

Entrada e Saída e Tratamento de Exceções

Roland Teodorowitsch

Fundamentos de Programação - Escola Politécnica - PUCRS

4 de maio de 2023

Objetivos

- Ler e escrever arquivos de texto
- Processar argumentos da linha de comando
- Lançar e capturar exceções
- Implementar programas que propagam exceções verificadas

Conteúdos

- Leitura e Escrita de Arquivos de Texto
- Entrada e Saída de Texto
- Argumentos da Linha de Comando
- Tratamento de Exceções
- Aplicação: Tratando Erros de Entrada

Lendo e Escrevendo Arquivos de Texto

- Arquivos de texto são muito usados para armazenar informações
 - Tanto números quanto palavras podem ser armazenados como texto
 - Arquivos de texto são o tipo de arquivos de dados mais portátil que existe
- A classe `Scanner` pode ser usada para ler arquivos de texto
 - Ela também é usada para entrada de dados via teclado
 - Ler de um arquivo exige a classe `File`
- A classe `PrintWriter` pode ser usada para escrever arquivos de texto
 - Ela permite usar métodos familiares como `print()`, `println()` e `printf()`

Entrada de Arquivo Texto

- Cria-se um objeto da classe `File`
 - Passa-se para ele o nome do arquivo entre aspas

```
File inputFile = new File("input.txt");
```

- Então cria-se um objeto da classe `Scanner`
 - Passa-se para ele o novo objeto `File`

```
Scanner in = new Scanner(inputFile);
```

- Então usa-se os métodos de `Scanner`, tais como:
 - `next()`, `nextLine()`, `hasNextLine()`, `hasNext()`, `nextDouble()`, `nextInt()`, etc.

```
while (in.hasNextLine()) {  
    String line = in.nextLine();  
    // Process line;  
}
```

Saída de Arquivo Texto

- Cria-se um objeto da classe `PrintWriter`
 - Passa-se para ele o nome do arquivo que deve ser escrito entre aspas

```
PrintWriter out = new PrintWriter("output.txt");
```

- Se `output.txt` existir, ele será “esvaziado”
 - Se `output.txt` não existir, ele será criado vazio
- `PrintWriter` é uma versão melhorada de `PrintStream`
- `System.out` é um objeto `PrintStream`

```
System.out.println("Hello World!");
```

- `print()`, `println()` e `printf()` são métodos herdados que podem ser usados para escrever em arquivos:

```
out.println("Hello, World!");  
out.printf("Total: %8.2f\n", totalPrice);
```

Fechando Arquivos

- O método `close()` deve ser chamado depois que a leitura ou escrita estiverem completos
 - Fechando um `Scanner`

```
while (in.hasNextLine()) {  
    String line = in.nextLine();  
    // Process line;  
}  
in.close();
```

- Fechando um `PrintWriter`

```
out.println("Hello, World!");  
out.printf("Total: %8.2f\n", totalPrice);  
out.close();
```

- O conteúdo escrito em um arquivo pode não ter sido definitivamente salvo em disco até `close()` ser chamado

Prévia sobre Exceções

- Há uma situação extraordinária que precisa ser considerada:
 - Se o arquivo de entrada ou saída para um `Scanner` não existir, ocorrerá uma exceção chamada `FileNotFoundException` quando o objeto `Scanner` for construído
 - O construtor `PrintWriter` também pode gerar uma exceção se não for possível abrir o arquivo para escrita
 - Se o nome for ilegal ou se o usuário não tiver permissão para criar o arquivo em determinada localização
- Até que se saiba como tratar exceções, deve-se adicionar duas palavras a qualquer método que use E/S de arquivo

```
public static void main(String[] args) throws FileNotFoundException
```


Algumas diretivas `import` necessárias

- Para usar E/S de arquivo e suas exceções, algumas diretivas `import` do pacote `java.io` deverão ser usadas

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

public class LineNumberer {
    public void openFile() throws FileNotFoundException {
        // ...
    }
}
```

Total.java (HORSTMANN, 2013, p. 319-320)

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

/**
   This program reads a file with numbers, and writes the numbers to another
   file, lined up in a column and followed by their total.
 */
public class Total {
    public static void main(String[] args) throws FileNotFoundException {
        // Prompt for the input and output file names
        Scanner console = new Scanner(System.in);
        System.out.print("Input file: ");
        String inputFileName = console.next();
        System.out.print("Output file: ");
        String outputFileName = console.next();
    }
}
```

Total.java (HORSTMANN, 2013, p. 319-320)

```
// Construct the Scanner and PrintWriter objects for reading and writing
File inputFile = new File(inputFileName);
Scanner in = new Scanner(inputFile);
PrintWriter out = new PrintWriter(outputFileName);
// Read the input and write the output
double total = 0;
while (in.hasNextDouble()) {
    double value = in.nextDouble();
    out.printf("%15.2f\n", value);
    total = total + value;
}
out.printf("Total: %8.2f\n", total);
in.close();
out.close();
}
```

Erros Comuns

Contra-barras em nomes de arquivos:

- Quando se usa uma *string* constante como nome de arquivo com informação sobre o caminho, deve-se usar contra-barras duplas:

```
File inputFile = new File("c:\\homework\\input.dat");
```

- Uma contra-barra única em um *string* corresponde a um caractere de escape, o que significa que o próximo caractere será interpretado de forma diferente (por exemplo, '`\n`' para o caractere de nova linha)
- Porém quando o usuário fornece um nome de arquivo com caminho a partir do terminal, não é preciso digitar contra-barras duplas

Erros Comuns (2)

Construindo um `Scanner` com um *string*:

- Quando se constrói um `PrintWriter` com um *string*, ele escreve neste arquivo:

```
PrintWriter out = new PrintWriter("output.txt");
```

- Isto não funciona com um objeto `Scanner`:

```
Scanner in = new Scanner("input.txt"); // Error?
```

- Isto **não** abre o arquivo. Em vez disto, simplesmente se faz a leitura a partir do *string* especificado
- Para ler de um arquivo, passa-se para `Scanner` um objeto `File`:

```
Scanner in = new Scanner(new File ("input.txt") );
```

- Ou

```
File myFile = new File("input.txt");  
Scanner in = new Scanner(myFile);
```

Entrada e Saída de Texto

- Nas seções a seguir, serão apresentados exemplos de processamento de texto com conteúdos complexos, de forma que se possa lidar com variações que frequentemente ocorrem com dados reais
- O processamento da entrada será necessário em praticamente todos os programas que interagem com o usuário
- Eventualmente pode ser necessário ler a entrada palavra por palavra, linha por linha, número por número ou caractere por caractere
- As classes `Scanner` e `String` fornecem métodos que combinados (com alguma experiência) podem tratar cada situação

Lendo Linhas ou Palavras

- Para ler uma **linha** de cada vez, usa-se um laço com `hasNextLine()` para testar se há uma linha para ser lida e `nextLine()` para ler a linha

```
while (in.hasNextLine()) {  
    String input = in.nextLine();  
    System.out.println(input);  
}
```

- Para ler uma **palavra** de cada vez, usa-se um laço com `hasNext()` para testar se há uma palavra para ser lida e `next()` para ler a palavra

```
while (in.hasNext()) {  
    String input = in.next();  
    System.out.println(input);  
}
```

Espaços em Branco

- O método `next()` de `Scanner` precisa decidir onde inicia e onde termina uma palavra
- São usadas regras simples para isto:
 - São consumidos todos os espaços em branco antes do primeiro caractere
 - Então ele lê caracteres até encontrar um espaço em branco ou chegar até o final da entrada
- Espaços em branco podem ser:
 - `' '` (espaço)
 - `'\n'` (nova linha)
 - `'\r'` (retorno do carro)
 - `'\t'` (tabulação)
 - `'\f'` (avanço de formulário)

O Método `useDelimiter()`

- A classe `Scanner` tem um método para alterar o conjunto padrão de delimitadores usado para separar palavras
- O método `useDelimiter()` recebe uma *string* que lista todos os caracteres que podem ser usados como delimitadores
- Esta *string* pode conter uma expressão regular para especificar casos de forma mais completa

```
Scanner in = new Scanner(...);  
in.useDelimiter("[^A-Za-z]+");
```

- `[^A-Za-z]+` diz que qualquer conjunto (`[]`) de pelo menos um (`+`) caractere que não (`^`) seja uma letra maiúscula (`A-Z`) ou minúscula (`a-z`) será usado como delimitador
- Expressões regulares são usadas em muitos outros contextos
- Pode-se obter maiores detalhes sobre expressões regulares na Internet

Lendo Caracteres

- Em `Scanner` **não** existem os métodos `hasNextChar()` e `nextChar()`
 - Em vez disto, deve-se definir `Scanner` para usar um delimitador vazio

```
Scanner in = new Scanner(...);
in.useDelimiter("");
while (in.hasNext()) {
    char ch = in.next().charAt(0);
    // Processa cada caractere
}
```

- `next()` retorna uma *string* de um caractere
- Usa-se `charAt(0)` para extrair o caractere no índice 0

Classificando Caracteres

- A classe `Character` provê diversos métodos úteis para classificar um caractere
- Passa-se para eles um `char` e eles retornam um `boolean`

```
if ( Character.isDigit(ch) ) ...
```

Método	Exemplos de caracteres aceitos
<code>isDigit()</code>	0, 1, 2
<code>isLetter()</code>	A, B, C, a, b, c
<code>isUpperCase()</code>	A, B, C
<code>isLowerCase()</code>	a, b, c
<code>isWhiteSpace()</code>	espaço, nova linha, tabulação

Lendo Linhas

- Alguns arquivos de texto são usados como bancos de dados simples
 - Cada linha contém um conjunto de pedaços de informação relacionados
 - Por exemplo:
China 1330044605
India 1147995898
United States 303824646
 - Neste exemplo, o fato de que há nomes de países formados por duas palavras, torna a leitura mais complicada
 - Será melhor ler a linha inteira e processá-la usando os métodos da classe `String`
- `nextLine()` lê a linha, eliminando o '`\n`' do final da linha

```
while (in.hasNextLine()) {  
    String line = in.nextLine();  
    // Processa cada linha  
}
```

- | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U | n | i | t | e | d | | S | t | a | t | e | s | | 3 | 0 | 3 | 8 | 2 | 4 | 6 | 4 | 6 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
- countryName population

- ```
int i = 0;
while (!Character.isDigit(line.charAt(i))) { i++; }
```

## Quebrando cada linha (2)

- O método `substring` é usado para extrair as duas partes
- E `trim()` remove espaços em branco no início e no final da *string*

```
String countryName = line.substring(0, i);
String population = line.substring(i);
// remove os espaços em branco no início e fim do nome do país
countryName = countryName.trim();
```

## Outra Forma Usando Métodos de Scanner

- Uma outra alternativa para o problema anterior poderia ser:
  - Ler a linha e armazená-la em uma variável `String`
  - Passar a *string* para um novo objeto `Scanner`
  - Neste caso, `hasNextInt()` poderá ser usado para encontrar números
  - Enquanto não for um número, usa-se `next()` para ler e concatena-se o que for lido

```
Scanner lineScanner = new Scanner(line);
String countryName = lineScanner.next();
while (!lineScanner.hasNextInt()) {
 countryName = countryName + " " + lineScanner.next();
}
```

# Convertendo *Strings* para Números

- *Strings* contém dígitos e não números
- Há métodos para fazer a conversão:
  - Para um valor inteiro:

```
String pop = "303824646";
int populationValue = Integer.parseInt(pop);
```

- Para um valor real:

```
String priceString = "3.95";
double price = Double.parseDouble(priceString);
```

- A *string* a ser convertida deve conter apenas caracteres válidos: nem mesmo espaços são permitidos
- Por isso, convém usar `trim()` antes da conversão:

```
int populationValue = Integer.parseInt(pop.trim());
```



# Lendo Números com Segurança

- Se a entrada não estiver corretamente formatada, os métodos `nextInt()` e `nextDouble()` irão gerar uma exceção `InputMismatchException`
- Os métodos `hasNextInt()` e `hasNextDouble()` podem ser usados, respectivamente, para verificar antecipadamente se a entrada é válida ou não

```
if (in.hasNextInt()) {
 int value = in.nextInt(); // seguro
}
```

- Se `hasNextInt()` e `hasNextDouble()` retornarem `true`, a respectiva leitura pode ser feita sem problema
- Se o resultado for `false`, deve-se evitar fazer a leitura

# Lendo Outros Tipos de Números

- A classe `Scanner` tem métodos para teste e leitura de quase todos os tipos primitivos de dados

| Tipo de dado         | Método de teste               | Método de Leitura          |
|----------------------|-------------------------------|----------------------------|
| <code>byte</code>    | <code>hasNextByte()</code>    | <code>nextByte()</code>    |
| <code>short</code>   | <code>hasNextShort()</code>   | <code>nextShort()</code>   |
| <code>int</code>     | <code>hasNextInt()</code>     | <code>nextInt()</code>     |
| <code>long</code>    | <code>hasNextLong()</code>    | <code>nextLong()</code>    |
| <code>float</code>   | <code>hasNextFloat()</code>   | <code>nextFloat()</code>   |
| <code>double</code>  | <code>hasNextDouble()</code>  | <code>nextDouble()</code>  |
| <code>boolean</code> | <code>hasNextBoolean()</code> | <code>nextBoolean()</code> |

- É importante lembrar que não há métodos semelhantes para o tipo `char`

# Misturando a leitura de números, palavras e linhas

- `nextDouble()` e `nextInt()` não consomem espaços em branco após números
- Isto pode causar um comportamento inesperado se `nextLine()` for chamado após a leitura de um número no final da linha:
  - `nextLine()` lerá uma *string* vazia ou uma *string* com um espaço em branco
- O melhor é não misturar as leituras ou ainda usar algum formato de armazenamento mais consistente, como, por exemplo, CSV (*Comma-Separated Values*)

# CSV (*Comma-Separated Values*)

- Trata-se de um formato de arquivo onde cada linha (terminada por ' \n' ) corresponde a um registro formado por campos, geralmente, separados por vírgula ( ' , ' )
- Outros separadores de campos (como ' ; ' ou ' : ' ) também costumam ser usados
- Por exemplo, um arquivo CSV poderia conter:

```
China,1330044605
India,1147995898
United States,303824646
```

- Para a leitura deste arquivo, pode-se usar o método `split()`:

```
while (in.hasNextLine()) {
 String linha = in.nextLine();
 String[] campos = linha.split(",");
 for (int i=0;i<campos.length;++i)
 System.out.println(campos[i]);
}
```

# Formatando a Saída

- `System.out.printf()` disponibiliza muitas opções de formatação, permitindo:
  - Alinhamento de *strings* e números
  - Definição do tamanho de exibição de campos
  - Alinhamento à esquerda (o padrão é fazer o alinhamento à direita)
- Por exemplo:

```
System.out.printf("%-10s%10.2f", items[i] + ":", prices[i]);
```

- `%-10s` faz o alinhamento à esquerda de um *string* completando-o com espaços até que 10 caracteres tenham sido impressos
  - `%10.2f` imprime um valor decimal de ponto flutuante, alinhado à direita, com 10 caracteres no total, contando 2 casas decimais
- Resultado:

Altura:                   1.82

# Especificação de Formato de `printf()`

- A especificação de formato de `printf()` tem a seguinte estrutura:
  - O primeiro caractere é um `%`
  - A seguir, aparecem *flags* opcionais – veja tabela a seguir
  - Depois aparece o tamanho do campo: número total de caracteres (incluindo espaços para alinhamento), seguido da precisão para números de ponto flutuante (que é opcional)
  - Por fim, aparece o tipo de formatação (por exemplo, `f` para valores em ponto flutuante ou `s` para *strings*) – veja a tabela a seguir

# Flags de Formatação para `printf()`

| <b>Flag</b> | <b>Significado</b>                       | <b>Exemplo</b>          |
|-------------|------------------------------------------|-------------------------|
| -           | Alinhamento à esquerda                   | 1.23 seguido de espaços |
| 0           | Preenchimento com zeros                  | 001.23                  |
| +           | Mostra o sinal para números positivos    | +1.23                   |
| (           | Envolve números negativos com parênteses | (1.23)                  |
| ,           | Mostra separador para milhares           | 12,300                  |
| ^           | Converte letras para maiúsculas          | 1.23E+1                 |

# Tipos de Formatação para `printf()`

| Código | Tipo                                                                                                         | Exemplo |
|--------|--------------------------------------------------------------------------------------------------------------|---------|
| d      | Inteiro decimal                                                                                              | 123     |
| f      | Ponto flutuante fixo                                                                                         | 12.30   |
| e      | Ponto flutuante exponencial                                                                                  | 1.23e+1 |
| g      | Ponto flutuante geral (a notação exponencial será usada apenas para valores muito grandes ou muito pequenos) | 12.3    |
| s      | <i>String</i>                                                                                                | Tax:    |



# Argumentos da Linha de Comando

- Pode-se parametrizar a execução de programas através de argumentos da linha de comandos
- Por exemplo, nomes de arquivos e opções são frequentemente digitados após o nome do programa no *prompt* de comandos:

```
java ProgramClass -v input.dat
```

- Java provê acesso a estas informações através de um *array* de *strings* chamado `args` passado como parâmetro para o método `main`:

```
public static void main(String[] args)
```

- Cada elemento de `args` corresponde a uma palavra digitada na linha de comandos após o nome do programa
- `args.length` contém, portanto, o número de argumentos

```
args[0] = "-v"
args[1] = "input.dat"
args.length = 2
```

- Tradicionalmente opções costuma iniciar com um sinal de menos ('-')

# Exemplo: Cifra de César

- Escreva um programa de linha de comando que use a Cifra de César (um algoritmo de criptografia baseado em substituição de caracteres) para:

- Criptografar um arquivo, sendo fornecidos nomes de arquivos de entrada e de saída

```
java CaesarCipher input.txt encrypt.txt
```

- Descriptografar um arquivo, fornecido como parâmetro de uma opção

```
java CaesarCipher -d encrypt.txt output.txt
```

|                |   |   |   |   |  |   |   |  |   |   |  |   |   |   |  |  |
|----------------|---|---|---|---|--|---|---|--|---|---|--|---|---|---|--|--|
| Plain text     | M | e | e | t |  | m | e |  | a | t |  | t | h | e |  |  |
|                | ↓ | ↓ | ↓ | ↓ |  | ↓ | ↓ |  | ↓ | ↓ |  | ↓ | ↓ | ↓ |  |  |
| Encrypted text | P | h | h | w |  | p | h |  | d | w |  | w | k | h |  |  |

## CaesarCipher.java (HORSTMANN, 2013, p. 331-333)

```

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

/**
 * This program encrypts a file using the Caesar cipher.
 */
public class CaesarCipher {
 public static void main(String[] args) throws FileNotFoundException {
 final int DEFAULT_KEY = 3;
 int key = DEFAULT_KEY;
 String inFile = ""; String outFile = "";
 int files = 0; // Number of command line arguments that are files
 for (int i = 0; i < args.length; i++) {
 String arg = args[i];
 if (arg.charAt(0) == '-') {
 // It is a command line option
 char option = arg.charAt(1);
 if (option == 'd') { key = -key; } else { usage(); return; }
 }
 else {
 // It is a file name
 files++;
 if (files == 1) { inFile = arg; } else if (files == 2) { outFile = arg; }
 }
 }
 }
}

```

# CaesarCipher.java (HORSTMANN, 2013, p. 331-333)

```
if (files != 2) { usage(); return; }
Scanner in = new Scanner(new File(inFile));
in.useDelimiter(""); // Process individual characters
PrintWriter out = new PrintWriter(outFile);
while (in.hasNext()) {
 char from = in.next().charAt(0);
 char to = encrypt(from, key);
 out.print(to);
}
in.close();
out.close();
}

/**
 * Prints a message describing proper usage.
 */
public static void usage() {
 System.out.println("Usage: java CaesarCipher [-d] infile outfile");
}
```

# CaesarCipher.java (HORSTMANN, 2013, p. 331-333)

```
/**
 * Encrypts upper- and lowercase characters by shifting them according to a key.
 * @param ch the letter to be encrypted
 * @param key the encryption key
 * @return the encrypted letter
 */
public static char encrypt(char ch, int key) {
 int base = 0;
 if ('A' <= ch && ch <= 'Z') { base = 'A'; }
 else if ('a' <= ch && ch <= 'z') { base = 'a'; }
 else { return ch; } // Not a letter
 int offset = ch - base + key;
 final int LETTERS = 26; // Number of letters in the Roman alphabet
 if (offset > LETTERS) { offset = offset - LETTERS; }
 else if (offset < 0) { offset = offset + LETTERS; }
 return (char) (base + offset);
}
```

# Passos para o Processamento de Arquivos de Texto

- Considere o seguinte problema:
  - Ler um arquivo com nomes de países e a população (`worldpop.txt`)
  - Ler um arquivo com nomes de países e a área de cada país (`worldarea.txt`)
  - Escrever um arquivo (`world_pop_density.txt`) que contenha os nomes dos países e sua densidade populacional (com nomes de países alinhados à esquerda e números alinhados à direita)

## Passos para o Processamento de Arquivos de Texto (2)

- Passos:

- 1 Entender a tarefa de processamento: os dados serão lidos e processados, ou serão armazenados e processados?
- 2 Determinar os arquivos de entrada: Qual o formato dos arquivos de entradas? Há exatamente os mesmos nomes de países nos dois arquivos ou é preciso cruzar informações?
- 3 Escolher como os nomes dos arquivos serão obtidos
- 4 Escolher a forma de leitura (linha, palavra ou caractere) – se todos os dados estão em uma linha, normalmente usa-se leitura de linhas
- 5 Considerando a leitura orientada à linha, é preciso determinar como os dados necessários serão extraídos da linha: Quais são os delimitadores?
- 6 Usar métodos para executar tarefas comuns

# Pseudocódigo para o Problema da Densidade Populacional

- Enquanto há linhas para serem lidas
  - Ler uma linha de cada arquivo
  - Extrair o nome do país
  - População = número após o Nome do país na linha do primeiro arquivo
  - Área = número após o Nome do país na linha do segundo arquivo
  - se área for diferente de zero,  $Densidade = População / Área$
  - Imprimir Nome de país e a densidade



# Tratamento de Exceções

- Há duas formas de tratar erros em tempo de execução:
  - Detectar os erros
    - É a forma mais simples
    - Pode-se usar o comando `throw` para sinalizar (“lançar”) uma exceção
  - Tratar os erros
    - Esta forma é mais complexa
    - É preciso capturar (`catch`) cada exceção possível e reagir a cada uma delas de forma apropriada
- O tratamento de erros pode ser feito de forma:
  - Simples: encerrando o programa
  - Amigável: solicitando que o usuário corrija o erro

# Sintaxe do Lançamento de Exceções

- Lançar uma exceção consiste em lançar um objeto da classe `Exception`
  - Deve-se escolher de forma cuidadosa
  - Pode-se passar uma *string* descrevendo o problema para a maioria dos objetos para exceções
  - Quando se lança uma exceção, o fluxo de controle normal é encerrado

```
if (amount > balance)
{
 throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

A new exception object is constructed, then thrown.

Most exception objects can be constructed with an error message.

This line is not executed when the exception is thrown.



# Capturando Exceções

- Exceções que são lançadas devem ser “capturadas” em algum lugar do programa
  - Chamadas de métodos que podem lançar exceções devem estar dentro de um bloco `try`
  - Usa-se blocos `catch` para cada exceção possível
    - Costuma-se chamar o parâmetro para a exceção no bloco `catch` como `e` ou `exception`

```
try {
 Scanner in = new Scanner(new File("nome.txt")); // <== FileNotFoundException (V)
 String input = in.next(); // <== NoSuchElementException
 int value = Integer.parseInt(input); // <== NumberFormatException
 // ...
}
catch (IOException exception) {
 exception.printStackTrace();
}
catch (NumberFormatException exception) {
 System.out.println(exception.getMessage());
}
```

# Capturando Exceções

- Quando uma exceção é capturada, a execução é desviada imediatamente para o primeiro bloco `catch` que corresponda à exceção
  - `IOException` captura a exceção `FileNotFoundException` (na hierarquia de classes, `FileNotFoundException` é descendente de `IOException`)
  - `NumberFormatException` é capturada pelo segundo bloco `catch`
  - `NoSuchElementException` não é capturada

```
try {
 Scanner in = new Scanner(new File("nome.txt")); // <= FileNotFoundException (V)
 String input = in.next(); // <= NoSuchElementException
 int value = Integer.parseInt(input); // <= NumberFormatException
 // ...
}
catch (IOException exception) {
 exception.printStackTrace();
}
catch (NumberFormatException exception) {
 System.out.println(exception.getMessage());
}
```

# Tratando Exceções

- Algumas opções de tratamento de exceções poderiam ser:
  - Simplesmente informar ao usuário o que está errado
  - Dar ao usuário outra chance de corrigir um erro de entrada
  - Imprimir a “pilha de chamadas”, que mostra a lista de métodos chamados

```
exception.printStackTrace();
```

# Sintaxe da Captura de Exceções

```
try
{
 Scanner in = new Scanner(new File("input.txt"));
 String input = in.next();
 process(input);
}
catch (IOException exception)
{
 System.out.println("Could not open input file");
}
catch (Exception except)
{
 System.out.println(except.getMessage());
}
```

This constructor can throw a `FileNotFoundException`.

This is the exception that was thrown.

A `FileNotFoundException` is a special case of an `IOException`.

When an `IOException` is thrown, execution resumes here.

Additional catch clauses can appear here. Place more specific exceptions before more general ones.

# Exceções Verificadas

- `Throw/catch` é aplicado a três tipos de exceções:
  - Erros: erros internos (não considerados nesta disciplina)
  - Não verificadas: exceções de execução causadas pelo programador (o compilador não verifica se o programador está tratando elas)
  - Verificados: todas as outras exceções, que estão associadas a situações que o programador não pode prevenir (o compilador verifica se elas estão sendo tratadas)



# Sintaxe da Cláusula `throws`

- Métodos que usam outros métodos que podem lançar exceções devem declarar que lançam estas exceções (se elas não forem tratadas)
  - Declaram-se as exceções verificadas que o método lança (*throws*) e
  - Pode-se também declarar exceções não verificadas
- Se o método tratar internamente a exceção verificada, ela não precisará ser declarada na cláusula `throws`
- A declaração de exceções na cláusula `throws` faz com que a exceção seja passada para o método chamador (que deverá capturá-la ou, por sua vez, passá-la adiante)

```
public static String readData(String filename)
 throws FileNotFoundException, NumberFormatException
```

You **must** specify all checked exceptions that this method may throw.

You may also list unchecked exceptions.

# A Cláusula `finally`

- `finally` é uma cláusula opcional em um bloco `try/catch`
- É usada quando for necessário tomar alguma ação no método quando uma exceção tiver sido lançada ou não
- O bloco `finally` é executado em ambos os casos
- Por exemplo: fechar um arquivo no método em todos os casos

```
public void printOutput(String filename) throws IOException {
 PrintWriter out = new PrintWriter(filename);
 try {
 writeData(out); // Method may throw an I/O Exception
 }
 finally {
 out.close();
 }
}
```

# Sintaxe da Cláusula `finally`

- O código no bloco `finally` será executado sempre que a execução entrar no bloco `try`

This variable must be declared outside the `try` block so that the `finally` clause can access it.

This code may throw exceptions.

```
PrintWriter out = new PrintWriter(filename);
```

```
try
```

```
{
```

```
 writeData(out);
```

```
}
```

```
finally
```

```
{
```

```
 out.close();
```

```
}
```

This code is always executed, even if an exception occurs.

# Dicas de Programação

- Lançar Antes

- Quando um método detecta um problema que ele não pode resolver, é melhor lançar uma exceção do que tentar encontrar uma correção imperfeita

- Capturar Depois

- Reciprocamente, um método somente deveria capturar uma exceção se ele realmente puder resolver a situação
- Caso contrário, a melhor solução é simplesmente propagar a exceção para o seu chamador, permitindo que ela seja capturada por um tratador mais capaz

## Dicas de Programação (2)

- Não silencie exceções

- Quando se chama um método que lança uma exceção verificada e nenhum tratador foi especificado, o compilador reclama
- É tentador escrever um bloco `try/catch` vazio para silenciar o compilador e retornar ao código posteriormente, porém isto é uma **péssima ideia!**
- Exceções foram projetadas para transmitir informações sobre problemas para tratadores capazes
- Instalar um tratador incapaz simplesmente esconde uma condição de erro que pode ser séria

## Dicas de Programação (3)

- Não use blocos `catch` e `finally` no mesmo bloco `try`
  - A cláusula `finally` é executada independentemente se o bloco `try` é encerrado de alguma das seguintes formas:
    - 1 Após o término do último comando de um bloco `try`
    - 2 Após o término do último comando de um bloco `catch`, se o respectivo bloco `try` capturar uma exceção
    - 3 Quando uma exceção for lançada em um bloco `try` e não for capturada
  - É melhor usar dois blocos `try` aninhados para controlar o fluxo

```
try {
 PrintWriter out = new PrintWriter(filename);
 try { /* Escrever Saida */ }
 finally { out.close(); } // Fechar recursos
}
catch (IOException exception) {
 // Trata excecao
}
```

# Tratando Erros de Entrada

- Exemplo de aplicação: ler um arquivo com dados numéricos
  - A primeira linha contém o número de valores
  - As demais linhas contêm os outros valores (reais)
- Riscos:
  - O arquivo pode não existir: o construtor de `Scanner` lançará a exceção `FileNotFoundException`
  - O arquivo pode estar no formato errado
    - Não começa com um contador: `NoSuchElementException`
    - Há muitos itens: `IOException`

## Exemplo de Arquivo com Dados de Entrada

```
3
1.45
-2.1
0.05
```

# Tratando Erros de Entrada: `main()`

- Estrutura de código para tratar todas as exceções

```
boolean done = false;
while (!done) {
 try {
 // Pergunta o nome do arquivo ao usuario...
 double[] data = readFile(filename); // Pode lancar excecoes
 // Processa os dados...
 done = true;
 }
 catch (FileNotFoundException exception) {
 System.out.println("File not found.");
 }
 catch (NoSuchElementException exception) {
 System.out.println("File contents invalid.");
 }
 catch (IOException exception) {
 exception.printStackTrace();
 }
}
```



# Tratando Erros de Entrada: `readFile()`

- Chama o construtor para `Scanner`
- Não possui tratamento de exceções (nenhuma cláusula `catch`)
- A cláusula `finally` fecha o arquivo em qualquer caso (se houver exceção ou não)
- `IOException` é lançado de volta para `main()`

```
public static double[] readFile(String filename) throws IOException {
 File inFile = new File(filename);
 Scanner in = new Scanner(inFile);
 try {
 return readData(in); // Pode lancar excecoes
 }
 finally {
 in.close();
 }
}
```

# Tratando Erros de Entrada: `readData()`

- Nenhum tratamento de exceção (nenhuma cláusula `try` ou `catch`)
- `throw` cria um objeto `IOException` e sai
- A exceção não verificada `NoSuchElementException` pode ocorrer

```
public static double[] readData(Scanner in) throws IOException {
 int numberOfValues = in.nextInt(); // NoSuchElementException
 double[] data = new double[numberOfValues];
 for (int i = 0; i < numberOfValues; i++) {
 data[i] = in.nextDouble(); // NoSuchElementException
 }
 if (in.hasNext()) {
 throw new IOException("End of file expected");
 }
 return data;
}
```

## Sumário: Entrada/Saída

- Usa-se as classes `File` e `Scanner` para ler arquivos de texto
- Para escrever arquivos de texto, usa-se a classe `PrintWriter` e os métodos `print()`, `println()` e `printf()`
- Deve-se fechar todos os arquivos depois de terminar de usá-los usando o método `close()`
- Programas iniciados a partir da linha de comandos recebem argumentos da linha de comandos através de argumentos do método `main()`

# Sumário: Processando arquivos de Texto

- O método `next()` lê uma *string* que é delimitado por espaços em branco
- A classe `Character` tem métodos para classificação de caracteres
- O método `nextLine()` lê uma linha inteira
- Se uma *string* contém os dígitos de um número, pode-se usar os métodos `Integer.parseInt()` ou `Double.parseDouble()` para obter os respectivos valores numéricos

## Sumário: Exceções (1)

- Para sinalizar uma condição especial, usa-se a sentença `throw` para lançar uma exceção em um objeto
- Quando se lança uma exceção, o processamento continua em um tratador de exceções
- Coloca-se sentenças que podem causar exceções dentro de um bloco `try`, e o tratador dentro de uma cláusula `catch`
- Exceções verificadas ocorrem devido a circunstâncias externas das quais o programador não pode se prevenir
- O compilador verifica se o programa trata estas exceções

## Sumário: Exceções (2)

- Aciciona-se uma cláusula `throws` em um método que pode lançar uma exceção verificada
- Se a execução entrar em um bloco `try`, as sentenças na cláusula `finally` são garantidamente executadas, independentemente se uma exceção for lançada ou não
- Lança-se uma exceção tão logo um problema seja detectado
- Captura-se uma exceção apenas se o problema pode ser tratado
- Quando se projeta um programa, deve-se perguntar que tipo de exceções podem ocorrer
- Para cada exceção, deve-se decidir que parte do programa pode tratá-la da melhor forma

# Referências

HORSTMANN, C. **Java for Everyone – Late Objects**. 2. ed. Hoboken: Wiley, 2013. xxxiv, 589 p.