Guia Completo de Exceções em Java

xAI

Agosto 2025

Conteúdo

1 Introdução

Exceções em Java são mecanismos para lidar com erros ou situações inesperadas durante a execução de um programa. Este guia explora de forma didática o que são exceções, seus tipos, como tratá-las, boas práticas e exemplos práticos. O objetivo é capacitar o leitor a implementar um tratamento de erros robusto e eficiente em Java.

2 O que são Exceções?

Uma exceção é um evento que interrompe o fluxo normal de execução de um programa. Em Java, exceções são objetos da classe `Throwable` ou de suas subclasses: `Error` e `Exception`. Elas representam condições anormais, como divisão por zero, acesso a arquivos inexistentes ou entrada de dados inválida.

Tipos de Exceções:

- **Error**: Representa problemas graves, geralmente fora do controle do programador (e.g., `OutOfMemoryError`, `StackOverflowError`). Não é comum tratá-los.
- **Exception**: Representa erros recuperáveis, divididos em:
  - **Checked Exceptions**: Herdam de `Exception`, mas não de `RuntimeException`. Devem ser tratadas ou declaradas (e.g., `IOException`, `SQLException`).
  - **Unchecked Exceptions**: Herdam de `RuntimeException`. Não exigem tratamento explícito (e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`).

3 Hierarquia de Exceções

A classe `Throwable` é a raiz da hierarquia de exceções. Abaixo está um resumo da estrutura:

```
Throwable
 Error
    OutOfMemoryError
    StackOverflowError
    ...
 Exception
     RuntimeException
        NullPointerException
        ArrayIndexOutOfBoundsException
        ...
     IOException
     SQLException
     ...
```

4 Tratamento de Exceções

O tratamento de exceções em Java é feito com os blocos `try`, `catch`, `finally` e a palavra-chave `throws`. Abaixo, detalhamos cada um.

4.1 Bloco try-catch

O bloco `try` contém o código que pode lançar uma exceção, enquanto o `catch` captura e trata a exceção.

Exemplo:

```
public class ExemploTryCatch {
    public static void main(String[] args) {
        try {
            int[] array = {1, 2, 3};
            System.out.println(array[5]); // Provoca
               ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Erro: Índice fora dos limites: "
               + e.getMessage());
        }
    }
}
```

Saída:

```
Erro: Índice fora dos limites: Index 5 out of bounds for length 3
```

4.2 Múltiplos Blocos catch

Você pode usar vários blocos `catch` para tratar diferentes tipos de exceções. Exceções mais específicas devem vir antes das mais gerais.

Exemplo:

```
public class ExemploMultiCatch {
    public static void main(String[] args) {
        try {
            String texto = null;
            int[] array = {1};
            if (texto.equals("teste") || array[2] == 0) {
                System.out.println("Operação válida");
            }
        } catch (NullPointerException e) {
            System.out.println("Erro: NullPointerException - " +
               e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Erro: Índice fora dos limites -
               " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Erro genérico: " +
               e.getMessage());
        }
    }
}
```

Saída:

```
Erro: NullPointerException - null
```

4.3 Multi-catch (Java 7+)

Desde o Java 7, é possível capturar múltiplas exceções em um único bloco `catch` usando o operador `|`.

Exemplo:

```java
public class ExemploMultiCatchJava7 {
    public static void main(String[] args) {
        try {
            String texto = null;
            System.out.println(texto.length()); // Provoca
               NullPointerException
            int[] array = {1};
            System.out.println(array[2]); // Provoca
               ArrayIndexOutOfBoundsException
        } catch (NullPointerException |
           ArrayIndexOutOfBoundsException e) {
            System.out.println("Erro capturado: " +
               e.getClass().getSimpleName());
        }
    }
}
```

Saída:

```
Erro capturado: NullPointerException
```

4.4 Bloco finally

O bloco `finally` é executado sempre, independentemente de uma exceção ser lançada ou não. É útil para liberar recursos.

Exemplo:

```java
public class ExemploFinally {
    public static void main(String[] args) {
        java.io.PrintWriter writer = null;
        try {
            writer = new java.io.PrintWriter("arquivo.txt");
            writer.println("Teste");
        } catch (java.io.FileNotFoundException e) {
            System.out.println("Erro: Arquivo não encontrado - "
               + e.getMessage());
        } finally {
            System.out.println("Executando finally");
            if (writer != null) {
                writer.close();
            }
        }
    }
```

```
}
```

Saída:

```
Executando finally
```

4.5 Try-with-Resources (Java 7+)

O `try-with-resources` gerencia automaticamente o fechamento de recursos que implementam `AutoCloseable`.

Exemplo:

```
public class ExemploTryWithResources {
    public static void main(String[] args) {
        try (java.io.PrintWriter writer = new
           java.io.PrintWriter("arquivo.txt")) {
            writer.println("Teste com try-with-resources");
        } catch (java.io.FileNotFoundException e) {
            System.out.println("Erro: Arquivo não encontrado - "
               + e.getMessage());
        }
    }
}
```

4.6 Cláusula throws

Métodos que podem lançar exceções *checked* devem declará-las com `throws` ou tratá-las.

Exemplo:

```
public class ExemploThrows {
    public static void verificarArquivo() throws
       java.io.IOException {
        java.io.FileReader file = new
           java.io.FileReader("arquivo_inexistente.txt");
    }

    public static void main(String[] args) {
        try {
            verificarArquivo();
        } catch (java.io.IOException e) {
            System.out.println("Erro: " + e.getMessage());
        }
    }
}
```

Saída:

```
Erro: arquivo_inexistente.txt (No such file or directory)
```

5 Lançando Exceções

Você pode lançar exceções usando a palavra-chave `throw`. Isso é útil para sinalizar erros específicos.

Exemplo:

```
public class ExemploThrow {
    public static void verificarIdade(int idade) throws
       IllegalArgumentException {
        if (idade < 18) {
            throw new IllegalArgumentException("Idade deve ser
               maior ou igual a 18");
        }
        System.out.println("Idade válida: " + idade);
    }

    public static void main(String[] args) {
        try {
            verificarIdade(16);
        } catch (IllegalArgumentException e) {
            System.out.println("Erro: " + e.getMessage());
        }
    }
}
```

Saída:

```
Erro: Idade deve ser maior ou igual a 18
```

6 Exceções Personalizadas

Você pode criar exceções personalizadas estendendo `Exception` (para *checked*) ou `RuntimeException` (para *unchecked*).

Exemplo:

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

public class ExemploExcecaoPersonalizada {
    public static void verificarIdade(int idade) throws
       IdadeInvalidaException {
        if (idade < 18) {
            throw new IdadeInvalidaException("Idade deve ser
               maior ou igual a 18");
        }
        System.out.println("Idade válida: " + idade);
    }
```

```
14
15      public static void main(String[] args) {
16          try {
17              verificarIdade(15);
18          } catch (IdadeInvalidaException e) {
19              System.out.println("Erro: " + e.getMessage());
20          }
21      }
22  }
```

Saída:

```
Erro: Idade deve ser maior ou igual a 18
```

7 Boas Práticas no Tratamento de Exceções

- **Especificidade**: Capture exceções específicas antes de exceções genéricas (e.g., `NullPointerException` antes de `Exception`).
- **Evite capturar Exception genérica**: Isso pode mascarar erros inesperados.
- **Use try-with-resources**: Para gerenciar recursos de forma segura e concisa.
- **Não engula exceções**: Evite blocos `catch` vazios; registre ou trate o erro adequadamente.
- **Documente exceções**: Use `throws` e JavaDoc para indicar possíveis exceções.
- **Crie exceções significativas**: Exceções personalizadas devem fornecer mensagens claras e úteis.

8 Exemplos Práticos

8.1 Validação de Entrada de Usuário

```
1  public class ExemploValidacao {
2      public static void main(String[] args) {
3          java.util.Scanner scanner = new
               java.util.Scanner(System.in);
4          try {
5              System.out.print("Digite um número: ");
6              String entrada = scanner.nextLine();
7              int numero = Integer.parseInt(entrada);
8              System.out.println("Número válido: " + numero);
9          } catch (NumberFormatException e) {
10             System.out.println("Erro: Entrada não é um número
                   válido - " + e.getMessage());
11         } finally {
12             scanner.close();
13         }
14     }
15 }
```

8.2 Leitura de Arquivo

```
public class ExemploLeituraArquivo {
    public static void main(String[] args) {
        try (java.io.BufferedReader reader = new
           java.io.BufferedReader(
                new java.io.FileReader("dados.txt"))) {
            String linha;
            while ((linha = reader.readLine()) != null) {
                System.out.println(linha);
            }
        } catch (java.io.IOException e) {
            System.out.println("Erro ao ler arquivo: " +
               e.getMessage());
        }
    }
}
```

9 Conclusão

O tratamento de exceções é essencial para criar programas Java robustos e confiáveis. Este guia cobriu os conceitos fundamentais, incluindo a hierarquia de exceções, blocos de tratamento, lançamento de exceções e boas práticas. Com os exemplos fornecidos, você pode aplicar essas técnicas em seus projetos. Experimente os códigos e adapte-os às suas necessidades.