UNIVERSIDADE FEDERAL DO CEARÁ – UFC SOBRAL
TÉCNICAS DE PROGRAMAÇÃO – 2019.1 – PROF. WENDLEY

AULA PRÁTICA: 06
Manipulação de arquivos e exceções

www.ec.ufc.br/wendley

Para realizar os exercícios práticos adiante, utilize o programa NetBeans com Java.

PARTE 1 – MANIPULAÇÃO DE ARQUIVOS

EXERCÍCIO PRÁTICO – 1.1 – Criando arquivos txt usando FileWriter e PrintWriter

```
1  Import java.io.FileWriter; //adicionar pacote acima
2  Import java.io.IOException;
3  Import java.io.PrintWriter;
4  Import java.util.Scanner;
5
6  public class Arquivo {
7
8  public static void main(String[] args) throws IOException {
9      Scanner ler = new Scanner(System.in);
10     int i, n;
11
12     System.out.printf("Informe o número para a tabuada:\n");
13     n = ler.nextInt();
14
15     FileWriterarq = new FileWriter("c:\\Tabuada.txt"); //verificar permissão
16     PrintWriter gravarArq = new PrintWriter(arq);
17
18     gravarArq.printf("+--Resultado--+%n");
19     for (i=1; i<=10; i++) {
20         gravarArq.printf("| %2d X %d = %2d |%n", i, n, (i*n));
21     }
22     gravarArq.printf("+-------------+%n");
23
24     arq.close();
25
26     System.out.printf("\nTabuada do %d foi gravada com sucesso em
27 \"d:\\Tabuada.txt\".\n", n);
28     }
29 }
```

Faça os seguintes procedimentos:

1. Execute o programa e digite um valor de 1 a 9;
2. Verifique o conteúdo do arquivo;
3. Feche o arquivo txt;
4. Execute o programa e digite outro valor de 1 a 9 (diferente do primeiro);
5. Verifique o conteúdo do arquivo;
6. Feche o arquivo txt.

Onde está o arquivo da tabuada do primeiro valor digitado?

Altere a linha 15 para:

```
FileWriterarq = new FileWriter("c:\\Tabuada.txt", true);
```

Em seguida faça, novamente, os procedimentos anteriores (etapas 1 a 6). Qual a diferença?

EXERCÍCIO PRÁTICO – 1.2 – Lendo arquivos usando BufferedReader

```
1  importjava.io.BufferedReader;
2  importjava.io.BufferedWriter;
3  importjava.io.FileReader;
4  importjava.io.FileWriter;
5  importjava.io.IOException;
6  importjava.util.Scanner;
7
8  public class Principal {
9
10 public static void leitor(String path) throws IOException {
11     BufferedReaderbuffRead = new BufferedReader(new FileReader(path));
12     String linha = "";
13     while (true) {
14         if (linha != null) {
15             System.out.println(linha);
16
17         } else
18             break;
19         linha = buffRead.readLine();
20     }
21     buffRead.close();
22 }
23
24 public static void escritor(String path) throws IOException {
25     BufferedWriterbuffWrite = new BufferedWriter(new FileWriter(path));
26         String linha = "";
27         Scanner in = new Scanner(System.in);
28     System.out.println("Escreva algo: ");
29     linha = in.nextLine();
30     buffWrite.append(linha + "\n");
31     buffWrite.close();
32 }
33
34 public static void main(String args[]) throws IOException {
35     String path = "d:\\Arquivo2.txt";
36
37     escritor(path);
38     leitor(path);
39 }
40 }
```

Execute e analise o comportamento do programa.

PARTE 2 – EXCEÇÕES

Quando se cria programas de computador em Java, há possibilidade de ocorrer erros imprevistos durante sua execução, esses erros são conhecidos como exceções e podem ser provenientes de erros de lógica ou acesso a dispositivos ou arquivos externos.

Alguns possíveis motivos externos para ocorrer uma exceção são:

- Tentar abrir um arquivo que não existe;
- Tentar fazer consulta a um banco de dados que não está disponível;
- Tentar escrever algo em um arquivo sobre o qual não se tem permissão de escrita.

Alguns possíveis erros de lógica para ocorrer uma exceção são:

- Tentar manipular um objeto que está com o valor nulo;
- Dividir um número por zero;
- Tentar utilizar um método ou classe não existentes.

Uma forma de tentar contornar esses imprevistos é realizar o tratamento dos locais no código que podem vir a lançar possíveis exceções, como por exemplo, trechos em que há divisões, consulta a arquivos etc. Para tratar as exceções em Java utilizam-se os comandos ***try*** e ***catch***.

Sintaxe:

```
try
{
       //trecho de código que pode vir a lançar uma exceção
}
catch (tipo_exceçao_1 e)
{
       //ação a ser tomada
}
catch (tipo_exceçao_2 e)
{
       //ação a ser tomada
}
```

Exercício 2.1 – Um exemplo é o de abrir um arquivo para leitura, onde pode ocorrer o erro do arquivo não existir:

```
class Teste {
  public static void metodo() {
    new java.io.FileInputStream("arquivo123.txt");
  }
}
```

O código acima não compila e o compilador avisa que é necessário tratar o **FileNotFoundException** que pode ocorrer. Tente executar para verificar as mensagens de erro.

Para compilar e fazer o programa funcionar, há duas maneiras de tratar o problema: a primeira, é tratá-lo com o *try/catch* do mesmo jeito que usamos no exemplo anterior, com um *array*:

```
public static void metodo() {

  try {
    new java.io.FileInputStream("arquivo.txt");
  } catch (java.io.FileNotFoundException e) {
```

```
    System.out.println("Nao foi possível abrir o arquivo para leitura");
  }
}
```

A segunda forma de tratar esse erro é delegá-lo para quem chamou o nosso método, isto é, passar “para a frente”:

```
public static void metodo() throws java.io.FileNotFoundException {
   new java.io.FileInputStream("arquivo.txt");
}
```

Exercício <u>complementar</u> (fazer no laboratório se houver tempo) – Para aprendermos um pouco mais os conceitos básicos das *exceptions* do Java, teste o seguinte código:

```
1  public class TesteErro {
2    public static void main(String[] args) {
3      System.out.println("inicio do main");
4      metodo1();
5      System.out.println("fim do main");
6    }
7
8    static void metodo1() {
9      System.out.println("inicio do metodo1");
10     metodo2();
11     System.out.println("fim do metodo1");
12   }
13
14   static void metodo2() {
15     System.out.println("inicio do metodo2");
16     int[] array = new int[10];
17     for (int i = 0; i <= 15; i++) {
18       array[i] = i;
19       System.out.println(i);
20     }
21     System.out.println("fim do metodo2");
22   }
23 }
```

Execute o programa e observe bem a saída, você irá comparar com outras saídas adiante.

O sistema de exceções do Java funciona da seguinte maneira: quando uma exceção é lançada (*throw*), a JVM entra em estado de alerta e vai ver se o método atual toma alguma precaução ao tentar executar esse trecho de código. Como podemos observar, o metodo2 não toma nenhuma medida diferente do que vimos até agora.

Como o metodo2 não está tratando esse problema, a JVM para a execução dele anormalmente, sem esperar ele terminar, e volta um *stackframe* pra baixo, onde será feita nova verificação: "o metodo1 está se precavendo de um problema chamado **ArrayIndexOutOfBoundsException**?" "Não..." Volta para o main, onde também não há proteção, então a JVM é encerrada (na verdade, quem *encerra* é apenas a *Thread* corrente).

Aqui estamos forçando esse caso e não faria sentido tomarmos cuidado com ele. Seria fácil solucionar esse problema: bastaria percorrermos a *array* no máximo até o seu comprimento. Porém, apenas para entender o controle de fluxo de uma *Exception*, vamos colocar o código que vai tentar (*try*) executar o bloco perigoso e, caso o problema seja do tipo **ArrayIndexOutOfBoundsException**, ele será pego (*caught*).

Adicione um **try/catch** em volta do **for**, "pegando" **ArrayIndexOutOfBoundsException**, conforme trecho de código abaixo (altere o trecho que inicia na linha 17 do código anterior). O que o código imprime?

```
try {
  for (int i = 0; i <= 15; i++) {
    array[i] = i;
    System.out.println(i);
  }
} catch (ArrayIndexOutOfBoundsException e) {
  System.out.println("erro: " + e);
}
```

Execute o programa e observe bem a saída, comparando com a saída anterior.

Em vez de fazer o *try* em torno do for inteiro, tente apenas com o bloco de dentro do for:

```
for (int i = 0; i <= 15; i++) {
  try {
    array[i] = i;
    System.out.println(i);
  } catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("erro: " + e);
  }
}
```

Execute o programa e observe bem a saída, comparando com a saída anterior. **Quais as diferenças?**

Retire o *try/catch* e coloque-o em volta da chamada do metodo2, conforme abaixo:

```
System.out.println("inicio do metodo1");
try {
  metodo2();
} catch (ArrayIndexOutOfBoundsException e) {
  System.out.println("erro: " + e);
}
System.out.println("fim do metodo1");
```

Execute o programa e observe bem a saída, comparando com a saída anterior. **Quais as diferenças?**

Faça o mesmo retirando o try/catch novamente e colocando em volta da chamada do metodo1, conforme abaixo. Execute os códigos, **o que acontece?**

```
System.out.println("inicio do main");
try {
  metodo1();
} catch (ArrayIndexOutOfBoundsException e) {
  System.out.println("Erro : "+e);
}
System.out.println("fim do main");
```

Repare que, a partir do momento que uma *exception* foi *catched* (pega, tratada, *handled*), a execução volta ao normal a partir daquele ponto.

No início, existe uma grande tentação de sempre passar o problema para frente, para outros o tratarem. Pode ser que faça sentido, dependendo do caso, mas não até o *main*, por exemplo. Acontece que quem tenta abrir um arquivo sabe como lidar com um problema na leitura. Quem chamou um método no começo do programa pode não saber ou, pior ainda, tentar abrir cinco arquivos diferentes e não saber qual deles teve um problema! Não há uma regra para decidir em que momento do seu programa você vai tratar determinada exceção. Analise com atenção onde tratar a exceção.

Adaptado de: https://www.caelum.com.br/apostila-java-orientacao-objetos/excecoes-e-controle-de-erros/#11-4-outro-tipo-de-excecao-checked-exceptions e http://www.devmedia.com.br/tratando-excecoes-em-java/25514