Tratativas de exceções com Java

Professor Valdemar Lorenzon Junior

O que são?

- São mecanismos utilizados em programação para lidar com situações inesperadas, validações ou desviar erros que podem ocorrer durante a execução de um programa;
- Servem para tratar e evitar interromper a execução do programa por situações de causas ou falhas;
- Estas situações que podem incluir:
  - Tentativas de acessar um arquivo inexistente;
  - Divisões por zero;
  - Erros de rede;
  - Entradas não previstas de usuários – validações não if-else;
  - Erros inesperados, não previstos, não tratados!
    - Podendo criar um log destes para avaliações.

Componentes básicos de TE

- **Try** (Tentar):
  - Bloco de código onde o programador espera que possa ocorrer uma exceção. Qualquer exceção lançada dentro deste bloco será tratada pelos blocos catch;
    - `try { ... //código`

      `}`
- **Catch** (Pegar):
  - Bloco de código que é executado quando uma exceção é lançada no bloco try. Esse bloco permite ao programador definir como tratar o erro específico, 1 a N exceções!
    - `catch (Classe objeto de exceção) { ... //Tratativa da ocorrência`

      `}`
- **Finally** (Finalmente, de qualquer forma):
  - Bloco opcional e se existir é bloco de código que é executado sempre, independentemente se uma exceção foi lançada ou não. Esse bloco é útil para liberar recursos, como fechar arquivos ou conexões de rede.

Tipos de exceções

- **Checked Exceptions** (Exceções Verificadas):
  - Checked exceptions são exceções que são verificadas pelo compilador em tempo de compilação.
    - Isso significa que o desenvolvedor é obrigado a tratar essas exceções:
      - Seja com um bloco try-catch;
      - Seja declarando a exceção no método usando a palavra-chave throws Classe;
    - Exemplos: IOException, SQLException, ClassNotFoundException
- **Unchecked Exceptions** (Exceções Não Verificadas):
  - Unchecked exceptions são exceções que não são verificadas pelo compilador em tempo de compilação.
    - Elas são subclasses de RuntimeException e, portanto, não precisam ser explicitamente tratadas ou declaradas;
    - Exemplos: NullPointerException, ArrayIndexOutOfBoundsException, IllegalArgumentException.

Blocos try-catch : Sintaxe

```
try {
    // Código que pode lançar uma exceção
} catch (CasseTipoDaExcecao e) {
    // Código para tratar a exceção
} ...
```

- Você pode capturar, pegar várias exceções em “catch { }” e tratar cada caso específico.
- Vamos criar um exemplo em Java que demonstra como tratar uma exceção de divisão por zero. A exceção que é lançada quando ocorre uma divisão por zero em Java é ArithmeticException:

```
public class DivisaoPorZeroExemplo {
    public static void main(String[] args) {

      int numerador = 10;
      int divisor = 0;

      try {
           int resultado = dividir(numerador, divisor);
           System.out.println("O resultado da divisão é: " +
           resultado);
      } catch (ArithmeticException e) {
           System.out.println("Erro: Divisão por zero não é " +
                                  permitida.");
      }
   }

   public static int dividir(int numerador, int divisor) {
      // Pode lançar ArithmeticException
      return numerador / divisor;
   }
}
```

throws

- A palavra-chave throws em uma forma Java usada para declarar que o método pode lançar uma ou mais exceções. Isso é útil para informar aos usuários do método que eles precisam tratar (handled) essas exceções, seja com um bloco try-catch ou propagando (cascata) a exceção para o método chamador.
- Você deve usar throws em um método quando:
  - O método pode lançar uma checked exception que ele não trata diretamente.
- Benefícios de Usar throws:
  - Clareza: Declara explicitamente quais exceções um método pode lançar, tornando o código mais legível e compreensível;
  - Manutenção: Facilita a manutenção e a depuração, pois fica claro quais exceções precisam ser tratadas pelos métodos chamadores;
  - Herança e encapsulamento: Permite que exceções sejam propagadas para níveis superiores da aplicação, onde podem ser tratadas de maneira mais apropriada. Você quer propagar a exceção para que o método chamador lide com ela.

```
public class DivisaoPorZeroExemplo {

    public static void main(String[] args) {
        int numerador = 10;
        int divisor = 0;

        try {
                int resultado = dividir(numerador, divisor);
                System.out.println("O resultado da divisão é: " + resultado);
         } catch (ArithmeticException e) {
                System.out.println("Erro: Divisão por zero não é permitida.");
           }
    }

    public static int dividir(int numerador, int divisor) throws
           ArithmeticException
    {
           // Não trata a exceção aqui, apenas a propaga identificando a tratar
           return numerador / divisor;
    }
}
```

Cenário de não tratamento da exceção por cascata de throws

1. Declaração da Exceção:

   O método declara que lança uma exceção;

2. Propagação da Exceção:

   O chamador do método propaga a exceção, declarando também que lança essa exceção;

3. Propagação até o main:

   Essa propagação pode continuar até alcançar o método main;

4. Execução e não Tratamento:

   Se a exceção não for tratada em nenhum ponto da cadeia de chamadas e alcançar o método main, a JVM (Java Virtual Machine) irá tratar a exceção não capturada, resultando em um término anormal do programa e a impressão do stack trace da exceção = Terminal!

É recomendado sempre tratar exceções de forma adequada para garantir que o programa possa lidar com situações de erro de maneira controlada e previsível.

Benefícios do thows

- Propagação de Exceções:
  - Permite que a exceção seja tratada em um nível superior do programa, facilitando a centralização do tratamento de erros.
- Claridade:
  - Especifica explicitamente que o método pode lançar uma exceção, tornando o contrato do método mais claro para outros desenvolvedores.
- Flexibilidade:
  - Permite que os métodos chamadores decidam como tratar a exceção, seja com um bloco try-catch a propagando.

```
/*Você pode ter vários blocos catch para capturar
diferentes tipos de exceções que podem ser lançadas pelo
código dentro do bloco try.*/

public class ExemploMultiplosCatch {
    public static void main(String[] args) {
        try {
            int[] numeros = new int[5];
            //Lança uma ArrayIndexOutOfBoundsException
            numeros[10] = 50;

        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Erro: Índice do array fora"
                               + " dos limites.");
        } catch (ArithmeticException e) {
            System.out.println
            ("Erro: Divisão por zero não é permitida.");
        }
    }
}
```

Finally

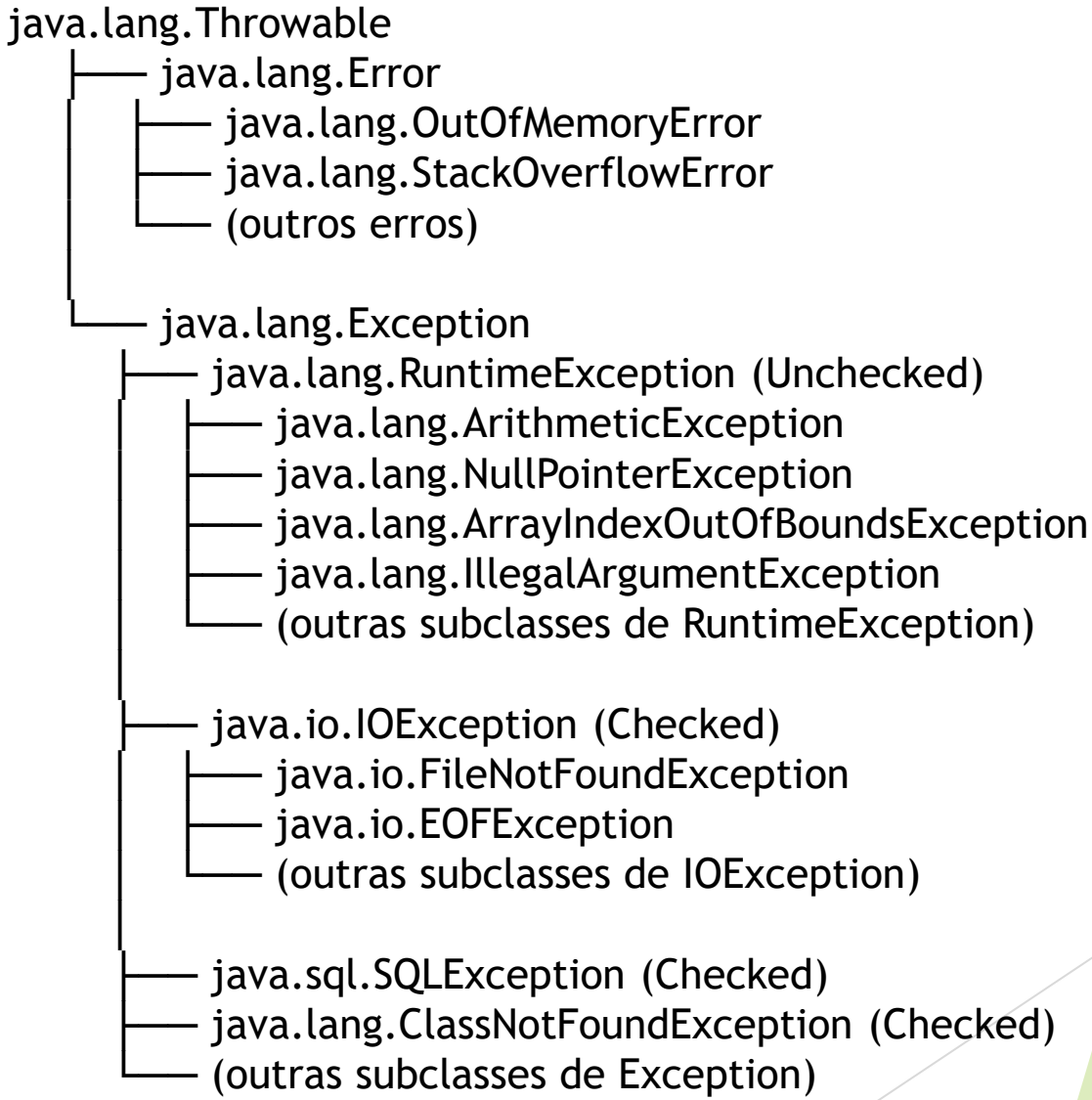
- O bloco finally é opcional e, se presente, é executado sempre, independentemente de uma exceção ter sido lançada ou não;
- É uma opção para fazer “acertos”, liberar recursos, etc...

```
public class ExemploFinally {
    public static void main(String[] args) {
        try {
            int resultado = 10 / 0;
                  // Este código lança uma ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println
                        ("Erro: Divisão por zero não é permitida.");
            System.out.println ( e.getMessage ( ) );
        } finally {
          System.out.println("Este bloco é sempre executado.");
        }
    }
}
```

A hierarquia de exceções Java

- A hierarquia de exceções em Java começa com a classe Throwable e se ramifica em duas subclasses principais: Error e Exception.

- Dentro de Exception, temos tanto exceções verificadas (checked exceptions) quanto exceções não verificadas (unchecked exceptions), que são subclasses de RuntimeException.



- Usar throws em um método é uma prática essencial para o tratamento adequado de exceções em Java.
- Ele permite que você escreva código mais robusto, mantendo a clareza sobre quais exceções precisam ser tratadas e garantindo que os métodos chamadores estejam cientes das possíveis exceções que podem ocorrer

```
import java.util.Scanner;
public class ExemploThrowsComEntradaDeUsuario {

 public static int dividir(int numerador, int divisor) throws
                                ArithmeticException {

        return numerador / divisor; // Pode lançar
                                ArithmeticException
 }
 public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {

            System.out.print("Digite o numerador: ");
            int numerador = scanner.nextInt();

            System.out.print("Digite o divisor: ");
            int divisor = scanner.nextInt();

            int resultado = dividir(numerador, divisor);
            System.out.println
                  ("O resultado da divisão é: " + resultado);

        } catch (ArithmeticException e) {

            System.out.println
               ("Erro: Divisão por zero não é perrmitida.");

        } finally {
            scanner.close();

        }
    }
}
```

Exceções personalizadas

- Para criar exceções personalizadas em Java, você precisa criar uma classe que estende uma das classes de exceção existentes, como Exception ou RuntimeException. Aqui está um exemplo simples de como criar e usar uma exceção personalizada:

```
// Definição da classe de exceção personalizada

public class MinhaExcecao extends Exception {

    // Construtor que recebe uma mensagem de erro

    public MinhaExcecao(String mensagem) {
    // Chama o construtor de (Exception) com a mensagem
        super(mensagem);
    }
}
```

Lançar exceções

- Baseados no exemplo da “nossa” exceção, para lançar uma exceção você deve fazer como exemplo no exemplo de código a seguir:

```
public class Exemplo {
        // Método que pode lançar a exceção
        // personalizada
    public void metodoExemplo(int valor)
        throws MinhaExcecao {
        if (valor < 0) {
        // Lança a exceção personalizada
        //com uma mensagem
            throw new MinhaExcecao
                ("O valor não pode ser negativo.");
        }
    }

    public static void main(String[] args) {
        Exemplo exemplo = new Exemplo();
        try {
            exemplo.metodoExemplo(-1);
           // Chamada com um valor inválido
        } catch (MinhaExcecao e) {
            // Captura a exceção e trata-a
            System.out.println
           ("Exceção capturada: " + e.getMessage());
        }
    }
}
```

O que mais pode conter minha classe de exceção?

- Ao criar classes de exceção personalizadas em Java, você pode incluir:
  - Construtores personalizados para suportar diferentes formas de criação da exceção;
  - Atributos adicionais que forneçam mais informações sobre o erro;
  - Métodos personalizados para obter melhores informações ou manipular a exceção, lembrando que pode realizar o **@Override** de métodos já existentes;
  - Criar códigos de erro para tornar suas exceções mais informativas e úteis para tratamento de erros robusto e eficaz:
  - Serializar a exceção para ser enviada através de uma rede. Veja a interface “Serializable”;
  - Gravar logs e registrar as exceções ocorridas.

Importância do TE

- **Robustez do Programa:**
  - Programas que lidam adequadamente com exceções são mais robustos e menos propensos a falhas inesperadas.
- **Manutenção do Código:**
  - Facilita a manutenção do código, pois os pontos de falha são tratados explicitamente.
- **Melhoria da Experiência do Usuário:**
  - Permite fornecer mensagens de erro mais claras e amigáveis, ajudando o usuário a entender o que deu errado e como proceder.
- **Segurança:**
  - Em alguns casos, tratar exceções pode evitar que erros levem a problemas de segurança, como vazamento de informações sensíveis.

Considerações

O tratamento de exceções é uma prática essencial em programação, que contribui significativamente para a qualidade, confiabilidade e usabilidade do software.

Ao dominar essa técnica, os programadores podem criar aplicações mais robustas e resilientes, capazes de lidar com situações inesperadas de forma elegante e controlada.