UFMS DIGITAL

Programação Orientada a Objetos

Prof. Dr. Anderson V. de Araujo

UFMS DIGITAL

Módulo IV - Conceitos avançados

Unidade III - Tratamento de Exceções

Contextualização

- O tratamento de erros é uma característica fundamental, pois é praticamente impossível criar uma aplicação totalmente livre de erros;
- Erros que não necessariamente são ocasionados por falhas do desenvolvimento:
  - Porém, se não tratados, os erros oriundos daquelas situações previstas, devem ser consideradas falhas no desenvolvimento de software.
- Com isso, faz-se necessário “preparar alguma resposta” nestas situações.

Objetivos do Tratamento de Exceções

- Detecção mais fácil de erros sem a escrita de um código especial para testar valores retornados;
- Manter um código de manipulação de exceções nitidamente separado do código que gerará a exceção;
- Lidar com as diferentes exceções possíveis usando o mesmo código de manipulação de exceções.

Exemplos

- Em uma aplicação que se comunica com o banco de dados, este pode estar fora do ar em determinada situação;
- Um arquivo de configuração ter sido acidentalmente removido;
- O usuário pode ter digitado um valor inválido;
- Retorno de uma pilha vazia de inteiros.

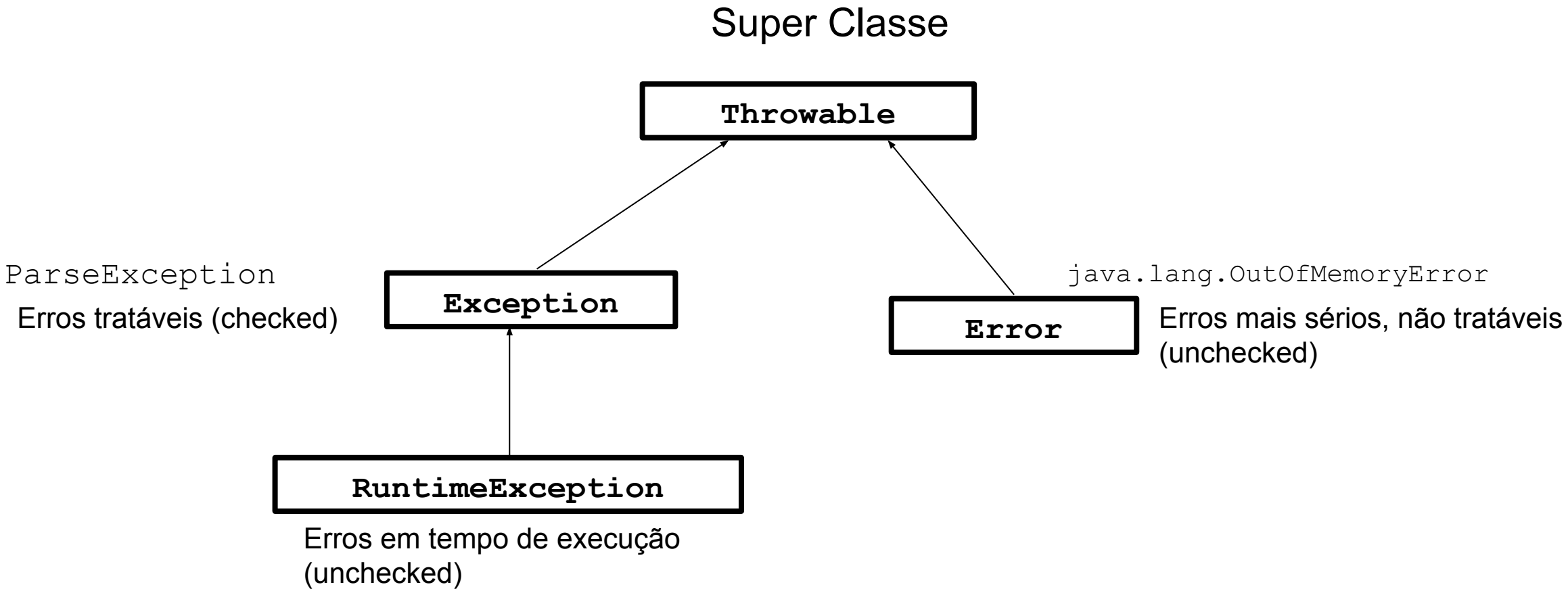
Ações a Serem Tomadas

- Por exemplo, se o BD estiver fora do ar, o que fazer?
  - A mensagem de erro deve ser impressa no console?
  - A mensagem de erro deve ser impressa em uma área visível pelo usuário (interface gráfica, aplicação web ...)?
  - Um e-mail deve ser enviado ao administrador do Banco de Dados?
  - A operação deve ser logada em um arquivo?
  - Os dados do usuário devem ser salvos em um arquivo temporário?
  - Todos os anteriores?

Java Exceptions

- Exceptions são classes comuns em Java e devem tratar os erros passíveis de ocorrer;
- Existem muitos tipos nativos da linguagem tratando um conjunto específico de problemas, por exemplo:
  - `SQLException, IOException, NullPointerException, NumberFormatException, ArrayIndexOutOfBoundsException…`
- Java possui dois tipos de exceções: *Unchecked* e *Checked*
  - A "checagem" significa se a exceção é verificada em tempo de compilação ou não.
    - Em C ++, todas as exceções são unchecked, portanto, o compilador não é forçado a manipular ou especificar a exceção

Hierarquia das Exceptions



Unchecked Exceptions

- Essas exceções ocorrem devido à programação incorreta;
- O programa não dará um erro de compilação;
- Não precisam ser declaradas, nem tratadas ou lançadas;
- Todas são filhas de *RuntimeException*;
- Exemplos:
  - `NullPointerException`
  - `ArrayIndexOutOfBound`
  - `IllegalArgumentException`
  - `ArithmeticException`

Unchecked Exceptions

- Para criarmos uma classe que modela uma *unchecked*; exception, devemos estender a classe *Error* ou *RuntimeException;*
- Tipicamente não criamos exceções desse tipo, elas são usadas pela própria linguagem para indicar condições de erro.

```
public static void main(String args[]) {
    String arr[] = { "A", "B", "C", "D", "E" };
    String myString = arr[7];//this throws
ArrayIndexOutOfBoundException
    System.out.println(myString);
}
```

Checked Exceptions

- Todas as exceções que não são filhas da classe *RuntimeException*;
- Neste tipo, o desenvolvedor é obrigado a realizar alguma operação no caso de erro:
  - Assim, o método deve manipular a exceção (`try-catch`) ou lançar a exceção para o método chamador na pilha usando `throws`.
- É verificado em tempo de compilação. Ou seja, não compila se não for tratada.

Exemplo Checked Exception

```
public static void main(String args[]) {
    FileReader file = new FileReader("C:\\test\\a.txt");
    BufferedReader fileInput = new BufferedReader(file);
    // Print first 3 lines of file "C:\test\a.txt"
    for (int counter = 0; counter < 3; counter++)
      System.out.println(fileInput.readLine());

    fileInput.close();
}
```

- A criação do `FileReader` "pede" o tratamento da exceção, pois o seu construtor (código chamado) lança uma exceção por meio do `throws`

```
public FileReader(String fileName) throws FileNotFoundException{
   super(new FileInputStream(fileName));
}
```

Sintaxe try-catch

- `try`: é usada para indicar/cercar um bloco de código que possa ocorrer uma exceção;
- `catch`: serve para manipular as exceções, ou seja, tratar o erro propriamente;
- `finally`: O importante é saber que esse bloco sempre será executado, útil para liberação de recursos, registros de logs, etc.

Exemplo

```
public static void main(String[] args) {
        int num = 0, den = 0;
        BufferedReader teclado = new BufferedReader(new
InputStreamReader(System.in));
        System.out.println("Digite dois números seguidos da tecla ENTER:");
        try {
                num = Integer.parseInt (teclado.readLine());
                den = Integer.parseInt(teclado.readLine());
                System.out.println (num+"/"+den+" = "+ (num/den));
        }
        catch (NumberFormatException e){
                System.out.println ("Erro de formato.");
        }
        catch (IOException e) {
                System.out.println ("Erro de E/S."); }
                        finally {
                                System.out.println("Saindo do Bloco \"Try\".");}

                        System.out.println("Fim do programa");
        }
}
```

Trecho de código tratado

Tratamento explícito da exceção

Trecho que será sempre executado

A Cláusula throws

- Os erros que devem ser tratados são lançados em métodos:
  - É necessário sinalizar esta possibilidade, para que o desenvolvedor tenha o conhecimento que deve tratá-los corretamente no método chamador.
- A indicação que estas falhas podem acontecer são definidas pela existência da cláusula `throws;`
- Use quando o erro não for de responsabilidade da classe;
  - Ela simplesmente será executada e lançará qualquer erro para ser mais adequadamente tratado em uma classe própria para isso.

Lançando Exceções

- O `throws` é usado da forma descrita a seguir para listar as exceções que um método pode lançar:

```
public class EntendendoExceptions {
    void myFunction() throws MyException1, MyException2 {
        // Código do método
    } //myFunction
} //EntendendoExceptions
```

Pode lançar mais de um tipo de exceção

Repassando Exceção

- Se quisermos usar o método div sem tratar a exceção, deve-se declarar que a exceção deve ser passada adiante:

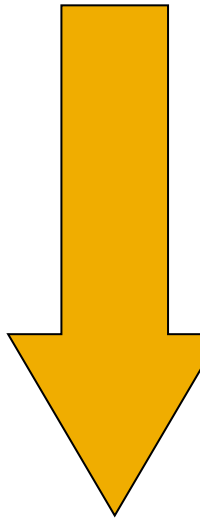
```
public void f() throws DivByZeroException {
      Calc calc = new Calc();
      int div = calc.div(a,b);
      System.out.println(div);
}
```

O método div desenvolvido lança uma exceção e deve ser tratada!

Tratando Múltiplas Exceções

- Devemos declarar primeiramente as Exceptions mais específicas, ou seja, as classes filhas, e depois, as mais genéricas.

Ex:
```
try { ... }
        catch (ClasseExceptionFilha) {}
        catch (ClasseExpcetionPai) {}
        catch (ClasseExceptionAvo) {}
```



Métodos da Classe Throwable

- `printStackTrace`(): É amplamente usado, sendo bastante útil para os desenvolvedores. Ele dá uma visão detalhada do erro, indicando qual método originou a falha, assim como cada método que a chamaram. Essa informação é impressa no console
  - Muito mais útil ao programador
- `getMessage`(): Análogo ao anterior, porém não fornece detalhes da origem do erro e das chamadas dos métodos, apenas informa por meio de uma mensagem o tipo da exceção ocorrida.

Criando minhas Exceptions

- Criar a nossa própria classe de Exceção, tem como?
  - Sim, claro!
- Lembrando-se que Exception é uma classe qualquer e, portanto, filha de Object. Logo, para usá-la basta estender a classe Exception.

Minha Exceção

```
public class MyException extends Exception {

    public MyException (String msg) {

        super(msg); //informa a msg para getMessage()

        metodoPersonalizado();

    }

    public void metodoPersonalizado() {
        System.out.println("Trate sua exceção como quiser");

        System.out.println(getMessage());

    }

}
```

Licenciamento