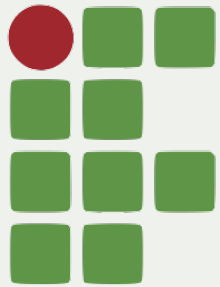
# APIs Java: Tratamento de exceções e Coleções

## POO29004 – Engenharia de Telecomunicações

Prof. Emerson Ribeiro de Mello

http://docente.ifsc.edu.br/mello/poo

18 de março de 2020



INSTITUTO FEDERAL
Santa Catarina
Câmpus São José

# Tratamento de Exceções

# Um simples programa Java

```java
import java.util.Scanner;
public class Principal{
  public static void main(String args[]){
    int[] vetor = new int[10];
    Scanner teclado = new Scanner(System.in);

    System.out.print("Entre com o número: ");
    int numero = teclado.nextInt();

    System.out.print("Em qual posição ficará?: ");
    int posicao = teclado.nextInt();

    vetor[posicao] = numero;
  }
}
```

- O código acima é seguro? Executará sempre sem problemas?



# Tratamento de Exceções

## Exceção

Evento que indica a ocorrência de algum problema durante a execução do programa



# Tratamento de Exceções

## Exceção

Evento que indica a ocorrência de algum problema durante a execução do programa

## Tratamento de exceções

Permite aos programas **capturar** e **tratar erros** em vez de deixá-los ocorrer e assim sofrer com as consequências

- Utilizado em situações em que o sistema pode recuperar-se do mau funcionamento que causou a exceção



# Em Java

- Em Java, o **tratamento de exceções** foi projetado para situações em que um método detecta um erro e é incapaz de lidar com este
- Quando um erro ocorre é criado um **objeto de exceção**
  - Contém informações sobre o erro, incluindo seu tipo e o estado do programa quando o erro ocorreu.



# Desenvolvendo códigos com tratamento de exceção

- O primeiro passo para tratar exceções é colocar todo o código que possa vir a disparar uma exceção dentro de um bloco **try...catch**
- As linhas dentro do bloco **try** são executadas sequencialmente
  - Se ocorrer uma exceção, o fluxo de execução passa automaticamente para um bloco **catch**
  - Se não ocorrer exceção, então o fluxo de execução passa para a próxima linha após os blocos **catch**

```
16 try{
17   // instruções que possam vir a disparar uma exceção
18 }catch(Tipo da excecao){
19   // instruções para lidar com a exceção gerada
20 }
21 System.out.println("continuando o programa");
```



# Exemplo 1: Tipo misturado (int *vs* String)

```
22 public static void main(String[] args){
23     Scanner ler = new Scanner(System.in);
24     int a, b;
25
26     try{
27         a = ler.nextInt();
28         b = ler.nextInt();
29
30         double res = (double) a / b;
31
32         System.out.println(a + " dividido por " + b + " = " + res);
33
34     }catch(Exception e){
35         System.err.println("Ocorreu o erro: " + e.toString());
36     }
37     System.out.println("Fim do programa");
38 }
```



# Exercício 1

- No exemplo anterior o objeto **e** da classe *Exception* tem várias informações sobre a exceção que fora gerada
- Use o depurador da IDE para verificar onde fica armazenada a informação sobre o número da linha que disparou a exceção
  - Adicione um *break point* na linha 27 da listagem do slide anterior
  - Analise os atributos do objeto **e**
- Dentro do bloco `catch` imprima esse número da linha



# Exercício 1

- No exemplo anterior o objeto **e** da classe *Exception* tem várias informações sobre a exceção que fora gerada
- Use o depurador da IDE para verificar onde fica armazenada a informação sobre o número da linha que disparou a exceção
  - Adicione um *break point* na linha 27 da listagem do slide anterior
  - Analise os atributos do objeto **e**
- Dentro do bloco `catch` imprima esse número da linha
- Agora faça modificações no primeiro exemplo dessa aula (com estouro de vetor) para imprimir o número da linha que disparou a exceção



# Determinando o tipo da exceção

- Para cada bloco **try** é possível ter um ou mais blocos **catch**
  - Cada bloco **catch** é responsável por tratar um tipo específico de exceção
- No exemplo anterior, o bloco **catch** capturava a exceção mais genérica possível em Java
  - Capturava objetos da `classe Exception`
- Em Java existem diversas outras classes para exceções, todas herdam da Exception
  - `ClassNotFoundException, ArithmeticException, FileNotFoundException, ...`



# Determinando o tipo da exceção

- Para cada bloco **try** é possível ter um ou mais blocos **catch**
  - Cada bloco **catch** é responsável por tratar um tipo específico de exceção
- No exemplo anterior, o bloco **catch** capturava a exceção mais genérica possível em Java
  - Capturava objetos da `classe Exception`
- Em Java existem diversas outras classes para exceções, todas herdam da Exception
  - `ClassNotFoundException`, `ArithmeticException`, `FileNotFoundException`, `...`

## Sequência de blocos catch

Deve-se colocar a captura de exceções específicas antes das exceções mais genéricas



# Capturando exceções específicas

```
39 public static void main(String[] args){
40     int[] numeros = new int[10];
41     Scanner teclado = new Scanner(System.in);
42   try{
43     System.out.print("Entre com o número: ");
44     int numero = teclado.nextInt();
45     System.out.print("Em qual posição ficará?: ");
46     int posicao = teclado.nextInt();
47
48     numeros[posicao] = numero;
49
50     }catch(java.util.InputMismatchException e){
51         System.err.println("Erro: Valores nao inteiros. ");
52     }catch(java.lang.ArrayIndexOutOfBoundsException e){
53         System.err.println("Erro: estouro de limite de vetor ");
54     }catch(Exception e){
55         System.err.println("Ocorreu o erro: " + e.toString());
56     }
57     System.out.println("Fim do programa");
58 }
```



# Bloco finally

- As linhas dentro do bloco **finally** sempre serão executadas, independente de ocorrer exceção ou não
- Códigos dentro dos blocos **catch** só serão executados somente se for lançada alguma exceção
- As linhas dentro do bloco **finally** sempre serão executadas, mesmo se houver instruções `return`, `continue` ou `break` dentro do bloco **try**
- O bloco **finally** é o local ideal para liberar recursos que foram adquiridos anteriormente

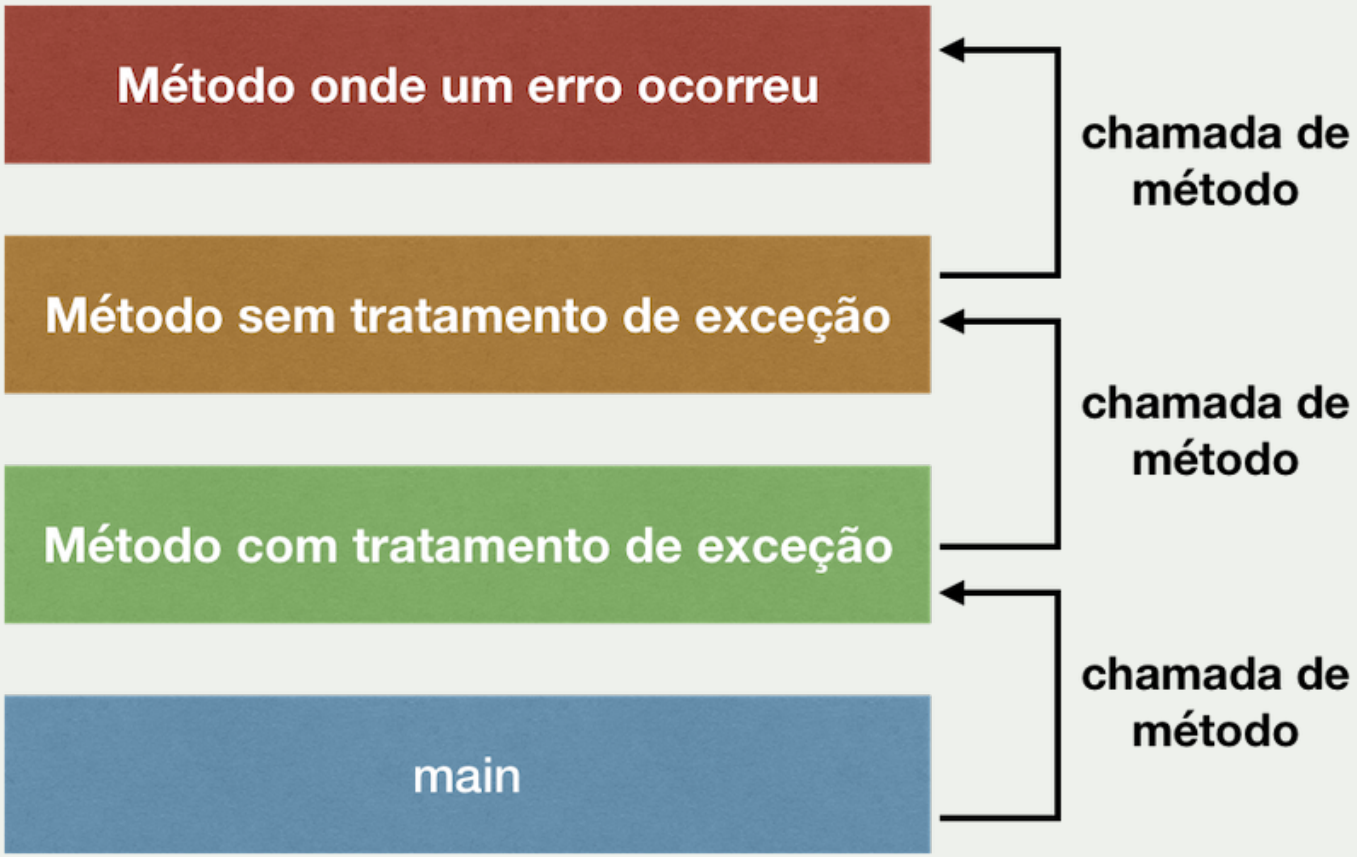


# Bloco finally – execução mesmo diante de uma instrução return

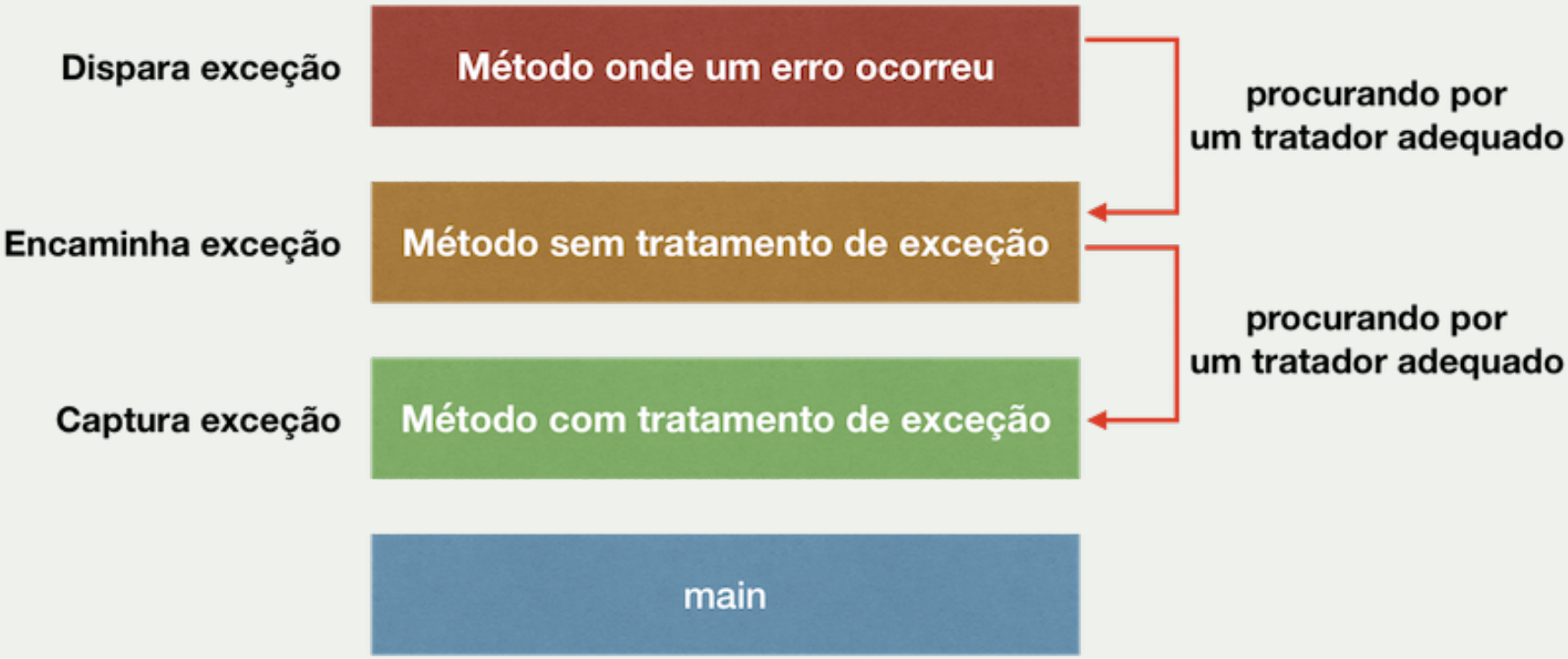
```
59 public int lerTeclado(){
60   Scanner teclado = new Scanner(System.in);
61   do{
62     try{
63       System.out.print("Entre com um número: ");
64       int num = teclado.nextInt();
65       return num;
66     }catch(Exception e){
67       System.err.println("Erro: " + e.toString());
68     }finally{
69       System.out.print("Sempre será executada");
70     }
71     System.out.print("Se o return for executado, então essa linha não
       será");
72   }while(true);
73 }
```



# Disparando e capturando exceções



# Disparando e capturando exceções



# Encaminhando e disparando exceção

- Encaminhando exceções para o método que o invocou

```
74 public void escreverArquivoNoDisco() throws IOException {
75 .....
76 }
```

- Criando um objeto de qualquer subclasse da classe `Throwable`

```
77 public Object pop() {
78     if (size == 0) {
79         throw new EmptyStackException();
80     }
81 .....
82 }
```



# Exercício

```
83 public class Exercicio02{
84   private Scanner ler = new Scanner(System.in);
85
86   public int lerNumero(){
87     System.out.print("Entre com um numero: ");
88     return ler.nextInt();
89   }
90
91   public double divisao(int a, int b){
92     return (double) a / b;
93   }
94 }
```

- Garanta que o *lerNumero* irá progredir se o usuário entrar com um *int*
- Garanta que o `divisao` dispare uma exceção para o método que o invocou se **b** for igual a zero



# Exemplo

# Classe MaskFormatter

- A classe `MaskFormatter` é usada para formatar e editar Strings
- A máscara indica quais são os caracteres válidos que podem estar contidos na String

| | |
|---|---|
| # | Qualquer número |
| U | Qualquer caractere e todos serão convertidos para maiúsculo |
| L | Qualquer caractere e todos serão convertidos para minúsculo |
| A | Qualquer caractere ou número |
| ? | Qualquer caractere |
| * | Qualquer coisa |
| H | Qualquer hexadecimal (0-9, a-f ou A-F) |

```
95 MaskFormatter mask = new MaskFormatter("(##) #####-####");
```

- O construtor dispara uma exceção do tipo `ParseException`



# Classe MaskFormatter – tratando a exceção

```
96  public String formata(String mascara, String valor){
97    MaskFormatter mask = null;
98    String resultado = "";
99    try {
100     mask = new MaskFormatter(mascara);
101     mask.setValueContainsLiteralCharacters(false);
102     mask.setPlaceholderCharacter('_');
103     resultado = mask.valueToString(valor);
104   } catch (ParseException e) {
105     e.printStackTrace();
106   }
107   return resultado;
108 }
109 public static void main(String[] args) {
110   Principal p = new Principal();
111   System.out.println(p.formata("(##) #####-####", "48998765432"));
112 }
```



# Classe MaskFormatter – encaminhando a exceção

```
113 public String formata(String m, String v) throws ParseException {
114   MaskFormatter mask = null;
115   String resultado = "";
116   mask = new MaskFormatter(m);
117   mask.setValueContainsLiteralCharacters(false);
118   mask.setPlaceholderCharacter('_');
119   resultado = mask.valueToString(v);
120   return resultado;
121 }
122 public static void main(String[] args) {
123   Principal p = new Principal();
124   try {
125     System.out.println(p.formata("(##) #####-####", "48998765432"));
126   } catch (ParseException e) {
127     e.printStackTrace();
128   }
129 }
```



# APIs Java: Coleções

# Coleção

- Em Java coleção é um objeto que agrupa múltiplos elementos dentro de uma única unidade
  - Usadas para armazenar, obter e manipular dados agregados
- Representam itens que formam um grupo natural
  - Baralho, pasta de e-mails, catálogo telefônico
- O *Java Collections Framework* provê também algoritmos para busca e ordenação em coleções



- **Set**
  - Coleção que não permite elementos duplicados
- **List**
  - Coleção ordenada de elementos e permite elementos duplicados
- **Queue**
  - Fila que ordena elementos para serem processados posteriormente, por exemplo, FIFO
- **Map**
  - Mapeia chaves para valores. Não permite chaves duplicadas e cada chave pode mapear somente um valor



# Lista

- **ArrayList** – Armazena elementos em um vetor, cujo tamanho aumenta automaticamente
- **LinkedList** – Armazena elementos em uma lista duplamente encadeada

```
130 List<String> ll = new LinkedList<>();
131 ArrayList<String> col = new ArrayList<>();
132
133 ll.add("Tele");
134 ll.add("IFSC");
135 col.add("Tele");
136
137 String[] vet = ll.toArray();
138
139 String nome = ll.get(1); // obtém elemento na posição 1
140
141 Collections.sort(ll); // ordena todos elementos da lista
142
143 ll.clear(); // remove todos elementos da lista
```

# ArrayList – exemplos

```
144 ArrayList<String> lista = new ArrayList<>();
145 int n = lista.size(); // obtendo o total de elementos na coleção
146
147 lista.add("IFSC");
148 lista.add("Telecomunicações");
149
150 String s = lista.toString(); // [IFSC, Telecomunicações]
151
152 lista.remove("IFSC"); // removendo elemento
153
154 boolean b = lista.contains("IFSC"); // verificando se existe elemento
```



# Percorrendo uma ArrayList

```
155 // Percorrendo e alterando um elemento específico
156 for(String elemento: lista){
157   if (elemento.equals("Telecomunicações")){
158     int posicao = lista.indexOf(elemento);
159     lista.set(posicao, "Engenharia de Telecomunicações");
160   }
161 }
162 //Iterando uma coleção
163 for(String elemento: lista){
164   System.out.println(elemento);
165 }
166
167 //Percorrendo com lambda
168 lista.forEach(elemento->System.out.println(elemento));
169
170 //Percorrendo com method reference
171 lista.forEach(System.out::println);
```



# Percorrendo uma ArrayList

```
172 // Percorrendo e alterando um elemento específico
173 for(String elemento: lista){
174   if (elemento.equals("Telecomunicações")){
175     int posicao = lista.indexOf(elemento);
176     lista.set(posicao, "Engenharia de Telecomunicações");
177   }
178 }
179
180 // Fazendo uso de lambda para o mesmo comportamento feito acima
181 lista.forEach(elemento->{
182   if (elemento.equals("Telecomunicações")){
183     int posicao = lista.indexOf(elemento);
184     lista.set(posicao, "Engenharia de Telecomunicações");
185   }
186 });
```



# Map

```
187 Map<String, String> cores = new HashMap<>();
188 // Adicionando elementos no HashMap
189 cores.put("Vermelho","FF0000");
190 cores.put("Verde","00FF00");
191 cores.put("Azul","0000FF");
192
193 // obter valor associado a chave Azul
194 String azul = cores.get("Azul");
195
196 StringBuilder sb = new StringBuilder();
197
198 // percorrendo todos elementos e montando uma StringBuilder
199 cores.forEach((chave, valor)->{
200   sb.append(chave+":"+valor+"\n");
201 });
202
203 // imprimindo o conteúdo da StringBuilder
204 System.out.println(sb.toString());
```



# Percorrendo um HashMap

```
205 Map<String, String> cores = new HashMap<>();
206 cores.put("Vermelho","FF0000");
207 cores.put("Verde","00FF00");
208 cores.put("Azul","0000FF");
209
210 // Percorrendo um HashMap
211 for(Map.Entry<String,String> elemento: cores.entrySet()){
212   System.out.println(elemento.getKey() + ":" + elemento.getValue());
213 }
214
215 // Percorrendo um HashMap usando lambda
216 cores.forEach((chave, valor)->{
217   System.out.println(chave + ":" + valor);
218 });
219
220 // Exatamente igual a instrução acima
221 cores.forEach((chave, valor)->System.out.println(chave + ":" + valor));
```



# Leitura obrigatória

- https://docs.oracle.com/javase/tutorial/essential/exceptions
- https://docs.oracle.com/javase/tutorial/collections/
- https://docs.oracle.com/javase/tutorial/java/data/buffers.html
- https://docs.oracle.com/javase/8/docs/technotes/guides/language/foreach.html
- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- https://blog.caelum.com.br/java-8-lambda-ou-method-reference-entenda-a-diferenca/
- https://docs.oracle.com/javase/8/docs/api/javax/swing/text/MaskFormatter.html