Tratamento de exceções e Threads em Java

Programação Orientada a Objetos

Prof. Emerson Ribeiro de Mello

Instituto Federal de Santa Catarina – IFSC
campus São José
mello@ifsc.edu.br

12 de maio de 2014

Conteúdo programático

1. Tratamento de Exceções

2. Threads – Programação Concorrente

Conteúdo programático

1. Tratamento de Exceções
2. Threads – Programação Concorrente

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

- Deve ser 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
  - Não é possível garantir que existirá algum trecho para tratar a exceção disparada
- 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

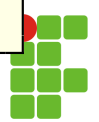
- O primeiro passo para tratar exceções é colocar todo o código que possa vir a disparar uma exceção dentro de um bloco **try...catch**

```
1  try{
2        instrucoes que possam vir a disparar uma excecao;
3  }catch(Tipo da excecao){
4        instrucoes para lidar com a excecao gerada
5  }
6  System.out.println("continuando o programa");
```

- 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**

Exemplo 1: Divisão por zero

```
7  public static void main(String[] args){
8     Scanner ler = new Scanner(System.in);
9     int a, b, res;
10
11    try{
12       a = ler.nextInt();
13       b = ler.nextInt();
14
15       res = a / b;
16
17       System.out.println(a + " dividido por " + b + " = " + res);
18
19    }catch(Exception e){
20       System.err.println("Ocorreu o erro: " + e.toString());
21    }
22    System.out.println("Fim do programa");
23 }
```



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

```
24 public static void main(String[] args){
25    Scanner ler = new Scanner(System.in);
26    int a, b, res;
27    try{
28       a = ler.nextInt();
29       b = ler.nextInt();
30
31       res = a / b;
32
33       System.out.println(a + " dividido por " + b + " = " + res);
34
35    }catch(java.util.InputMismatchException e){
36       System.out.println("Erro: Valores nao inteiros. ");
37    }catch(java.lang.ArithmeticException e){
38       System.out.println("Erro: Divisao por zero ");
39    }catch(Exception e){
40       System.out.println("Ocorreu o erro: " + e.toString());
41    }
42    System.out.println("Fim do programa");
43 }
```

Bloco finally

- As linhas dentro do bloco **finally** sempre serão executadas, independente de ocorrer exceção ou não
  - O bloco **finally** é o local ideal para colocar o código que liberará os recursos que foram adquiridos em um bloco **try**

Exemplo

Um arquivo é aberto dentro do bloco **try** e o local para fechar este arquivo é dentro do bloco **finally**, pois independente de ocorrer ou não uma exceção após a abertura do arquivo dentro do bloco **try**, este arquivo sempre será fechado.

Bloco finally

```
44        System.out.println("Ola mundo");
45  try{
46        System.out.println("Primeira instrucao");
47        int a = 10 / 0 ;
48        System.out.println("Terceira instrucao");
49
50  }catch(Exception e){
51        System.out.println("Executada somente se ocorrer excecao");
52
53  }finally{
54        System.out.println("Executa sempre");
55  }
56  System.out.println("Executa sempre - fora do bloco");
```

Bloco finally

```
44        System.out.println("Ola mundo");
45  try{
46        System.out.println("Primeira instrucao");
47        int a = 10 / 0 ;
48        System.out.println("Terceira instrucao");
49
50  }catch(Exception e){
51        System.out.println("Executada somente se ocorrer excecao");
52
53  }finally{
54        System.out.println("Executa sempre");
55  }
56  System.out.println("Executa sempre - fora do bloco");
```

Leitura recomendada

HOSRTMANN, C. S., CORNELL, G. **Core Java** - 8 ª Edição, 2010. Capítulo 11.

Conteúdo programático

1. Tratamento de Exceções

2. Threads – Programação Concorrente

Programação concorrente

- Sistemas operacionais modernos são caracterizados como **multitarefas**
  - Executam diversos processos simultaneamente

Programação concorrente

- Sistemas operacionais modernos são caracterizados como **multitarefas**
  - Executam diversos processos simultaneamente

Comunicação entre processos

Cada processo possui suas próprias variáveis e a troca de informações entre processos é feita através de arquivos em disco ou através de *sockets* de rede

Programação concorrente

- **O uso de threads permite que aplicativo que realize diversas tarefas de forma concorrente**
  - Ex: Uma *thread* fica responsável por interagir com o usuário (leitura de teclas) e outra *thread* fica responsável por escrever em um *sockets* de rede
- Por estarem dentro de um mesmo processo, compartilhando variáveis, a comunicação entre *threads* tende a ser mais eficiente e mais fácil de programar

Programação concorrente

- **O uso de threads permite que aplicativo que realize diversas tarefas de forma concorrente**
  - Ex: Uma *thread* fica responsável por interagir com o usuário (leitura de teclas) e outra *thread* fica responsável por escrever em um *sockets* de rede
- Por estarem dentro de um mesmo processo, compartilhando variáveis, a comunicação entre *threads* tende a ser mais eficiente e mais fácil de programar

Laboratório 0: Programa com uma única Thread

Execute o programa Principal.java do Laboratório **thread-ex0**

Aplicação com uma única Thread (linha de execução)

```
57 public class Fluxo1{
58    public void disparar(){
59       for(int i=0; i<1000;i++){
60          System.err.println("Fluxo 1");
61       }
62    }
63 }
64 public class Fluxo2{
65    public void disparar(){
66       for(int i=0; i<1000;i++){
67          System.err.println("Fluxo 2");
68       }
69    }
70 }
71 public class Principal{
72    public static void main(String[] args){
73       Fluxo1 f1 = new Fluxo1(); Fluxo2 f2 = new Fluxo2();
74       f1.disparar();
75       f2.disparar();
76       System.err.println("Fim do programa");}
77 }
```

Desenvolvendo aplicação multithread em Java

- Para desenvolver uma aplicação *multithread* é necessário
  1. Escrever o código que será executado pela *Thread*
  2. Escrever o código que irá disparar a *Thread*

Desenvolvendo aplicação multithread em Java

- Para desenvolver uma aplicação *multithread* é necessário
  1. Escrever o código que será executado pela *Thread*
  2. Escrever o código que irá disparar a *Thread*

Em Java é possível criar uma Thread de duas formas

1. Criar uma classe que estenda a classe **Thread**
   - Deve-se sobrescrever o método `public void run()`
2. Criar uma classe que implemente a interface **Runnable**
   - Deve-se implementar o método `public void run()`
   - Opção interessante já que Java não possui o conceito de herança múltipla

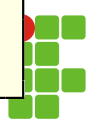
Threads em Java

- Herança

```
79 public class Fluxo1 extends Thread{
80    public void run(){
81       for(int i=0; i<1000;i++){
82          System.err.println("Fazendo uso de heranca");
83       }
84    }
85 }
```

- Interface

```
86 public class Fluxo2 implements Runnable{
87    public void run(){
88       for(int i=0; i<1000;i++){
89          System.err.println("Fazendo uso de interface");
90       }
91    }
92 }
```



Exemplo de uso com herança e interface

```
71 public static void main(String[] args){
72    Thread comHeranca = new Fluxo1();
73    Thread comInterface = new Thread(new Fluxo2());
74
75    //executando as threads
76    comHeranca.start();
77    comInterface.start();
78
79    System.err.println("Fim do programa");
80 }
```

Exemplo de uso com herança e interface

```
71 public static void main(String[] args){
72    Thread comHeranca = new Fluxo1();
73    Thread comInterface = new Thread(new Fluxo2());
74
75    //executando as threads
76    comHeranca.start();
77    comInterface.start();
78
79    System.err.println("Fim do programa");
80 }
```

Qual a será a saída do programa acima? Veja o lab. **thread-ex1**

1. 1000 linhas com herança + 1000 linhas com interface + Fim do programa
2. Fim do programa + 1000 linhas com herança + 1000 linhas com interface
3. Não tenho como prever

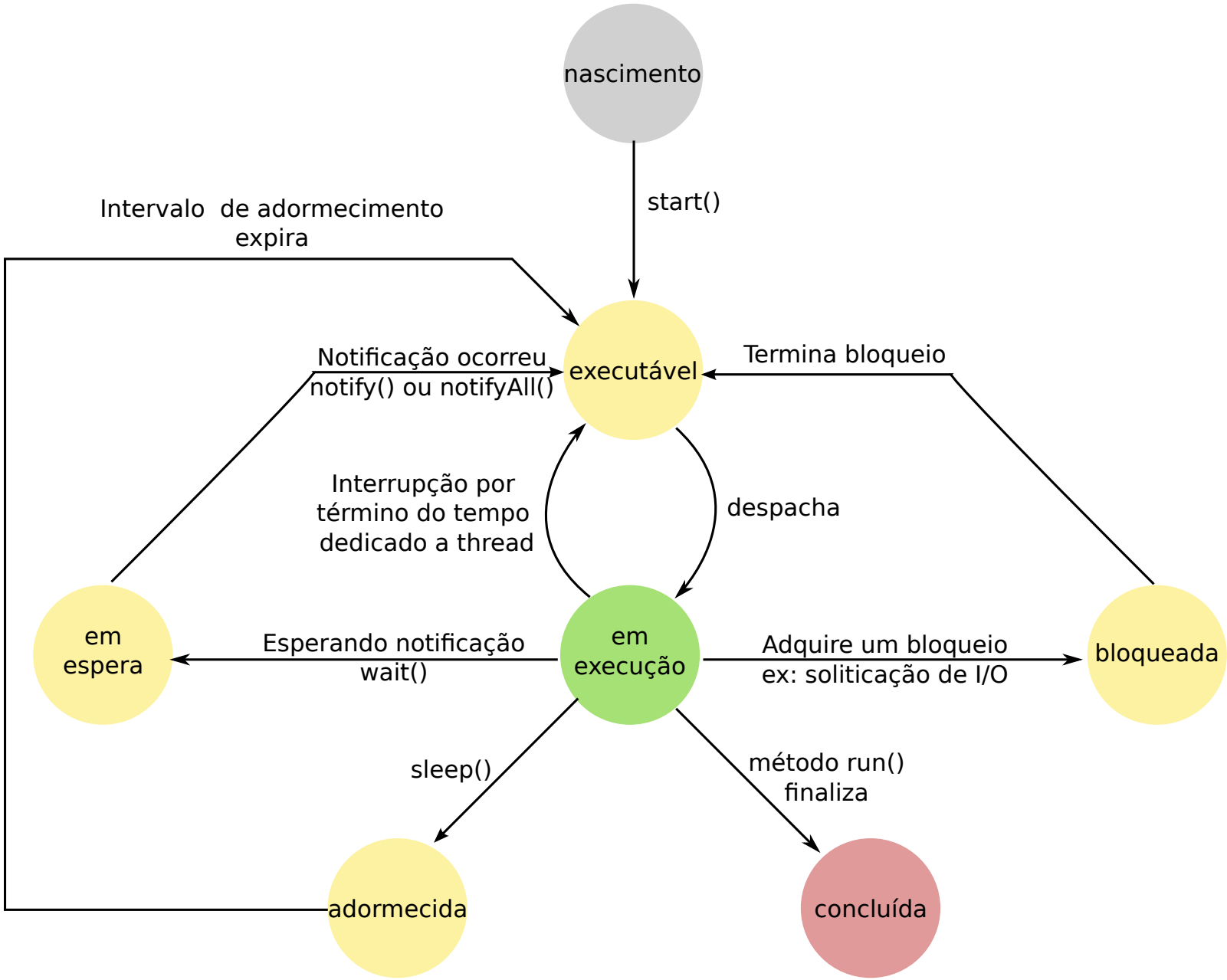
Exemplo de uso com herança e interface

```
71 public static void main(String[] args){
72    Thread comHeranca = new Fluxo1();
73    Thread comInterface = new Thread(new Fluxo2());
74
75    //executando as threads
76    comHeranca.start();
77    comInterface.start();
78
79    System.err.println("Fim do programa");
80 }
```

Qual a será a saída do programa acima? Veja o lab. **thread-ex1**

1. 1000 linhas com herança + 1000 linhas com interface + Fim do programa
2. Fim do programa + 1000 linhas com herança + 1000 linhas com interface
3. Não tenho como prever

Ciclo de vida de uma thread

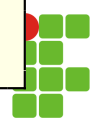


Principais métodos para trabalhar com threads

start	Ocorre a invocação do método **run** da Thread. • Após disparar a *thread*, o fluxo de execução retorna para o seu chamador imediatamente
run	Onde é colocada a lógica do fluxo • Ao finalizar este método, a *thread* morre
sleep	Faz com que a *thread* durma por alguns milisegundos • **Importante**: Enquanto uma *thread* dorme, ela não disputa o processador • Exemplo de uso dentro do método **run**: `Thread.sleep( 1000 );`
join	Espera que a *thread* que fora invocada morra antes de retornar para a *thread* que a invocou

Fazendo a thread dormir por 1000 milisegundos

```
81 public class Fluxo3 extends Thread{
82
83    public Fluxo3(String nome){
84       super(nome);
85    }
86
87    public void run(){
88       try{
89
90          System.err.println( this.getName() + " vair dormir...");
91          Thread.sleep(1000);
92
93       }catch(InterruptedException e){
94          System.err.println(e.toString());
95       }
96       System.err.println(this.getName() + " acordou...");
97    }
98 }
```



Exemplo de uso do método join

```
99  public static void main(String[] args){
100    Thread f3 = new Fluxo3();
101
102    //disparando a thread
103    f3.start();
104    System.err.println("Depois do start e antes do join");
105    try{
106
107       f3.join();
108       // a linha abaixo e' executada somente depois
109       // de finalizar o metodo run do objeo f3
110       System.err.println("Depois do join");
111
112    }catch(InterruptedException ex) {
113       System.err.println(ex.toString());
114    }
115
116    System.err.println("Fim do programa");
117 }
```

Concorrência e sincronismo entre threads

O uso de memória compartilhada entre as *threads* requer a sincronização das ações que serão executadas sobre essa memória. Ou seja, somente uma thread por vez pode acessar essa memória compartilhada

- Java implementa o conceito de `monitor` para impor o acesso mutuamente exclusivo aos métodos
  - Tais métodos devem apresentar a palavra **synchronized**
- Quando um método sincronizado é executado o monitor é consultado
  - Se não existir outro método sincronizado em execução, então continua; Senão, aguarde pela notificação
- Métodos para trabalhar com sincronismo:
  - **wait**, **notify** e **notifyAll**;
- **Nota:** Veja exemplo da barbearia na página da disciplina e no livro Java Como Programar