Notas Práticas e Exercícios 3

Disciplina: Programação Orientada a Objetos

{*Anotações para uso pessoal}

1 try-catch

O try-catch é uma construção da linguagem Java que permite lidar com exceções (erros) em tempo de execução do programa.

Em resumo, o try-catch é utilizado para tentar executar um bloco de código (o try) que pode potencialmente gerar uma exceção. Caso uma exceção ocorra durante a execução do bloco de código, o fluxo de execução é desviado para o bloco catch, onde é possível tratar a exceção.

A sintaxe básica do try-catch é a seguinte:

```
try {
    // código que pode gerar exceção
} catch (Excecao1 e1) {
    // tratamento da exceção Excecao1
} catch (Excecao2 e2) {
    // tratamento da exceção Excecao2
} catch (Exception e) {
    // tratamento de outras exceções não previstas nos catch anteriores
} finally {
    // código a ser executado independentemente de ter ocorrido exceção ou
       não
}
```

Dentro do bloco try, é colocado o código que pode gerar exceção. Caso ocorra uma exceção, o fluxo de execução é desviado para um dos blocos catch, que contém o código responsável por tratar a exceção. É possível ter vários blocos catch para lidar com diferentes tipos de exceções.

O bloco finally é opcional e contém código a ser executado independentemente de ter ocorrido exceção ou não. Por exemplo, é comum utilizar o bloco finally para fechar recursos abertos no bloco try, como arquivos ou conexões de banco de dados.

É importante utilizar o try-catch para lidar com exceções em tempo de execução, pois isso permite que o programa possa se recuperar de erros e continuar a execução em vez de simplesmente encerrar a execução devido a um erro. Além disso, o tratamento de exceções adequado pode tornar o código mais robusto e seguro, evitando que o programa caia em um estado inconsistente ou inesperado.

Exemplo:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class LeituraArquivo {
    public static void main(String[] args) {
```

```
6           try {
7               File arquivo = new File("arquivo.txt");
8               Scanner scanner = new Scanner(arquivo);
9               while (scanner.hasNextLine()) {
10                  String linha = scanner.nextLine();
11                  System.out.println(linha);
12              }
13              scanner.close();
14          } catch (FileNotFoundException e) {
15              System.out.println("Arquivo não encontrado: " + e.getMessage());
16          }
17      }
18  }
```

Neste exemplo, o try é utilizado para tentar criar um objeto Scanner que irá ler o conteúdo do arquivo arquivo.txt. Caso o arquivo não seja encontrado, o fluxo de execução é desviado para o bloco catch onde é exibida uma mensagem de erro indicando o motivo da exceção.

É importante destacar que a exceção FileNotFoundException é uma subclasse de IOException, que é uma exceção mais genérica que pode ser lançada por qualquer operação de entrada/saída em Java. Portanto, é possível utilizar o bloco catch para tratar outras exceções relacionadas a operações de entrada/saída que possam ocorrer durante a leitura do arquivo. Explicação das linhas:

- **Linha 1**: importa as classes necessárias para ler um arquivo de texto com `Scanner` e tratamento de exceções.
- **Linha 4**: declaração da classe `LeituraArquivo`.
- **Linha 5**: declaração do método `main`, que é o ponto de entrada do programa.
- **Linha 6**: início do bloco `try`, que contém o código que pode gerar exceções.
- **Linha 7**: cria um objeto `File` que representa o arquivo a ser lido.
- **Linha 8**: cria um objeto `Scanner` para ler o conteúdo do arquivo.
- **Linha 9**: início do laço que percorre todas as linhas do arquivo.
- **Linha 10**: lê a próxima linha do arquivo e armazena em uma variável.
- **Linha 11**: exibe a linha lida na saída padrão.
- **Linha 12**: fim do laço.
- **Linha 13**: fecha o objeto `Scanner` para liberar os recursos.
- **Linha 14**: fim do bloco `try`.
- **Linha 14**: início do bloco `catch`, que trata a exceção `FileNotFoundException` caso o arquivo não seja encontrado.
- **Linha 15**: exibe uma mensagem de erro informando o motivo da exceção.
- **Linha 16**: fim do bloco `catch`.

2 Exercícios Práticos

2.1 Andando de Elevador

Vamos criar um exercício simples em Java que envolve um prédio, um elevador e uma pessoa. O objetivo é simular o funcionamento de um elevador que leva pessoas entre os andares do prédio.

Crie as seguintes classes:

1. Classe Pessoa
2. Classe Elevador
3. Classe Predio
4. Classe Main (para testar o programa)

Classe Pessoa:

```java
public class Pessoa {
    private String nome;
    private int destino;

    public Pessoa(String nome, int destino) {
        this.nome = nome;
        this.destino = destino;
    }

    public String getNome() {
        return nome;
    }

    public int getDestino() {
        return destino;
    }
}
```

Classe Elevador:

```java
import java.util.ArrayList;

public class Elevador {
    private int andarAtual;
    private ArrayList<Pessoa> passageiros;

    public Elevador() {
        andarAtual = 0;
        passageiros = new ArrayList<>();
    }

    public void embarcar(Pessoa pessoa) {
        passageiros.add(pessoa);
        System.out.println(pessoa.getNome() + " entrou no elevador.");
    }

    public void desembarcar(Pessoa pessoa) {
```

```
        passageiros.remove(pessoa);
        System.out.println(pessoa.getNome() + " saiu do elevador.");
    }

    public void mover(int andar) {
        System.out.println("Elevador se movendo para o andar " + andar);
        for (Pessoa pessoa : new ArrayList<>(passageiros)) {
            if (pessoa.getDestino() == andar) {
                desembarcar(pessoa);
            }
        }
        andarAtual = andar;
    }
}
```

Classe Prédio:

```
public class Predio {
    private int totalAndares;
    private Elevador elevador;

    public Predio(int totalAndares) {
        this.totalAndares = totalAndares;
        this.elevador = new Elevador();
    }

    public Elevador getElevador() {
        return elevador;
    }
}
```

Classe Main:

```
    public class Main {
    public static void main(String[] args) {
        Predio predio = new Predio(10);
        Elevador elevador = predio.getElevador();

        Pessoa pessoa1 = new Pessoa("João", 5);
        Pessoa pessoa2 = new Pessoa("Maria", 8);

        elevador.embarcar(pessoa1);
        elevador.embarcar(pessoa2);

        elevador.mover(5);
        elevador.mover(8);
    }
}
```

2.2 Calculadora com sobrecarga de métodos

Crie uma classe chamada Calculadora que realiza operações matemáticas básicas, como adição, subtração, multiplicação e divisão. Utilize a sobrecarga de métodos para que a calculadora possa lidar com diferentes

tipos de dados, como inteiros e números de ponto flutuante (double).

```
--------------------------------
|          Calculadora          |
--------------------------------
|                               |
--------------------------------
| + somar(a: int, b: int): int  |
| + somar(a: double, b: double):|
|             double            |
| + subtrair(a: int, b: int): int|
| + subtrair(a: double, b: double):|
|             double            |
| + multiplicar(a: int, b: int): int|
| + multiplicar(a: double, b: double):|
|             double            |
| + dividir(a: int, b: int): int|
| + dividir(a: double, b: double):|
|             double            |
--------------------------------
```

2.3 Sistema de Pedidos com sobrecarga de métodos, agregação e associação

Crie um sistema de pedidos simples que inclua as classes Cliente, Endereco, ItemPedido e Pedido. Utilize a sobrecarga de métodos, agregação e associação entre as classes. As classes devem se comunicar da seguinte maneira:

- A classe Cliente possui uma **agregação** com a classe Endereco, armazenando informações sobre o endereço do cliente.
- A classe Pedido possui uma **agregação** com a classe Cliente, armazenando informações sobre o cliente que fez o pedido.
- A classe Pedido possui uma **composição** com a classe ItemPedido, armazenando uma lista de itens do pedido.

2.4 Locadora de Carros

A classe **Carro** representa um carro da locadora e possui um modelo, uma placa e um preço por dia de locação. A classe **Locacao** representa uma locação de um carro por um cliente e possui um número, uma data de início, uma data de fim, um cliente e um carro. A classe **Cliente** representa um cliente da locadora e possui um nome, um CPF e uma lista de locações realizadas.
-Faça um método que calcule o total do valor da Locação baseado na data de início e final.

Vamos implementar as classes:

```java
public class Carro {
    private String modelo;
    private String placa;
    private double precoDiaria;

    public Carro(String modelo, String placa, double precoDiaria) {
        this.modelo = modelo;
        this.placa = placa;
```

```
9           this.precoDiaria = precoDiaria;
10      }
11
12      // getters e setters
13
14      /**
15       * @return the modelo
16       */
17      public String getModelo() {
18          return modelo;
19      }
20
21      /**
22       * @param modelo the modelo to set
23       */
24      public void setModelo(String modelo) {
25          this.modelo = modelo;
26      }
27
28      /**
29       * @return the placa
30       */
31      public String getPlaca() {
32          return placa;
33      }
34
35      /**
36       * @param placa the placa to set
37       */
38      public void setPlaca(String placa) {
39          this.placa = placa;
40      }
41
42      /**
43       * @return the precoDiaria
44       */
45      public double getPrecoDiaria() {
46          return precoDiaria;
47      }
48
49      /**
50       * @param precoDiaria the precoDiaria to set
51       */
52      public void setPrecoDiaria(double precoDiaria) {
53          this.precoDiaria = precoDiaria;
54      }
55  }
56
57
58  public class Cliente {
59      private String nome;
```

```
60      private String cpf;
61      private ArrayList<Locacao> locacoes;
62
63      public Cliente(String nome, String cpf) {
64          this.nome = nome;
65          this.cpf = cpf;
66          this.locacoes = new ArrayList<>();
67      }
68
69      public void adicionarLocacao(Locacao locacao) {
70          getLocacoes().add(locacao);
71      }
72      //coloque os metodos set e get
73  }
74
75  public class Locacao {
76      private static int proximoNumero = 1;
77      private int numero;
78      private String dataInicio;
79      private String dataFim;
80      private Cliente cliente;
81      private Carro carro;
82
83      public Locacao(String dataInicio, String dataFim, Cliente cliente,
            Carro carro) {
84          this.numero = proximoNumero;
85          proximoNumero++;
86          this.dataInicio = dataInicio;
87          this.dataFim = dataFim;
88          this.cliente = cliente;
89          cliente.adicionarLocacao(this);
90          this.carro = carro;
91      }
92
93       //coloque os metodos set e get
94  }
```

Main:

```
1   public static void main(String[] args) {
2
3           Carro c1 = new Carro("Fiat Uno", "ABC-1234", 50.0);
4           Carro c2 = new Carro("Volkswagen Gol", "DEF-5678", 60.0);
5           Carro c3 = new Carro("Ford Fiesta", "GHI-9012", 55.0);
6
7           Cliente cl1 = new Cliente("João Silva", "123.456.789-00");
8           Cliente cl2 = new Cliente("Maria Santos", "987.654.321-00");
9
10          Locacao loc1 = new Locacao("10/01/2023", "10/03/2023", cl1, c1);
11          Locacao loc2 = new Locacao("10/03/2023", "10/04/2023", cl2, c2);
12          Locacao loc3 = new Locacao("13/01/2023", "19/01/2023", cl1, c3);
13
```

```
14          System.out.println("O carro " + loc1.getCarro().getModelo() + " foi
               alugado por " + loc1.getCliente().getNome());
15          System.out.println("O valor da locação do carro " +
               loc2.getCarro().getModelo() + " foi de " +
               loc2.getCarro().getPrecoDiaria() + " reais por dia.");
16  }
```