EXCEÇÕES

Introdução à Orientação a Objetos

Prof. Dr. Diego Marczal

PROF. DR. DIEGO MARCZAL

Universidade Tecnológica Federal do Paraná
Câmpus Guarapuava

Curso

Tecnologia em Sistemas para Internet

Disciplina

Introdução à Orientação a Objetos

Linguagem de Programação

Java

Assunto

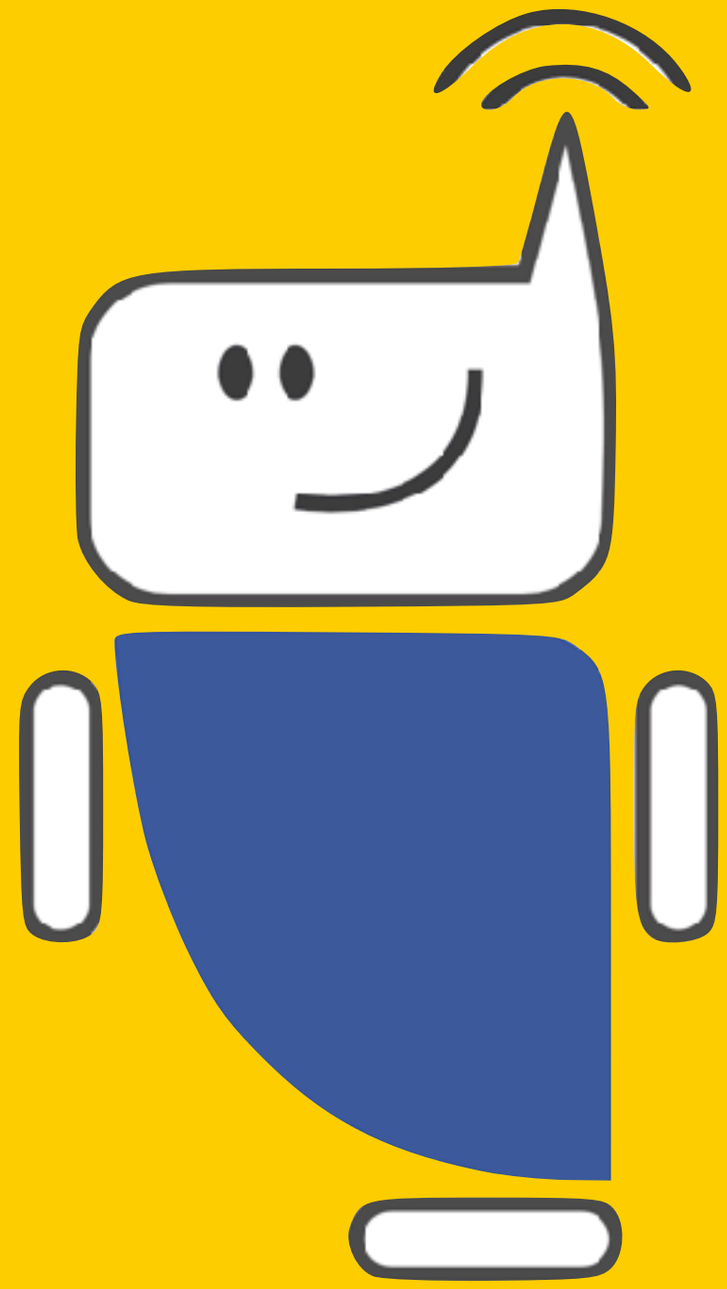
Exceções

UTFPR

CONTEÚDO

1. Programação defensiva.
2. Tratamento de exceções
    1. Divisão por zero.
3. Categorias de exceção.
4. Notificação por meio de exceções.

PROGRAMAÇÃO DEFENSIVA



PROGRAMAÇÃO DEFENSIVA

Um dos princípios de design por trás do Java é que não interessa saber quanto demora para fazer um software rapidamente e de qualquer maneira. **A medida real é quanto tempo leva para se escrever um código sólido**.

James Gosling - criador do Java (set/2003).
Disponível em: http://www.artima.com/intv/solidP.html

O "CAMINHO ÚNICO DA VERDADE" E O TRATAMENTO DE EXCEÇÕES

Normalmente na universidade só aprende-se a programar o "caminho único da verdade" (*one true path*). Mas, nas aplicações reais essa abordagem precisa mudar radicalmente se você quiser sobreviver.

"Caminho único da verdade"

Programar considerando que falhas nunca ocorrerão

O "CAMINHO ÚNICO DA VERDADE" E O TRATAMENTO DE EXCEÇÕES

Em um extremo (**aviônica, informática médica**) pode-se gastar 90% do tempo tratando exceções. Em aplicações financeiras, pode se gastar muito mais.

James Gosling diz que um amigo seu perdeu **13 bilhões de doláres** em uma noite por falta de tratamento de exceção.

Tudo depende de quais são as conseqüências da falha.

ALGUMAS CAUSAS DAS SITUAÇÕES DE ERROS

- Implementação incorreta.
- Não atende à especificação.
- Solicitação de objeto inapropriado.
  - Por exemplo, o uso de um variável do tipo objeto que está nula
- Estado do objeto inconsistente ou inadequado.
  - Valores inadequados nos parâmetros, o que causa o estado inconsistente.

NEM SEMPRE ERRO DO PROGRAMADOR

Erros surgem frequentemente do ambiente

- URL incorreto inserido; e interrupção da rede.

Processamento de arquivos é particularmente propenso a erros

- Arquivos ausentes.
- Falta de permissões apropriadas.

SITUAÇÕES DE ERROS

Quais as causas de erros?

- Erro de programação.
- Operações impossíveis.
- Fatores externos: memória, arquivos ausentes, etc.

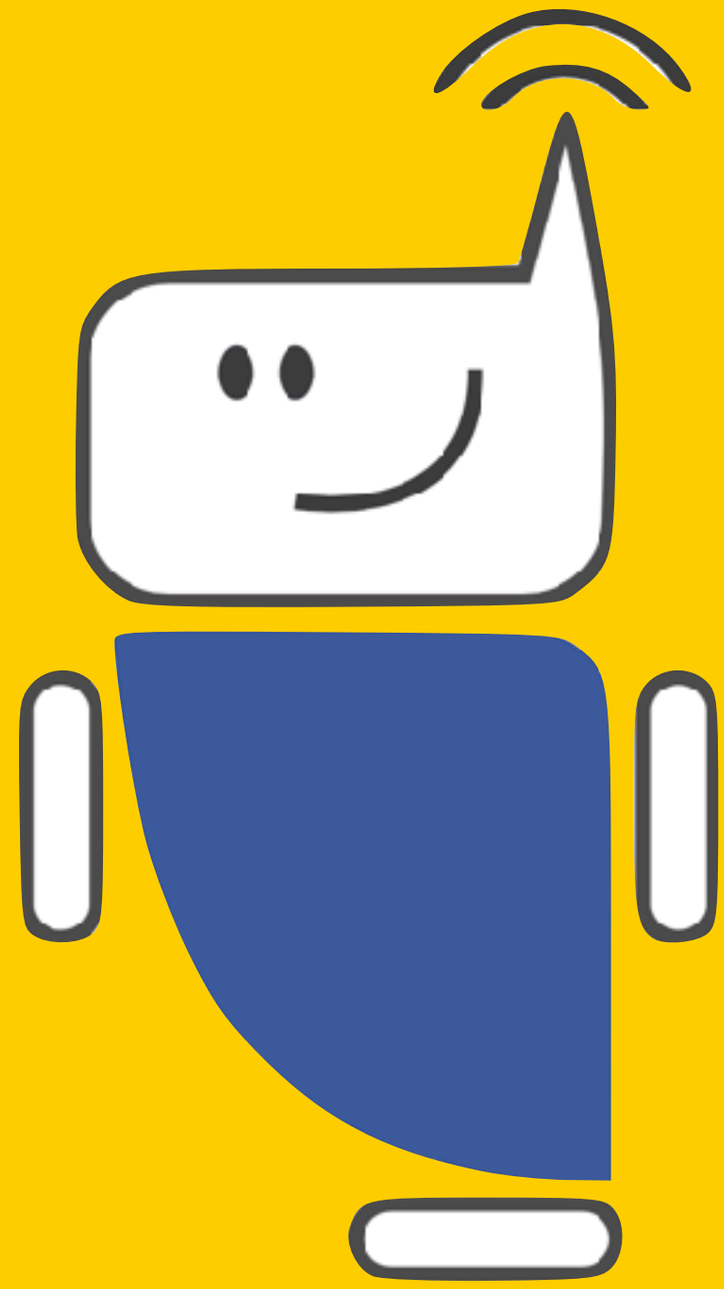
É necessário saber

- Reagir a erros quando eles ocorrem.
- Relatar erros.
- Tratar erros de entrada/saída textuais.

DEVO SEMPRE PROGRAMAR DEFENSIVAMENTE?

- Uma classe pode supor que os clientes sabem o que estão fazendo.
- Uma classe pode antecipar todas as possibilidades de problemas e estar pronta para reagir a elas.
- **A verdade está em algum lugar entre estes extremos.**

TRATAMENTO DE EXCEÇÕES



DEFINIÇÃO DE EXCEÇÃO

- É o indicativo de que algum tipo de condição excepcional ocorreu durante a execução do programa.
- Exceções estão associadas a condições de erro que não tinham como ser verificadas durante a compilação do programa.
- O tratamento de exceções ajuda a aprimorar a tolerância a falhas de um programa.
- Em um programa, quando ocorre uma exceção e esta não é tratada, este para sua execução, podendo ocorrer prejuízos catastróficos.
    - Pense se o sistema de uma aeronave falhar em pleno voo!

NULLPOINTEREXCETION

- Provavelmente, em seus códigos, durante a tentativa de execução, já ocorreu a exceção `NullPointerException`.
- Esta acontece quando uma variável com referência null é usada onde se esperava um objeto.
- Por exemplo, usar uma variável de instância do tipo referência sem essa ter recebido uma instância de um objeto.
- Perceba que execução do código ao lado lançará um exceção do tipo `NullPointerException`.

```
Exception in thread "main" java.lang.NullPointerException
    at Agenda.adiciona(Agenda.java:10)
    at Principal.main(Principal.java:4)
```

```java
import java.util.ArrayList;

public class Agenda {

    private ArrayList<String> notas;

    public void adiciona(String nota) {
        notas.add(nota);
    }

    public int getQuantidadeDeNotas() {
        return notas.size();
    }

    public String getNota(int indice) {
        return notas.get(indice);
    }

    public String removerNota(int indice) {
        return notas.remove(indice);
    }
}

public class Principal {
    public static void main(String[] args) {
        Agenda agenda = new Agenda();
        agenda.adiciona("Estudar OO");
    }
}
```

O EFEITO DE UMA EXCEÇÃO

- A execução do método em que ocorre uma exceção termina prematuramente.
- Nenhum valor é retornado.
- O fluxo de execução não retorna para o ponto de invocação do método.
- Portanto, de qualquer maneira, o método que invocou a ação não pode prosseguir, porém, pode 'capturar' a exceção para evitar seu encerramento prematuro.
- De modo geral o efeito de um exceção, quando não tratada encerra a execução do programa.

DIVISÃO POR ZERO

```java
import java.util.Scanner;

public class Calculadora {
    public static int dividir(int a, int b) {
        return a / b;
    }
}

public class InterfaceTexto {
    private Scanner entrada;

    public InterfaceTexto() {
        entrada = new Scanner(System.in);
    }

    public void executar() {
        int a = entrada.nextInt();
        int b = entrada.nextInt();

        int resultado = Calculadora.dividir(a, b);

        sln("Resultado: " + resultado);
    }

    private void sln(String s) {
        System.out.println(s);
    }
}

public class Main {
    public static void main(String[] args) {
        InterfaceTexto it = new InterfaceTexto();
        it.executar();
    }
}
```

Observação: Os tipos float e double implementam o padrão IEEE 754, que define que uma divisão por zero deve retornar um valor especial “infinity".

```
1º Execução:
10
2
Resultado: 5

2º Execução:
10
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Calculadora.dividir(Main.java:6)
        at InterfaceTexto.executar(Main.java:22)
        at Main.main(Main.java:35)
```

- A primeira execução é bem sucedida.
- Na segunda execução ocorre uma exceção do tipo java.lang.ArithmeticException, pois o método detecta um problema e é incapaz de tratá-lo.
- A execução é encerrada no ponto em que ocorreu a exceção, perceba que o resultado não é impresso como na primeira execução.
- As informações apresentadas são chamadas de rastreamento de pilha (*stack trace*). Permite visualizar todo caminho da execução até ocorrer a exceção.

TRATAMENTO DA DIVISÃO POR ZERO

```java
import java.util.Scanner;

class Calculadora {
  public static int dividir(int a, int b) throws ArithmeticException
  {
      return a / b;
   }
}

class InterfaceTexto {
     private Scanner entrada;

     public InterfaceTexto() {
          entrada = new Scanner(System.in);
     }

     public void executar() {
       try {
          int a = entrada.nextInt();
          int b = entrada.nextInt();

          int resultado = Calculadora.dividir(a, b);

          sln("Resultado: " + resultado);
       } catch (ArithmeticException e) {
          sln("Exceção: " + e.getMessage());
          sln("Zero não é um denominador válido,
               por favor tente novamente");
       }
     }

     private void sln(String s) {
          System.out.println(s);
     }
}

public class Main {
     public static void main(String[] args) {
          InterfaceTexto it = new InterfaceTexto();
          it.executar();
     }
}
```

1º Execução:

```
10
2
Resultado: 5

10
0
Exceção: / by zero
Zero não é um denominador válido, por favor tente novamente
```

- A primeira execução é bem sucedida.
- Na segunda o usuário entrar com o denominador zero.
  - A exceção acontece no método dividir da classe Calculadora.
  - A exceção é passada para quem a chamou pela instrução **`throws`** `ArithmeticException`.
  - O bloco **`try`** `{}` **`catch`** () `{}` captura a exceção.
  - O código do bloco **`catch`** é executado para que o sistema se recupere da exceção ocorrida.

TRATAMENTO DA DIVISÃO POR ZERO

```java
import java.util.Scanner;

class Calculadora {
  public static int dividir(int a, int b) throws ArithmeticException
  {
      return a / b;
   }
}

class InterfaceTexto {
     private Scanner entrada;

     public InterfaceTexto() {
          entrada = new Scanner(System.in);
     }

     public void executar() {
       try {
          int a = entrada.nextInt();
          int b = entrada.nextInt();

          int resultado = Calculadora.dividir(a, b);

          sln("Resultado: " + resultado);
       } catch (ArithmeticException e) {
          sln("Exceção: " + e.getMessage());
          sln("Zero não é um denominador válido,
               por favor tente novamente");
       }
     }

     private void sln(String s) {
          System.out.println(s);
     }
}

public class Main {
     public static void main(String[] args) {
          InterfaceTexto it = new InterfaceTexto();
          it.executar();
     }
}
```

A instrução **`throws`** é utilizada enviar a exceção ocorrida dentro do método que o define para o método que o invocou.

No caso esta repassa a exceção `ArithmeticException`, quando essa ocorre.

Neste casso quem deve tratar a exceção é a classe InterfaceTexto, pois sua responsabilidade é ler dados do usuário assim como apresentar informações a ele.

O escopo do bloco **`try`** deve conter o código que pode lançar uma exceção, pois quando essa ocorrer o fluxo da execução será repassada para o bloco **`catch.`**

O **`catch`** capturará a exceção, no caso do tipo `ArithmeticException`. O seu escopo deve conter o código necessário para que o sistema se recupere do erro ocorrido e possa continuar com a execução.

ENTRADA INVÁLIDA

```java
import java.util.Scanner;

class Calculadora {
  public static int dividir(int a, int b) throws ArithmeticException
  {
      return a / b;
   }
}

class InterfaceTexto {
      private Scanner entrada;

      public InterfaceTexto() {
            entrada = new Scanner(System.in);
      }

      public void executar() {
        try {
            int a = entrada.nextInt();
            int b = entrada.nextInt();

            int resultado = Calculadora.dividir(a, b);

            sln("Resultado: " + resultado);
        } catch (ArithmeticException e) {
            sln("Exceção: " + e.getMessage());
            sln("Zero não é um denominador válido,
                  por favor tente novamente");
        }
      }

      private void sln(String s) {
            System.out.println(s);
      }
}

public class Main {
      public static void main(String[] args) {
            InterfaceTexto it = new InterfaceTexto();
            it.executar();
      }
}
```

```
1º Execução:
10
2
Resultado: 5

2º Execução:
10
dois
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:939)
        at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
        at InterfaceTexto.executar(Main.java:19)
        at Main.main(Main.java:38)
```

O método `nextInt()` também pode lançar uma exceção quando a entrada lida não for um inteiro.

Perceba que na segunda execução entrada para o denominador não é um número inteiro. Quando este valor é recebido pelo programa esse não consegue converter para inteiro e então lança a exceção do tipo java.util.InputMismatchException.

Após a exceção ser lançada a execução do programa é encerrada pois a mesma não é tratada.

TRATANDO ENTRADA INVÁLIDAS

```java
import java.util.InputMismatchException;
import java.util.Scanner;

class Calculadora {
	public static int dividir(int a, int b) throws ArithmeticException {
		return a / b;
	}
}

class InterfaceTexto {
	private Scanner entrada;

	public InterfaceTexto() {
		entrada = new Scanner(System.in);
	}

	public void executar() {
		try {
			int a = entrada.nextInt();
			int b = entrada.nextInt();

			int resultado = Calculadora.dividir(a, b);

			sln("Resultado: " + resultado);
		} catch (ArithmeticException e) {
			sln("Exceção: " + e.getMessage());
			sln("Zero não é um denominador válido,
			    por favor tente novamente");
		} catch (InputMismatchException e) {
			sln("Exceção: Entrada inválida");
			sln("A entrada digitada não é um número inteiro");
		}
	}

	private void sln(String s) {
		System.out.println(s);
	}
}

public class Main {
	public static void main(String[] args) {
		InterfaceTexto it = new InterfaceTexto();
		it.executar();
	}
}
```

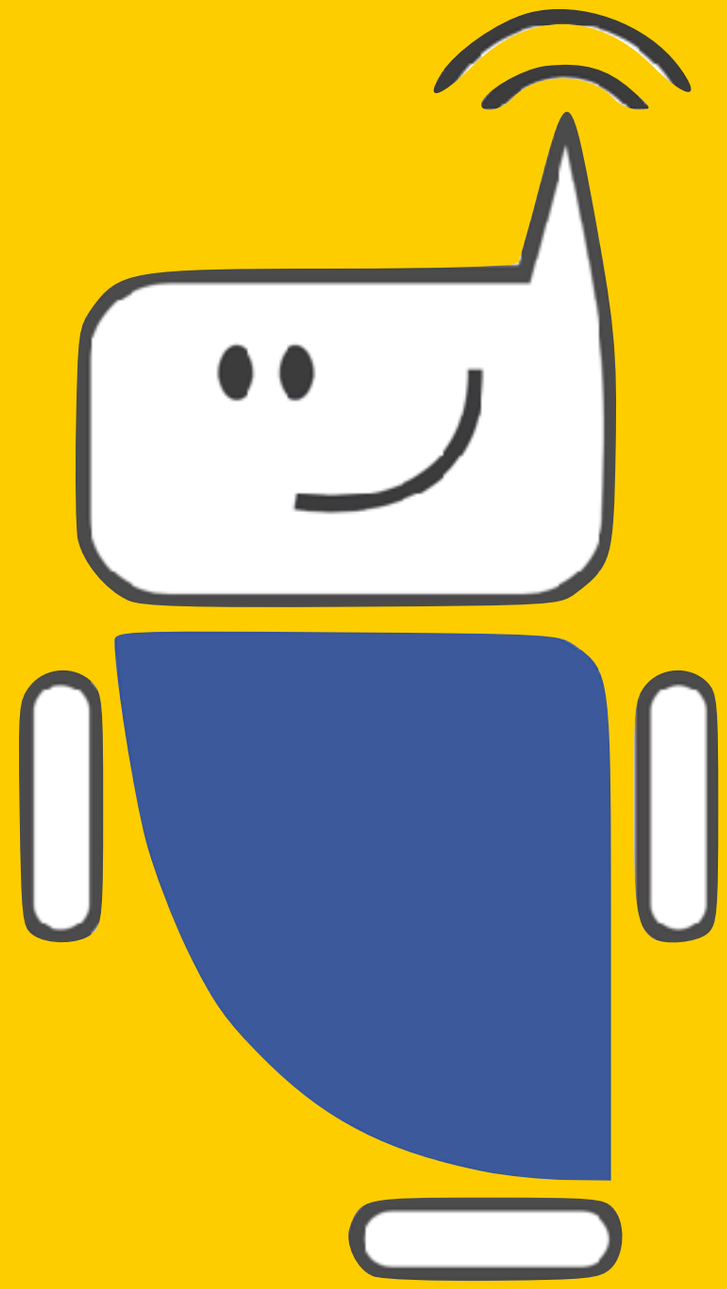
1º Execução:
10
2
Resultado: 5

2º Execução:
10
dois
Exceção: Entrada inválida
A entrada digitada não é um número inteiro

O escopo do bloco **try** deve conter o código que pode lançar exceções, quando uma exeção ocorrer o fluxo da execução será repassada para o bloco **catch.**

Um bloco **try** pode conter quantos **catchs** forem necessários. No caso o primeiro **catch** o captura a exceção do tipo `ArithmeticException` e o segundo do tipo `InputMismatchException.`

CATEGORIAS DE EXCEÇÕES



EXISTEM DOIS TIPOS DE EXCEÇÃO

Exceções não-verificadas

- **Não** é necessário estar preparado para elas e nem tratá-las se não for necessário.
  - O uso do bloco `try` e `catch` não é exigido.
- A divisão de um inteiro por zero (`ArithmeticException`) e entradas inválidas (`java.util.InputMismatchException`) são exemplos.

Exceções verificadas

- Utilizadas para falhas antecipáveis.
- É necessário tratar a exceção.
- O compilador ajuda a garantir que você está ciente de todas as exceções verificadas que podem ser lançadas, pois se não tratadas o código não compila.
- Abertura e escrita em arquivos exigem tratamento de exceções, por isso são exceções verificadas, como por exemplo `FileNotFoundException` e `IOException`.

EXCEÇÕES VERIFICADAS – ESCRITA E LEITURA DE ARQUIVO TXT

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class BancoDeDados {

	private static final String NOME_DO_ARQUIVO = "nota.txt";
	private File arquivo = new File(NOME_DO_ARQUIVO);

	public BancoDeDados() {
		try {
			arquivo.createNewFile();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
	public void salvar(String texto) {
		try {
			FileWriter escritor = new FileWriter(arquivo);
			escritor.write(texto);
			escritor.close();
		} catch(IOException e) {
			e.printStackTrace();
		}
	}
	public String getTexto() {
		try {
			FileReader leitor = new FileReader(arquivo);
			BufferedReader buffer = new BufferedReader(leitor);

			String segredo = buffer.readLine();

			buffer.close();
			return segredo;
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}

		return null;
	}
}
```

A classe `BancoDeDados` simula um banco de dados simples, com dois objetivos: 1) Escrever uma string em um arquivo TXT; e Ler uma String de um arquivo TXT.

Constante ***NOME_DO_ARQUIVO***, contém a `String` com o nome do arquivo a ser tuilizado.

O atributo `arquivo` do tipo `File`, representa um arquivo, o qual será lido e escrito pelo sistema.

No construtor, cria-se um arquivo se ele não existir, isso é feito invocando o método `createNewFile()` do objeto referenciado pela variável `arquivo`.

Perceba que aqui é feito o tratamento da exceção `IOException`, uma exceção **verificada** que pode ocorrer na escrita do arquivo ou na verificação se o arquivo já existe.

Caso a exceção ocorra, neste exemplo, apenas é listado a pilha de rastreamento, pelo método `printStackTrace()` da exceção referenciada pela variável `e`.

EXCEÇÕES VERIFICADAS - ESCRITA E LEITURA DE ARQUIVO TXT

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class BancoDeDados {

	private static final String NOME_DO_ARQUIVO = "nota.txt";
	private File arquivo = new File(NOME_DO_ARQUIVO);

	public BancoDeDados() {
		try {
			arquivo.createNewFile();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
	public void salvar(String texto) {
		try {
			FileWriter escritor = new FileWriter(arquivo);
			escritor.write(texto);
			escritor.close();
		} catch(IOException e) {
			e.printStackTrace();
		}
	}
	public String getTexto() {
		try {
			FileReader leitor = new FileReader(arquivo);
			BufferedReader buffer = new BufferedReader(leitor);

			String segredo = buffer.readLine();

			buffer.close();
			return segredo;
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}

		return null;
	}
}
```

O método `salvar` é responsável por salvar uma `String`, recebida pelo parâmetro `texto`. Para isso é necessário um escritor do tipo `FileWriter`, o qual recebe em seu construtor o atributo `arquivo` como argumento. Este será usado pelo escritor para escrever o `String` recebido.

Após faz uso do método `write` para escrever o conteúdo do parâmetro `texto`.

Por fim, é necessário fechar o `arquivo` por meio do método `close` para evitar um possível corrompimento do arquivo.

O `FileWriter` pode gerar uma exceção verificada do tipo `IOException`, por isso é necessário realizar o tratamento da exceção com o bloco **`try`** e **`catch.`**

Caso a exceção ocorra, neste exemplo, apenas é listado a pilha de rastreamento, pelo método `printStackTrace()` da exceção referenciada pela variável `e`.

EXCEÇÕES VERIFICADAS – ESCRITA E LEITURA DE ARQUIVO TXT

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class BancoDeDados {

	private static final String NOME_DO_ARQUIVO = "nota.txt";
	private File arquivo = new File(NOME_DO_ARQUIVO);

	public BancoDeDados() {
		try {
			arquivo.createNewFile();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
	public void salvar(String texto) {
		try {
			FileWriter escritor = new FileWriter(arquivo);
			escritor.write(texto);
			escritor.close();
		} catch(IOException e) {
			e.printStackTrace();
		}
	}
	public String getTexto() {
		try {
			FileReader leitor = new FileReader(arquivo);
			BufferedReader buffer = new BufferedReader(leitor);

			String texto = buffer.readLine();

			buffer.close();
			return texto;
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}

		return null;
	}
}
```

O método `getTexto()` é responsável por retornar a `String` lida do arquivo TXT.

Para isso é necessário um `leitor` do tipo `FileReader`, o qual recebe em seu construtor o atributo `arquivo` como argumento. Este será usado pelo leitor para acessar o arquivo.

Além disso é necessário um `buffer` do tipo `BufferedReader` para carregar o conteúdo do arquivo em memória, para que o acesso ao conteúdo seja mais rápido. Este recebe em seu construtor o `leitor`, o qual tem acesso ao arquivo por meio do atributo `arquivo`.

Pelo `buffer` é possível ler cada linha do conteúdo do arquivo por meio do método `readLine()`; Nesse caso, só lemos um linha e seu conteúdo é referenciado pela variável local `texto`.

Então é necessário encerrar o buffer pelo método `close()`, e retornar a variável local `texto`, que referencia o conteúdo lido.

O `FileReader` pode gerar uma exceção verificada do tipo `FileNotFoundException`, por isso é necessário realizar o tratamento da exceção com o bloco **`try`** e **`catch.`**

O método `readLine()` do `BufferedReader` pode lançar uma exceção **verificada** do tipo `IOException`, por isso é necessário realizar o tratamento da exceção com o bloco **`try`** e **`catch.`**

Em ambos os casos, se a exceção ocorrer, apenas é listado a pilha de rastreamento, pelo método `printStackTrace()` da exceção referenciada pela variável `e`. Porém, pelo tratamento da exceção o programa não é encerrado prematuramente.

RELATO DE ERROS

NOTIFICAÇÃO

- Relatar erros torna mais provável que eles sejam percebidos e/ou eliminados.
- O retorno de um método é uma forma de indicar se uma operação foi bem sucedida ou não.
  - É útil para indicar se a operação foi bem sucedida, mas não um parâmetro incorreto.
  - Não interrompe a execução do programa, dessa forma ninguém fica sabendo do que ocorreu.

```
public class Conta {

    private double saldo;

    public boolean depositar(double valor)
    {
        if (valor > 0) {
            this.saldo += valor;
            return true;
        } else {
            return false;
        }
    }

    // demais métodos omitidos
}
```

NOTIFICAÇÃO

- Uma aplicação pode relatar erros para os usuários:
  - Através de instruções System.*out*.println.
  - Através da exibição de diálogos e janelas de erros.
- A desvantagens desta abordagem é que nem sempre são usuários humanos que executarão a aplicação
- Mesmo se humanos usam a aplicação, normalmente eles não podem fazer nada a respeito dos erros que vêem.
- Além disso, pode violar, facilmente o princípio do conhecimento mínimo, por exemplo não é responsabilidade da classe conta imprimir algo na tela.

```
public class Conta {

  private double saldo;

  public void depositar(double valor) {
    if (valor == 0) {
      System.out.println("Incremento
                          desnecessário!");
    } else if (valor < 0) {
      System.out.println("Valores negativos
                          não são incrementados");
    } else {
      this.saldo += valor;
    }
  }

  // demais métodos omitidos
}
```

NOTIFICAÇÃO

- Pode-se se relatar erros por meio de exceções.
- No construtor da classe `Conta` é lançada um exceção do tipo `IllegalStateException` se o saldo for menor que zero.
- No método `depositar`(**double** valor), é lançada uma exceção quando o valor for zero ou quando for menor que zero.
- Perceba que pode se usar mensagens específicas para cada tipo de erro.
- A classe que instanciar um objeto de classe conta poderá tratar a exceção.
- O objeto que invocar o método depositar poderá tratar as possíveis exceções.

```
public class Conta {

  private double saldo;

  public Conta(double saldo) {
    if (saldo < 0)
        throw new IllegalStateException("Saldo
                                          inválido!");
    this.saldo = saldo;
  }

  public void depositar(double valor) {
    if (valor == 0) {
          throw new IllegalArgumentException("Zero é um
                                        argumento inválido!");
    } else if (valor < 0) {
       throw new IllegalArgumentException("Não é
                                  permitido valor negativo");
    } else {
       this.saldo += valor;
    }
  }

   // demais métodos omitidos
}
```

OBSERVAÇÃO

Objetivo para esta disciplina é entender o básico sobre exceções, a saber:

- O que são exceções?
- Quando podem ocorrerem?
- Diferenciar exceções verificadas de não verificadas.
- Como tratar exceções.
- Como lançar exceções.

O assunto de exceções é mais extenso do que apresentado neste material. Para se aprofundar mais leia os capítulos apresentados nas referências e aprofunde-se estudando:

- Bloco Finally
- Hierarquia de Exceções
- Definindo classes que representam exceções.

MOMENTO DESCONTRAÇÃO

Two students are leaving a Java seminar

- The guy turns to the girl and says "So... how much do you weigh?”
- The girl says, "I'm not telling you! That's private!”
- Taken aback, the guy says "But I thought we were in the same class!!"

“

Os justos clamam, o Senhor os ouve e os livra de todas as suas tribulações.

Salmos 34:17 NVI

REFERÊNCIAS

- Livros
  - BARNES, David J.; KÖLLING, Michael. **Programação orientada a objetos com java: uma introdução prática usando o BlueJ**. 4. ed. São Paulo, SP: Pearson Prentice Hall, 2009. xxii, 455 p. ISBN 9788576051879. [005.117 B261p 4. ed.]
    - **Capítulo 11**: Tratando erros
  - DEITEL, Paul J.; DEITEL, Harvey M. **Java, como programar**. 8. ed. São Paulo, SP: Pearson Prentice Hall, 2010. xxix, 1144 p. + 1 CD-ROM (4 3⁄4 pol.) ISBN 9788576055631. [005.133 D325j 8. ed.]
    - **Capítulo 11**: Tratamento de exceções