x Teste (run) x

```
run:  
true  
true  
true  
false
```

Expressões Regulares

```
1 public class Teste {  
2  
3     public static void main(String args[]){  
4  
5         String string1 = "879.987";  
6         String string2 = "789.ABC";  
7         String string3 = "987-789";  
8         String string4 = "000.777";  
9  
10        System.out.println(string1.matches("\\d\\d\\d\\.\\d\\d\\d"));  
11        System.out.println(string2.matches("\\d\\d\\d\\.\\d\\d\\d"));  
12        System.out.println(string3.matches("\\d\\d\\d\\d\\.\\d\\d\\d"));  
13        System.out.println(string4.matches("\\d\\d\\d\\.\\d\\d\\d"));  
14    }  
15 }  
16 }
```

Saída x Matcher.java x

Console do Depurador x Teste (run) x

```
run:  
true  
false  
false  
true
```

Quantificadores

- Anteriormente tivemos que especificar manualmente o número de dígitos, letras ou espaços nas expressões regulares
- Podemos utilizar **quantificadores para auxiliar na quantificação** dos dígitos/letras/espaço **para facilitar a definição de expressões regulares e aumentar sua flexibilidade**

Quantificador	Correspondências
*	Localiza zero ou mais ocorrências do padrão
+	Localiza uma ou mais ocorrências do padrão
?	Localiza zero ou uma ocorrência do padrão
{ <i>n</i> }	Localiza exatamente <i>n</i> ocorrências do padrão
{ <i>n</i> ,}	Localiza <i>n</i> ou mais ocorrências do padrão
{ <i>n</i> , <i>m</i> }	Localiza entre <i>n</i> e <i>m</i> (inclusive) ocorrências

Quantificadores

```
21 public static void main(String[] args) {
22
23     String str1 = ".0.0-";
24     String str2 = "0.0.0-";
25     String str3 = "000.070.007-25";
26     String str4 = "07.00.07-0";
27     String str5 = "0704570.004540970.07000-01052";
28
29     String padrao1 = "\\d+\\.\\d+\\.\\d+";
30     String padrao2 = "\\d+\\.\\d+\\.\\d+";
31     String padrao3 = "\\d{3}\\d{3}\\d{3}\\d{2}";
32     String padrao4 = "\\d{2,3}\\d{2,3}\\d{2,3}\\d{1,2}";
33     String padrao5 = "\\d{3,}\\d{3,}\\d{3,}\\d{2,}";
34
35     System.out.println("padrao1\\tpadrao2\\tpadrao3\\tpadrao4\\tpadrao5");
36     System.out.println(str1 + str1.matches(padrao1) + str1.matches(padrao2) + str1.matches(padrao3) + str1.matches(padrao4) + str1.matches(padrao5));
37     System.out.println(str2 + str2.matches(padrao1) + str2.matches(padrao2) + str2.matches(padrao3) + str2.matches(padrao4) + str2.matches(padrao5));
38     System.out.println(str3 + str3.matches(padrao1) + str3.matches(padrao2) + str3.matches(padrao3) + str3.matches(padrao4) + str3.matches(padrao5));
39     System.out.println(str4 + str4.matches(padrao1) + str4.matches(padrao2) + str4.matches(padrao3) + str4.matches(padrao4) + str4.matches(padrao5));
40     System.out.println(str5 + str5.matches(padrao1) + str5.matches(padrao2) + str5.matches(padrao3) + str5.matches(padrao4) + str5.matches(padrao5));
41 }
42 }
```

Aula13.Aula13

Saída * Resultados da Pesquisa *

Aula12 (run) x Console do Depurador x Aula12 (run) 02 x Aula13 (run) x

run:

	padrao1	padrao2	padrao3	padrao4	padrao5
str1	true	false	false	false	false
str2	true	false	false	false	false
str3	true	true	true	true	true
str4	true	true	false	true	false
str5	true	true	false	false	true

Substituindo *Substrings*

- Método `replaceAll(String regex, String replacement)`: substitui os padrões casados em uma *string* pelo texto definido em `(replacement)`
- Método `replaceFirst(String regex, String replacement)`: semelhante ao método `replaceAll`, porém só irá substituir a primeira ocorrência do padrão informado pela expressão regular

Substituindo Substring e Dividindo *String* com ERs

```
10 public class Teste {  
11  
12     public static void main(String[] args) {  
13  
14         //Substituindo todos os números de uma string por "_"  
15         String st = new String("5 de janeiro de 1985");  
16         System.out.println("String antes: " + st);  
17         String newSt = st.replaceAll("\\d+", "_");  
18         System.out.println("String depois: " + newSt);  
19         String newSt2 = st.replaceAll("\\d", "_");  
20         System.out.println("String depois depois: " + newSt2);  
21     }  
22 }  
23  
24 }
```

Saída - Teste (run) x

```
run:  
String antes: 5 de janeiro de 1985  
String depois: _ de janeiro de _  
String depois depois: _ de janeiro de _
```

Substituindo Substring e Dividindo *String* com ERs

- Vale ressaltar que *strings* simples podem ser utilizadas no método `replaceAll`

```
12 public class Teste {  
13  
14     public static void main(String[] args) {  
15  
16         String st1 = new String("Programação Orientada a Objetos");  
17         st1 = st1.replaceAll("a", "e");  
18  
19         System.out.println(st1);  
20     }  
21 }  
22
```

Resultados da Pesquisa Saída x

Console do Depurador x Teste (run) x

run:
Programação Orientada e Objetos

Substituindo *Substrings* e Dividindo *String* com ERs

- Expressões regulares também podem ser utilizadas no método `split`

```
10 public class Teste {
11
12     public static void main(String[] args) {
13
14         //Separando os números em token considerando como delimitador uma sequencia de números 0
15         String st = new String("123157800012547456460000021315456789790011545648789012315478000123454879");
16         String[] partes = st.split("0+");
17         for(int i=0;i<partes.length;i++){
18             System.out.println("Parte " + (i+1) + ": " + partes[i]);
19         }
20     }
21 }
22
```

Saída - Teste (run) x

```
run:
Parte 1: 1231578
Parte 2: 1254745646
Parte 3: 2131545678979
Parte 4: 11545648789
Parte 5: 12315478
Parte 6: 123454879
```

Classes Pattern e Matcher

- Além das capacidade de expressão regular da classe String, o Java fornece outras classe no pacote `java.util.regex` que ajudam os desenvolvedores a manipular expressões regulares
- A classe `Pattern` representa uma expressão regular
- A classe `Matcher` contém tanto um padrão de expressão regular como uma `CharSequence` na qual procurar o padrão

Classes Pattern e Matcher

- `CharSequence` (pacote `java.lang`) é uma interface que permite acesso de leitura a uma sequência de caracteres
- A interface exige que os métodos `charAt`, `length`, `subSequence` e `toString` sejam declarados
- Tanto `String` como `StringBuilder` implementam a interface `CharSequence` → qualquer instância dessas classes podem ser utilizadas com a classe `Matcher`

Classes Pattern e Matcher

- Se uma expressão regular vai ser utilizada apenas uma vez, o método `static Pattern matches` pode ser utilizado
- Esse método aceita uma *string* que especifica a expressão regular e um `CharSequence` em que realiza a correspondência ou casamento
- Esse método retorna um `boolean` que indica se o objeto de pesquisa (o segundo argumento) corresponde à expressão regular

Classes Pattern e Matcher

```
13 public class Teste {  
14  
15     public static void main(String[] args) {  
16  
17         String CEP = "13660-000";  
18         String regex = "\\d{5}-\\d{3}";  
19  
20         if (Pattern.matches(regex, CEP)) {  
21             System.out.println("É um CEP");  
22         } else {  
23             System.out.println("Não é um CEP");  
24         }  
25     }  
26 }  
27 }
```

teste.Teste > main > if (Pattern.matches(regex, CEP)) else

Resultados da Pesquisa Saída x

Console do Depurador x Teste (run) x

run:
É um CEP

Classes Pattern e Matcher

- Se uma expressão regular vai ser utilizada mais de uma vez (ex: em um *loop*), é mais eficiente utilizar o método `static Pattern compile` para criar um objeto `Pattern` específico para essa expressão regular
- Esse método recebe uma `String` que representa o padrão e retorna um novo objeto `Pattern`, que então pode ser utilizado para chamar o método `matcher`
- Esse método recebe um `CharSequence` para procurar e retorna um objeto `Matcher`

Classes Pattern e Matcher

- O método `find` da classe `Matcher` tenta casar uma parte do objeto de pesquisa ao padrão de pesquisa
- Cada chamada para esse método inicia no ponto em que a última chamada terminou → com isso múltiplos casamentos podem ser realizados
- O método `lookingAt` executa da mesma maneira que o método `find`, exceto que sempre busca desde o início do objeto de pesquisa e sempre realizará o primeiro casamento se houver um

Classes Pattern e Matcher

- O método `Matcher.group` retorna a *string* do objeto de pesquisa que corresponde ao padrão de pesquisa correspondente à *string* casada pelos métodos `find` ou `lookingAt`
- **OBSERVAÇÕES:**
 - O método `matches` da classe `String`, `Pattern` ou `Matcher` retornará `true` somente se o objeto de pesquisa **INTEIRO** corresponder à expressão regular
 - Os métodos `find` e `lookingAt` (da classe `Matcher`) retornarão `true` se **UMA PARTE** do objeto de pesquisa corresponder à expressão regular

Classes Pattern e Matcher

```
11 public class Teste {
12
13     public static void main(String[] args) {
14
15         //Criando uma expressão regular para validar o CEP
16         Pattern expressaoR = Pattern.compile("[0-9]{5}-[0-9]{3}");
17
18         String teste1 = "A0125-000";
19         String teste2 = "13660-000";
20         String teste3 = "13660-000 abacate blá blá blá cabeçaõ 12587-013";
21
22         System.out.println("Imprimindo o que foi parado na string teste1:");
23         Matcher matcher = expressaoR.matcher(teste1);
24         while(matcher.find()){
25             System.out.println("- " + matcher.group());
26         }
27
28         System.out.println("Imprimindo o que foi parado na string teste2:");
29         matcher = expressaoR.matcher(teste2);
30         while(matcher.find()){
31             System.out.println("- " + matcher.group());
32         }
33
34         System.out.println("Imprimindo o que foi parado na string teste3:");
35         matcher = expressaoR.matcher(teste3);
36         while(matcher.find()){
37             System.out.println("- " + matcher.group());
38         }
39     }
40 }
```

Saída - Teste (run) x

```
run:
Imprimindo o que foi parado na string teste1:
Imprimindo o que foi parado na string teste2:
- 13660-000
Imprimindo o que foi parado na string teste3:
- 13660-000
- 12587-013
```

Grupo de Opções

- Pode-se fornecer um grupo de opções de formar que a expressão possa um casada considerando um dos possíveis itens do grupo
- Para isso vamos utilizar os () e delimitar as opções utilizando |

Grupo de Opções

```
1  import java.util.regex.Matcher;
2  import java.util.regex.Pattern;
3
4  public class Teste {
5
6      public static void main(String args[]){
7
8          String teste = "No dia 22 de abril de 1500, foi descoberto o brasil. "
9              + "Já no dia 15 de novembro de 1989, foi declarada a república. "
10             + "No dia 5 de agosto de 1985 eu nasci.";
11
12          Pattern er = Pattern.compile("\\d{1,2} de (abril|novembro) de \\d{4}");
13
14          Matcher matcher = er.matcher(teste);
15          while(matcher.find()){
16              System.out.println(matcher.group());
17          }
18      }
19  }
20
21  }
22
```

Salida x Matcher.java x

Console do Depurador x Teste (run) x

run:
22 de abril de 1500
15 de novembro de 1989

Grupo de Opções

```
1 import java.util.regex.Matcher;
2 import java.util.regex.Pattern;
3
4 public class Teste {
5
6     public static void main(String args[]){
7
8         String teste = "No dia 22 de abril de 1500, foi descoberto o brasil. "
9             + "Já no dia 15 de novembro de 1989, foi declarada a república. "
10            + "No dia 5/8/1985 eu nasci.";
11
12         Pattern er = Pattern.compile("(\\d{1,2} de \\w+ de \\d{4})|\\d{1,2}/\\d{1,2}/\\d{4}");
13
14         Matcher matcher = er.matcher(teste);
15         while(matcher.find()){
16             System.out.println(matcher.group());
17         }
18     }
19 }
20
21 }
22
```

Saída x Matcher.java x

Console do Depurador x Teste (run) x

run:
22 de abril de 1500
15 de novembro de 1989
5/8/1985

Grupo de Opções

- Pode-se utilizar grupos para agrupar padrões que possam repetir mais de um vez

```
1 public class Teste {  
2  
3     public static void main(String args[]){  
4  
5         String www1 = "www.ufms.br";  
6         String www2 = "www.lives.ufms.br";  
7         String www3 = "www.bb";  
8  
9         System.out.println(www1.matches("www\\.(\\w+\\.)+\\w{2,3}"));  
10        System.out.println(www2.matches("www\\.(\\w+\\.)+\\w{2,3}"));  
11        System.out.println(www3.matches("www\\.(\\w+\\.)+\\w{2,3}"));  
12  
13    }  
14  
15 }
```

Matcher.java x

Console do Depurador x Teste (run) x

run:
true

Quantificadores Gananciosos × Quantificadores Preguiçosos

- Todos os quantificadores são **gananciosos** → identificarão um padrão com maior número de caracteres possível que cases com a expressão regular

```
11 public class Teste {  
12  
13     public static void main(String[] args) {  
14  
15         //Criando uma expressão regular para validar o CEP  
16         Pattern expressaoR = Pattern.compile("A[a-z]+a");  
17  
18         String testel = "Assustadoramente";  
19  
20         System.out.println("Imprimindo o que foi parado na string testel:");  
21         Matcher matcher = expressaoR.matcher(testel);  
22         while(matcher.find()){  
23             System.out.println("- " + matcher.group());  
24         }  
25     }  
26 }  
27  
28
```

teste.Teste > main > testel >

Saída - Teste (run) x

```
run:  
Imprimindo o que foi parado na string testel:  
- Assustadora
```

Quantificadores Gananciosos × Quantificadores Preguiçosos

- Se qualquer um desses quantificadores for seguido por um ponto de interrogação (?), o quantificador se tornará **relutante** ou **preguiçoso**

```
11 public class Teste {  
12  
13     public static void main(String[] args) {  
14  
15         //Criando uma expressão regular para validar o CEP  
16         Pattern expressaoR = Pattern.compile("[a-z]+?a");  
17  
18         String testel = "Assustadoramente";  
19  
20         System.out.println("Imprimindo o que foi parado na string testel:");  
21         Matcher matcher = expressaoR.matcher(testel);  
22         while(matcher.find()){  
23             System.out.println("- " + matcher.group());  
24         }  
25     }  
26 }  
27  
28
```

Saída - Teste (run) x

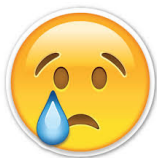
```
run:  
Imprimindo o que foi parado na string testel:  
- Assusta
```

Retrovisores

- Ao usar um grupo qualquer, o texto casado por esse grupo fica armazenado e pode ser usado em outras partes da mesma expressão regular

Sem esse tipo de recurso seria possível fazer uma expressão regular para automaticamente detectar palavras repetidas (ex: para para, lango lango, pula pula?)

Resp: Não!



Retrovisores

- Porém, podemos fazer uso dos **retrovisores**
- Como o próprio nome diz, retorvisor implica em olhar pra trás
- Para utilizar os retrovisores, utilizaremos sequências de escape que varia de 1 a 9, isto é, \1, \2, ... \9
- O retrovisor \1 armazena o conteúdo armazenado no primeiro grupo, o \2 no segundo grupo e assim por diante

Retrovisores

Exemplo de utilização de ER para padronizar textos de redes sociais

```
15 public class TesteER {
16
17     public static void main(String[] args){
18
19         String str = new String("kkkkkkk.queeeeroo.quero.queroooo.muito.muiiiiiiito.semssupervisionado.nããããã");
20
21         System.out.println("Texto original: " + str);
22
23         Pattern pER = Pattern.compile("[A-Za-zÀ-ÿ]\\1(2,");
24
25         Matcher matcher = pER.matcher(str);
26
27         System.out.println("Coisas que casaram: ");
28         while(matcher.find()){
29             String casamento = matcher.group();
30             System.out.println("- " + casamento);
31             str = str.replace(casamento, casamento.substring(0, 1));
32         }
33
34         System.out.println("Texto após os replaces: " + str);
35
36         System.out.println("Terminou");
37     }
38 }
39
```

Salda x Resultados da Pesquisa x

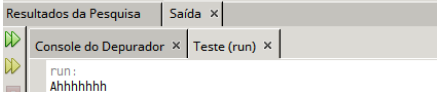
TextCategorizationTool_MultiThreading (run) x Teste (run) x

run:
Texto original: kkkkkkk.queeeeroo.quero.queroooo.muito.muiiiiiiito.semssupervisionado.nããããã
Coisas que casaram:
- kkkkkkk
- eeeee
- oooo
- iiiiiti
- ããããã
Texto após os replaces: k.quero.quero.quero.muito.muito.semssupervisionado.nã
Terminou

Retrovisores

Atentar para o uso dos parêntes quando for utilizar retrovisores

```
8 public class TesteER {  
9  
10     public static void main(String[] args) {  
11  
12         String teste = "lango lango";  
13  
14         if(teste.matches("[A-Za-z]{2,} \\1")){  
15             System.out.println("Uhu!!!");  
16         }else{  
17             System.out.println("Ahhhhhhh");  
18         }  
19     }  
20 }  
21 }
```



Retrovisores

Atentar para o uso dos parêntes quando for utilizar retrovisores

```
5 public class TesteER {  
6  
7     public static void main(String[] args) {  
8  
9         String teste = "lango lango";  
10  
11         if(teste.matches("[A-Za-z]{2,} \\1")){  
12             System.out.println("Uhu!!!!");  
13         }else{  
14             System.out.println("Ahhhhhhh");  
15         }  
16     }  
17 }  
18  
19
```

Resultados da Pesquisa | Saída ×

Console do Depurador × | Teste (run) ×

run:
Uhu!!!!



Retrovisores

Vale ressaltar que se quiser utilizar o retrovisor de uma expressão regular em uma outra expressão regular, deve-se utilizar o caractere \$ seguido do número do respectivo retrovisor

```
3 public class TesteER {
4
5     public static void main(String[] args) {
6
7         String str = "pula pula";
8         str = str.replaceAll("[A-Za-z]+ \\1", "$1");
9
10
11         System.out.println(str);
12
13     }
14 }
15
```

run:
pula

Exercício

- Atualizar o Projeto Banco para validar todas as entradas
- Criar uma classe `Validacao`, a qual conterá métodos estáticos responsáveis por realizarem a validação
 - Caso o formato dos campos sejam incorretos, esse métodos devem retornar objetos de exceção
- **Nº da conta:** os números de conta devem ser compostos por quatro caracteres numéricos, seguidos de um hífen, seguido de um caractere numérico ou uma letra maiúscula

Exercício

- **Nome de pessoas:** Primeiro e último nomes devem obrigatoriamente iniciar com letras maiúsculas, restante das letras devem ser minúsculas e só deve-se permitir o uso de caracteres alfabéticos e espaços em branco
 - Caso o usuário digite mais de um espaço em branco no meio do nome, eliminar espaços duplicados
- **Telefone:** aceitar telefones apenas nos formatos
(##)####-#### ou (##)#####-####

Exercício

- **CPF:** deve ter o formato ###.###.###-##
- **E-mail:** sequência de caracteres alfanuméricos, _ ou ., seguido de um @, seguido com uma sequência de caracteres alfanuméricos, _ ou ., seguido por um . e dois ou três caracteres alfabéticos (caracteres alfabéticos sempre em minúsculo)

Material Complementar

- Professor Isidro Explica - Episódio 2 - As Strings

https://www.youtube.com/watch?v=51piVA_EKbY

- A Classe StringBuilder em Java

<http://www.devmedia.com.br/a-classe-stringbuilder-em-java/25609>

- Conceitos básicos sobre Expressões Regulares em Java

<http://www.devmedia.com.br/>

[conceitos-basicos-sobre-expressoes-regulares-em-java/27539](#)

Material Complementar

- Diferenças entre String, StringBuilder e StringBuffer em Java
<http://www.devmedia.com.br/diferencas-entre-string-stringbuilder-e-stringbuffer-em-java/29865>
- Curso de Expressões Regulares <https://www.youtube.com/watch?v=-xbDHWjDuSM&list=PLRWVK6AfTtaevhfZZPVz36mhVmrSbcQ-->
- Java String format Example
<https://examples.javacodegeeks.com/core-java/lang/string/java-string-format-example/>

Recomendação de Leitura

Expressões Regulares - Uma Abordagem Divertida



Imagem do Dia



Programação Orientada a Objetos

<http://lives.ufms.br/moodle/>

Rafael Geraldeli Rossi
rafael.g.rossi@ufms.br

Slides baseados em [Deitel and Deitel, 2010]

Referências Bibliográficas I



Deitel, P. and Deitel, H. (2010).

Java: How to Program.

How to program series. Pearson Prentice Hall, 8th edition.