Java Developer

Java Core

Sumário

Objetivos deste Módulo

Ao final deste módulo, objetiva-se que o aluno adquira:

Conhecimentos:

- Exceções
  - Trabalhar com exceções checked
  - Trabalhar com exceções unchecked
  - Como lançar exceções
  - Como declarar exceções
  - Como tratar exceções

1. Exceções

Em Java, exceções são eventos ou condições anormais que ocorrem durante a execução de um programa. Quando um programa encontra uma exceção, ele pode interromper sua execução normal e lidar com a exceção de maneira apropriada. As exceções são usadas para tratar erros e situações imprevistas que podem ocorrer durante a execução de um programa.

Existem duas categorias principais de exceções em Java:

- **Exceções verificadas (Checked Exceptions)**: Essas exceções são verificadas em **tempo de compilação** e o código que pode levantar essas exceções **deve tratá-las explicitamente** usando blocos try-catch ou lançar novamente a exceção (para ser tratada posteriormente).

Exemplo de código que lida com uma exceção verificada:

```
// ERRO DE COMPILAÇÃO
FileInputStream file = new FileInputStream("arquivo.txt");

// TRATANDO com bloco try-catch
try {
    FileInputStream file = new FileInputStream("arquivo.txt");
} catch (FileNotFoundException e) {
    System.out.println("O arquivo não foi encontrado.");
}
```

- **Exceções não verificadas (Unchecked Exceptions)**: Também conhecidas como exceções em **tempo de execução**, essas exceções não são verificadas em tempo de compilação. Elas geralmente indicam erros lógicos no código e **não precisam ser tratadas explicitamente**.

Exemplo de código que pode lançar uma exceção não verificada:

```
int[] array = {1, 2, 3};
int valor = array[4]; // Isso lançará ArrayIndexOutOfBoundsException em tempo de execução
```

Para lidar com exceções, você pode usar blocos try-catch para capturar exceções específicas e fornecer código para lidar com elas. Aqui está um exemplo genérico:

```
try {
    // Código que pode lançar uma exceção

} catch (ExcecaoEspecifica e) {
    // Tratar a exceção

} finally {

    // Este bloco é executado, independentemente de ocorrer uma exceção ou não
    // Este bloco é opcional
}
```

Bloco finally: O bloco finally é usado para conter código que deve ser executado independentemente de ocorrer uma exceção ou não. Este bloco é frequentemente usado para liberar recursos, como fechar arquivos, conexões de banco de dados ou recursos de rede, garantindo que esses recursos sejam liberados de forma apropriada.

Exemplo de uso do bloco finally:

```
FileInputStream file = null;

try {

    file = new FileInputStream("arquivo.txt");
    // Realiza operações no arquivo...
    // ...

} catch (FileNotFoundException e) {

    System.out.println("O arquivo não foi encontrado.");

} finally {

    // Lança exceção checkada, precisa bloco try-catch ou lançar com throws
    //file.close();

    // Certifique-se de fechar o arquivo, independentemente do que aconteça
    if (file != null) {
        try {
            file.close();
        } catch (IOException e) {
            System.out.println("Erro ao fechar o arquivo.");
        }
    }
}
```

Criar exceções personalizadas: Em muitos casos, você pode precisar criar suas próprias exceções personalizadas para representar situações específicas do seu aplicativo. Isso é feito criando uma classe que herda de **Exception** ou **RuntimeException**, dependendo se a exceção será **verificada** ou **não verificada**.

Por exemplo, você pode criar uma exceção personalizada UsuarioJaExistenteException:

```
public class UsuarioJaExistenteException extends Exception {

    public UsuarioJaExistenteException(String mensagem) {
        super(mensagem);
    }
}
```

E então o código ficaria assim:

```
public class Testes {

    public static void main(String[] args) {

        // ERRO EM TEMPO DE COMPILAÇÃO - exception checked
        //procurarUsuarioByEmail("qlqcoisa@gmail.com");

        try {
            // Código que pode lançar UsuarioJaExistenteException
            procurarUsuarioByEmail("qlqcoisa@gmail.com");

        } catch (UsuarioJaExistenteException e) {
            System.out.println("Exceção personalizada capturada: " + e.getMessage());
        }

    }

    public static void procurarUsuarioByEmail(String email) throws UsuarioJaExistenteException {

        // procurar no banco de dados...
        boolean temUsuario = true;

        // se encontrar, lança exception
        if(temUsuario) {
            throw new UsuarioJaExistenteException("Já existe usuário com email " + email);
        }
    }
}
```

Se herdar de **RuntimeException**:

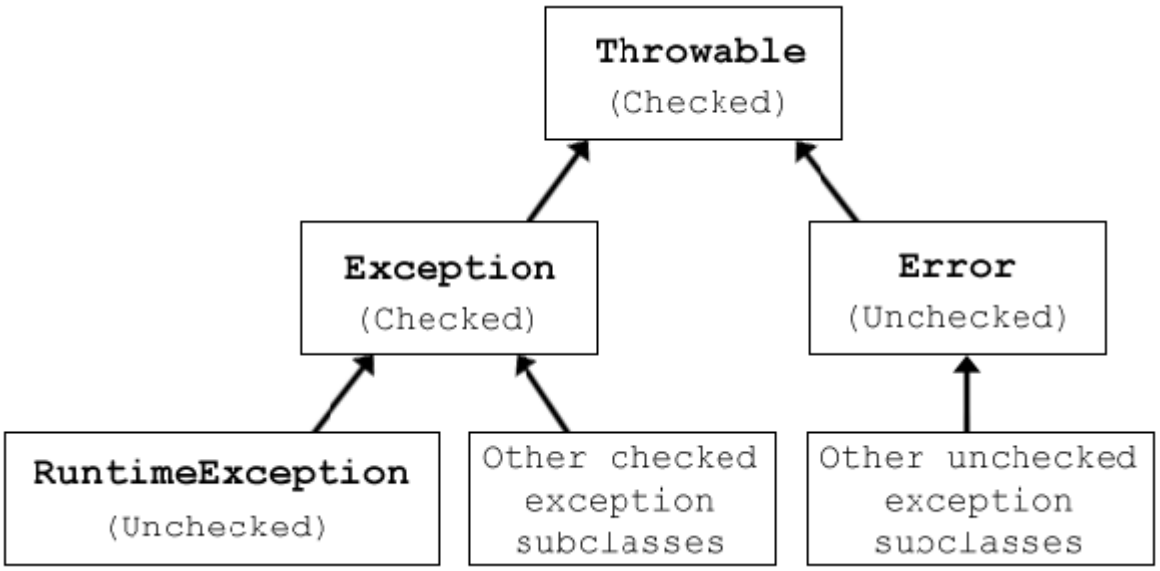
```
public class UsuarioJaExistenteException extends RuntimeException {

    public UsuarioJaExistenteException(String mensagem) {
        super(mensagem);
    }
}
```

Agora, o código ficaria assim:

```
public class Testes {

    public static void main(String[] args) {

        // ERRO EM TEMPO DE EXECUÇÃO - exception unchecked
        procurarUsuarioByEmail("qlqcoisa@gmail.com");
    }

    public static void procurarUsuarioByEmail(String email) {

        // procurar no banco de dados...
        boolean temUsuario = true;

        // se encontrar, lança exception
        if(temUsuario) {
            throw new UsuarioJaExistenteException("Já existe usuário com email " + email);
        }
    }
}
```

Entender o uso correto de exceções e como lidar com elas é essencial para o desenvolvimento de software robusto e confiável em Java. O tratamento adequado de exceções ajuda a evitar falhas catastróficas e permite que seu programa se recupere de erros de maneira controlada.

Esta é a **hierarquia**, mas como já vimos, podemos criar nossas próprias "exceções" com uso de herança:



Resumindo:

throw: Dispara uma exceção.

throws: Indica que um método dispara uma exceção checada.

try: Inicia um bloco onde é esperada uma determinada exceção.

catch: Indica o que fazer quando ocorrer determinada exceção.

finally: Código que deve ser executado independentemente de ocorrer uma exceção ou não.

Empilhamento de exceções (Exception Chaining): Em Java, você pode capturar uma exceção e lançar outra exceção (ou a mesma exceção com informações adicionais) no bloco catch. Isso pode ser útil para adicionar contexto à exceção original ou converter exceções para tipos mais apropriados. Você pode fazer isso usando a palavra-chave **throw**. Chamamos isso de re-throws e throws sem captura - para fazer isso, você pode usar a palavra-chave throw novamente no mesmo bloco catch ou em um bloco catch posterior. Isso permite que você propague a exceção para um nível superior de tratamento.

Exemplo de empilhamento de exceções:

```
public class Testes {

    public static void main(String[] args) {
        fazAlgo();
    }
    public static void fazAlgo() {

        try {
            // Código que pode lançar uma exceção
            int resultado = 1/0;

        } catch (Exception e) {
            throw new IllegalStateException("Ocorreu um erro", e);
            // OU:
            //throw e; // lançará a mesma exceção
        }
    }
}
```

Bloco try-with-resources: você pode usar o bloco try-with-resources para gerenciar automaticamente recursos que precisam ser fechados, como arquivos ou conexões de banco de dados. Isso elimina a necessidade de escrever explicitamente um bloco finally para fechar esses recursos.

Exemplo de uso do bloco try-with-resources:

```
try (FileInputStream file = new FileInputStream("arquivo.txt")) {

    // Realiza operações no arquivo…

} catch (IOException e) {

    System.out.println("Erro ao manipular o arquivo.");

}
```

Hierarquia de exceções e múltiplos blocos catch: Java possui uma hierarquia de exceções em que as exceções mais específicas herdam de exceções mais gerais. Isso permite que você capture exceções específicas antes de exceções mais gerais. Por exemplo, ArithmeticException é uma subclasse de RuntimeException. Portanto, você pode capturar ArithmeticException antes de RuntimeException se desejar tratar erros aritméticos específicos. Em um bloco try, você pode ter vários blocos catch para lidar com diferentes tipos de exceções. Isso permite que você capture exceções específicas e trate-as de maneira adequada. Os blocos catch são verificados na ordem em que são declarados, e o primeiro bloco catch que corresponder ao tipo de exceção será executado.

```
public class Testes {

   public static void main(String[] args) {

      try {

         int i = 1 / 0;

         // testar com valor diferente de zero
         //int i = 1 / 1;

         fazAlgo();

      } catch (ArithmeticException e) {

         System.out.println("Aconteceu ArithmeticException");
         System.out.println(e.getMessage());

      } catch (RuntimeException e) {

         System.out.println("Aconteceu RuntimeException");
         System.out.println(e.getMessage());

      }
   }
```

```
    public static void fazAlgo() {
        // Um método que lance uma exceção RuntimeException
        throw new RuntimeException();
    }

}
```

2. Exercícios

Exercício 1: Calculadora de Divisão por Zero -> Crie um programa de calculadora que permita ao usuário inserir dois números e realize a divisão. No entanto, se o segundo número for zero, lance uma exceção personalizada (por exemplo, **DivisaoPorZeroException**). Capture essa exceção e forneça uma mensagem adequada ao usuário.

Exercício 2: Gerenciamento de Estoque -> Crie uma classe Produto que represente um produto com uma quantidade em estoque. Implemente métodos para adicionar e remover itens do estoque. Se a quantidade disponível ficar negativa após uma remoção, lance uma exceção personalizada (por exemplo, **EstoqueNegativoException**). Em seguida, capture essa exceção ao usar os métodos da classe Produto e trate-a adequadamente.