Programação Avançada - AULA 10

Matheus Moresco
Engenharia de Software - 5º Período
2025/01

Objetivos da aula

- Compreender o conceito de exceções e sua importância na robustez do código.
- Utilizar os blocos try, catch, finally e throw para manipulação de erros.
- Criar e utilizar exceções personalizadas.
- Aplicar boas práticas no tratamento de exceções em Java.

O que são Exceções?

- Eventos inesperados que ocorrem durante a execução de um programa.
- Interrompem o fluxo normal do programa.
- Precisam ser tratadas para evitar falhas.

Exemplo:

```
int a = 10;
int b = 0;
int resultado = a / b; // ArithmeticException
```

Diferença entre erros e exceções

- Em Java, Erros e Exceções são subclasses de Throwable, mas possuem propósitos diferentes:
- Erros (**Error**)
  - Representam problemas graves que normalmente não podem ser tratados pelo programa.
  - São gerados pelo próprio ambiente de execução da JVM.
  - Exemplos incluem falta de memória, erro de inicialização da JVM ou problemas de hardware.
  - O código geralmente não deve capturar um erro, pois são problemas irreversíveis.
- Exceções (**Exception**)
  - Representam condições anormais que podem ocorrer durante a execução do programa.
  - São problemas previsíveis e que podem ser tratados.
  - Podem ser verificadas (checked) ou não verificadas (unchecked).

Diferença entre erros e exceções

Erros

```
Run | Debug | Run main | Debug main
public static void main(String[] args) {
    // Simula um erro de recursão infinita
    // que leva a um StackOverflowError
    causarErro();
}

public static void causarErro() {
    causarErro(); // Recursão infinita
}
```

Exceções

```
try {
    int resultado = 10 / 0; // Gera ArithmeticException
} catch (ArithmeticException e) {
    System.out.println(x:"Erro: Divisão por zero!");
}
```

Hierarquia de Exceções

- java.lang.Throwable
  - java.lang.Error (Erros graves, não tratados pelo programa)
    - OutOfMemoryError
    - StackOverflowError
    - VirtualMachineError ...
  - java.lang.Exception (Exceções que podem ser tratadas)
    - Exceções Verificadas (Checked)
      - IOException
      - SQLException
      - ClassNotFoundException ...
    - Exceções Não Verificadas (Unchecked - herdam de RuntimeException)
      - NullPointerException
      - ArithmeticException
      - IndexOutOfBoundsException
      - IllegalArgumentException ...

Hierarquia de Exceções - Classe Throwable

- Superclasse de todas as exceções e erros.
- A classe Throwable fornece métodos como:
  - **getMessage**(): Retorna a mensagem da exceção.
  - **printStackTrace**(): Exibe o rastreamento do erro.
  - **getCause**(): Obtém a causa original da exceção.

Error (Problemas Críticos)

- Os erros são eventos graves que geralmente indicam falhas na JVM ou no sistema. O tratamento desses erros não é recomendado.
- Um exemplo é o StackOverflowError que representa um erro de recursão infinita, onde não podemos tratar esse problema durante a execução do programa.

Exception (Erros tratáveis)

- As exceções representam problemas previsíveis que podem ser tratados pelo código.
- Exceções Verificadas (**Checked Exceptions**)
  - Precisam ser tratadas obrigatoriamente (**try-catch** ou **throws**).
  - Usadas em operações que envolvem recursos externos (arquivos, banco de dados, rede).
  - Exemplo: IOException, SQLException, ClassNotFoundException.

- Exceções Não Verificadas (**Unchecked Exceptions**)
  - Herdam de **RuntimeException**.
  - Não precisam ser obrigatoriamente tratadas.
  - Resultam de erros de lógica do programador.
  - Exemplo: NullPointerException, ArithmeticException, IndexOutOfBoundsException.

Diferença entre Exceções Verificadas e Não Verificadas

Característica	Exceções Verificadas (Checked)	Exceções Não Verificadas (Unchecked)
Herança	Exception, mas NÃO RuntimeException	RuntimeException e subclasses
Obrigatório tratar?	Sim (try-catch ou throws)	Não obrigatório
Ocorrência	Operações externas (arquivos, BD, rede)	Erros de lógica do programa
Exemplos	IOException, SQLException, ClassNotFoundException	NullPointerException, ArithmeticException

Mecanismos de Tratamento de Exceções

Os principais mecanismos para tratarmos exceções são:

- **try** {}: Bloco onde ocorre a operação que pode gerar exceção.
- **catch** {}: Bloco onde a exceção é tratada.
- **finally** {}: Executado sempre, independente da exceção.
- **throw**: Lança uma exceção manualmente.
- **throws**: Declara exceções que um método pode lançar.

Utilizando os mecanismos

- Try: Para usarmos este mecanismo adicionamos dentro de seu contexto um código que possivelmente pode gerar uma exceção;
- Catch: Trecho de código que será executado caso alguma exceção seja disparada, captura o erro do trecho de código de dentro do try e atribui a uma variável;
- Finally: Será executado após o try e o cacth, independente se o erro ocorreu ou não.

```
try {
    int numero = Integer.parseInt(s:"abc"); // NumberFormatException
} catch (NumberFormatException e) {
    System.out.println("Erro ao converter número: " + e.getMessage());
} finally {
    System.out.println(x:"Finalizando execução...");
}
```

Utilizando os mecanismos

- **throw**: Vai disparar uma exceção de maneira manual.
- **throws**: Determina as exceções que um determinado método pode disparar durante a execução.

```
public static void validarIdade(int idade) throws IllegalArgumentException {
    if (idade < 18) {
        throw new IllegalArgumentException(s:"Idade mínima é 18 anos.");
    }
    System.out.println(x:"Idade válida!");

}
```

Criando Exceções Personalizadas

- Em um sistema Java, podemos criar exceções personalizadas, para expressar comportamentos ou definir tratamentos personalizados para os casos de uso de nosso projeto.
- Para isso, criamos classes para representarem as exceções que herdam de **Exception** ou **RuntimeException**.

```
class SaldoInsuficienteException extends Exception {
    public SaldoInsuficienteException(String mensagem) {
        super(mensagem);
    }
}
```

Criando Exceções Personalizadas

- Ao criar uma exceção personalizada, com uma classe que herda de **Exception** ou **RuntimeException**, podemos utilizar essa classe para o tratamento de exceções nos códigos do nosso projeto.

```
public void sacar(double valor) throws SaldoInsuficienteException {
    if (valor > 0 && valor <= saldo) {
        saldo -= valor;
        System.out.println("Saque de R$" + valor + " realizado com sucesso.");
    } else {
        throw new SaldoInsuficienteException(mensagem:"Saldo insuficiente para saque.");
    }
}
```

try-with-resources

- Introduzido no Java 7.
- Permite que recursos como arquivos, conexões e streams sejam fechados automaticamente ao final do bloco try.
- Recursos devem implementar a interface AutoCloseable.

- Vantagens:
  - ✓Reduz código repetitivo.
  - ✓Evita vazamento de recursos.
  - ✓Dispensa o uso do bloco finally para fechar recursos manualmente.

try-with-resources

Com try-with-resources:

```
try (
    BufferedReader br = new BufferedReader(
        new FileReader(fileName:"arquivo.txt")
    )
) {
    String linha;
    while ((linha = br.readLine()) != null) {
        System.out.println(linha);
    }
} catch (IOException e) {
    System.out.println("Erro ao ler arquivo: " + e.getMessage());
}
```

Sem try-with-resources:

```
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader(fileName:"arquivo.txt"));
    String linha;
    while ((linha = br.readLine()) != null) {
        System.out.println(linha);
    }
} catch (IOException e) {
    System.out.println("Erro ao ler arquivo: " + e.getMessage());
} finally {
    if (br != null) {
        try {
            br.close();
        } catch (IOException e) {
            System.out.println("Erro ao fechar arquivo: " + e.getMessage());
        }
    }
}
```

Boas Práticas no Tratamento de Exceções

- Evitar catch genérico (catch (Exception e)).
- Lançar exceções significativas (IllegalArgumentException, IllegalStateException).
- Não suprimir exceções silenciosamente.
- Usar finally para liberar recursos.
- Preferir try-with-resources para recursos como arquivos e conexões.

Exemplo pratico

- Na classe Carro, verificar se há combustível antes de acelerar o carro.
- Verificar capacidade do tanque antes de abastecer.
- Construir erros personalizados.

Exercícios

1. Na classe ContaBancaria:
   a) Criar uma exceção personalizada SaldoInsuficienteException.
   b) Implementar o tratamento adequado para evitar saques indevidos.