Linguagem de Programação I

Aula 7
Tratamento de Exceções

l.bertholdo@ifsp.edu.br

Conteúdo

- Tratamento de Exceções
  - Classes de Exceção
  - Try e Catch
  - Multi-Catch
  - Finally
  - Throw/Throws

l.bertholdo@ifsp.edu.br

Tratamento de Exceções

- Durante o uso de um programa, podem ocorrer alterações em seu fluxo normal de execução, que ocasionam erros como travamentos e emissão de mensagens do sistema operacional.
- Estas alterações ocorrem por vários motivos: entrada de argumentos inválidos, acesso indevido a locais de memória, indisponibilidade de algum recurso do qual a aplicação depende, etc.
- Por isso, as aplicações devem ter a capacidade de detectar e tratar erros de forma adequada, a fim de minimizar o problema para o usuário.
- Assim como outras linguagens, Java tem um mecanismo de **tratamento de exceções** para sinalizar e tratar erros durante a execução de um programa.

l.bertholdo@ifsp.edu.br

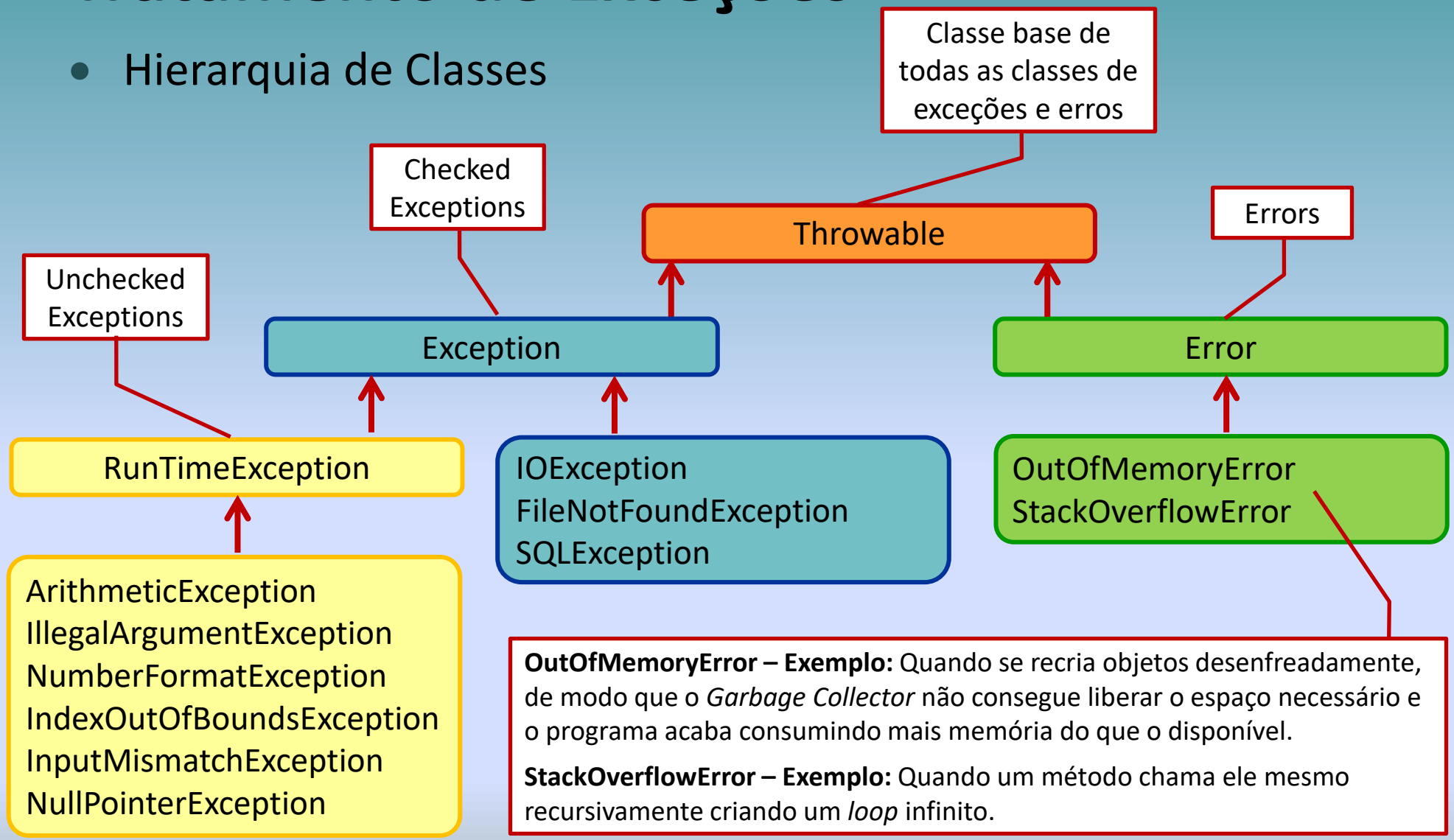
Tratamento de Exceções

- **Errors:** Erros não monitorados pelo compilador. Envolvem eventos irrecuperáveis, que impedem a execução da aplicação e não podem ser tratados no código do programa.
- Tipos de Exceções:
  - **Checked Exceptions:** Exceções monitoradas pelo compilador. Envolvem falhas em recursos externos à aplicação, que podem ser tratadas pelo desenvoledor.
  - **Unchecked Exceptions:** Exceções não monitoradas pelo compilador. Envolvem erros de lógica da própria aplicação, que também podem ser tratados pelo desenvoledor.

l.bertholdo@ifsp.edu.br

Tratamento de Exceções

- Hierarquia de Classes



* A hierarquia de classes completa pode ser encontrada em:
https://docs.oracle.com/javase/8/docs/api/overview-tree.html

Classes de Exceção

- Exemplos

Classe de Exceção	Motivo de Lançamento
IOException	Quando ocorre algum tipo de erro em operações de entrada e saída de dados.
FileNotFoundException	Quando se tenta acessar um arquivo que não existe no local especificado.
SqlException	Quando ocorre erro ao tentar acessar um banco de dados.
ArithmeticException	Para erros em operações aritméticas, como divisão por zero, por exemplo.
IllegalArgumentException	Quando um dos argumentos apresentados a um método não é válido.
NumberFormatException	Quando se tenta converter uma cadeia de caracteres para um formato numérico.
IndexOutOfBounds Exception	Quando se tenta acessar um elemento de um *array* com um índice que está fora de seus limites.
InputMismatchException	Quando um método da classe Scanner recebe uma cadeia de caracteres que não representa um tipo de dado válido.
NullPointerException	Quando uma referência null é utilizada onde um valor é esperado.

Classes de Exceção

- Alguns métodos das classes de exceção

Métodos	Descrição
getMessage	Retorna uma breve descrição do erro que provocou a exceção.
getClass	Retorna o nome da classe que lançou a exceção.
getStackTrace	Retorna uma matriz contendo a pilha de chamada de métodos, sendo útil para rastrear o caminho percorrido até a exceção.
printStackTrace	Imprime a pilha de chamada de métodos em formato de mensagem.

Try e Catch

- As duas principais palavras-chave do Java para tratar exceções são o **try** (tentar) e o **catch** (capturar). O código a ser tratado deve ser colocado dentro de blocos iniciados com estas palavras.

- Ao entrar em um bloco **try**, o programa tenta executar todas as instruções presentes nele.

- Se ocorrer erro em alguma instrução, a execução sai do bloco **try** e prossegue no bloco subsequente (**catch**), onde devem estar as rotinas de tratamento de erros.

Try e Catch

- Após um bloco **try**, podem existir vários blocos **catch**, um para cada tipo de exceção.

- Assim, quando o bloco **try** lança uma exceção, a execução é encaminhada para o bloco **catch** específico, que trata o tipo de exceção lançado.

- A palavra-chave **catch** deve vir acompanhada, entre parênteses, do tipo de exceção, que na verdade é uma classe, e de uma instância desta classe que permitirá acessar os métodos desta classe.

  Sintaxe: catch (<classe_exceção> <instância>)

l.bertholdo@ifsp.edu.br

Try e Catch

Note que, usando **try/catch** para chamar um método, pode-se tratar exceções causadas dentro dos métodos chamados, caso estes não tenham bloco **try/catch**.

```
Metodo1(){
    código...

    try
        Metodo2();
    catch (NumberFormatException e)
        System.out.print(e.getMessage());

    código...
}
```

2) Metodo2 não tem **try/catch**, então procura **try/catch** no método chamador.

```
Metodo2(){
    código...

    Metodo3();

    código...
}
```

3) Encontra o **try/catch**, trata a exceção e continua a execução após o **catch**.

1) **Exceção!** (Metodo3 não tem **try/catch**, então procura **try/catch** no método chamador).

A execução prossegue no **Metodo1** (que contém o bloco **catch** que tratou a exceção) e não no **Metodo3** que causou a exceção.

```
Metodo3(){
    código...

    int num = Integer.parseInt("12a4");

    código...
}
```

l.bertholdo@ifsp.edu.br

Try e Catch

- Se no exemplo anterior o Metodo1 não tivesse o bloco **try/catch**, o programa seria encerrado com uma exceção não tratada.

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "12a4"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at aula7.Exemplo1.main(Exemplo1.java:6)
```

Try e Catch

```java
public class Exemplo1 {
    public static void main(String[] args) {
        try { // Lança a exceção.
            int numero = Integer.parseInt("12a4");
            System.out.println("Entrada convertida para o tipo inteiro!");
        } catch (NumberFormatException e) { // Captura a exceção.
            // Trata a exceção.
            System.out.println("A operação retornou o seguinte erro:");
            System.out.println(e.getMessage());
        }

        System.out.println("Fim do programa!"); // A execução continua após o bloco catch.
    }
}
```

```
A operação retornou o seguinte erro:
For input string: "12a4"
Fim do programa!
```

l.bertholdo@ifsp.edu.br

Try e Catch

- Um determinado bloco de código pode estar sujeito a muitos tipos de exceção, sendo necessários vários blocos **catch**.
- Para não ter que usar um bloco **catch** para cada tipo de exceção, é possível usar a classe **Exception** que herda para todas as outras classes de exceção (*Checked Exceptions* e *Unchecked Exceptions*).
- Ao usar a classe **Exception**, o primeiro tipo de exceção que for encontrado é capturado, sendo que os demais não são verificados.

```
try { // Lança a exceção.
    int numero = Integer.parseInt("12a4");
    System.out.println("Entrada convertida para o tipo inteiro!");
} catch (Exception e) { // Captura a exceção.
    // Trata a exceção.
    System.out.println("A operação retornou o seguinte erro:");
    if (e.getClass().toString().equals("class java.lang.NumberFormatException"))
        System.out.println("O número informado é inválido!");
}
```

Ao usar a classe **Exception**, torna-se necessário recuperar o tipo de exceção capturado para realizar tratamentos customizados para cada tipo de exceção, o que pode demandar uma sequência de IFs.

```
A operação retornou o seguinte erro:
O número informado é inválido!
Fim do programa!
```

l.bertholdo@ifsp.edu.br

Multi-Catch

- Uma alternativa ao uso de vários blocos **catch**, é usar o recurso **multi-catch**, que permite a um bloco **catch** capturar mais de um tipo de exceção ao mesmo tempo.

- Para isso, basta separar as classes de exceção pelo caractere “|”.

- Assim como ocorre no uso classe genérica **Exception**, ao usar **multi-catch**, torna-se necessário recuperar o tipo de exceção capturado para realizar tratamentos customizados para cada tipo de exceção.

Multi-Catch

```java
public class Exemplo4 {
    public static void main(String[] args) {
        Scanner dado = new Scanner(System.in);

        try {
            System.out.print("Informe o índice: ");
            int i = dado.nextInt();
            System.out.println(GetNumero(i));
        } catch (IndexOutOfBoundsException | InputMismatchException e) {
            if (e.getClass().toString().equals("class java.lang.ArrayIndexOutOfBoundsException"))
                System.out.println("O índice informado não existe!");
            else if (e.getClass().toString().equals("class java.util.InputMismatchException"))
                System.out.println("O índice informado não é um número!");
        }
        dado.close();
    }

    public static int GetNumero(int indice) {
        int[] numeros = { 10, 20, 30 };
        return numeros[indice];
    }
}
```

```
Informe o índice: 3
O índice informado não existe!
```

```
Informe o índice: x
O índice informado não é um número!
```

l.bertholdo@ifsp.edu.br

Finally

- Quando uma exceção é lançada, ela altera o fluxo de execução do programa.
- Por exemplo: Quando um bloco **catch** trata uma exceção, o fluxo de execução continua imediatamente após este bloco, e não após o código que lançou a exceção.
- Porém, em alguns casos, pode-se desejar que o fluxo de execução continue logo após o código que causou a exceção.
- Para garantir que uma instrução sempre seja executada, mesmo que uma exceção tenha sido lançada, deve-se usar o bloco **finally**.
- Este bloco deve ser colocado logo após um bloco **try**, ou logo após o último bloco **catch** de um bloco **try/catch**.

l.bertholdo@ifsp.edu.br

Finally

```
Metodo1(){
    código...

    try
        Metodo2();
    catch (NumberFormatException e)
        System.out.print(e.getMessage());

    código...
}
```

2) Metodo2 não tem **try/catch**, então procura **try/catch** no método chamador.

```
Metodo2(){
    código...

    Metodo3();

    código...
}
```

3) Encontra o **try/catch**, trata a exceção e continua a execução após o **catch**.

```
Metodo3(){
    código...
    try
        int num = Integer.parseInt("12a4");
    finally
        código...
}
```

1) **Exceção!** Metodo3 não tem **catch** para tratar a exceção (ou não tem um **catch** para tratar este tipo de exceção). Nesse caso, o programa primeiro executa o código do bloco **finally** e, em seguida, procura um **try/catch** no método chamador.

l.bertholdo@ifsp.edu.br

Finally

Arquivo existente

```
public static void main(String[] args) {
    DataInputStream arq = null;
    try { // Lança a exceção.
        // Abre o arquivo texto.
        arq = new DataInputStream(new FileInputStream("C:\\teste\\dados.txt"));
        System.out.println("Arquivo aberto!");
    } catch (FileNotFoundException e) { // Captura a exceção.
        // Trata a exceção.
        System.out.println("A operação retornou o seguinte erro: ");
        System.out.println(e.getMessage());
    } finally { // Executa o código independentemente se houve ou não exceção.
        if (arq != null) {
            try {
                arq.close(); // Fecha o arquivo texto.
                System.out.println("Arquivo fechado!");
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else
            System.out.println("Erro na abertura do arquivo!");
    }
}
```

```
Arquivo aberto!
Arquivo fechado!
```

l.bertholdo@ifsp.edu.br

Finally

Arquivo inexistente

```
public static void main(String[] args) {
    DataInputStream arq = null;
    try { // Lança a exceção.
        // Abre o arquivo texto.
        arq = new DataInputStream(new FileInputStream("C:\\teste\\dados2.txt"));
        System.out.println("Arquivo aberto!");
    } catch (FileNotFoundException e) { // Captura a exceção.
        // Trata a exceção.
        System.out.println("A operação retornou o seguinte erro: ");
        System.out.println(e.getMessage());
    } finally { // Executa o código independentemente se houve ou não exceção.
        if (arq != null) {
            try {
                arq.close(); // Fecha o arquivo texto.
                System.out.println("Arquivo fechado!");
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else
            System.out.println("Erro na abertura do arquivo!");
    }
}
```

```
A operação retornou o seguinte erro:
C:\teste\dados2.txt (O sistema não pode encontrar o arquivo especificado)
Erro na abertura do arquivo!
```

Throw

- Instrução usada para lançar uma exceção de **forma explícita**.
- A instrução **throw** precisa instanciar a classe da exceção a ser lançada através da palavra **new**.
- Normalmente, essa instrução é usada em conjunto com um bloco **try/catch** para tratar a exceção lançada.

Throw

```
public class Exemplo2 {
    public static void main(String[] args) {
        try {
            System.out.println(GetNumero(3));
        } catch (IndexOutOfBoundsException e) {
            System.out.println(e.getMessage());
        }
    }

    public static int GetNumero(int indice) {
        int[] numeros = { 10, 20, 30 };

        if (indice > (numeros.length - 1))
            throw new IndexOutOfBoundsException("Este índice não existe!");

        return numeros[indice];
    }
}
```

Nesse exemplo, a mensagem padrão da classe de exceção é substituída pela mensagem lançada pela instrução **throw**. Assim, esta mensagem padrão seria impressa apenas para outros métodos chamados dentro do bloco **try** e que não usassem **throw** para lançar uma mensagem customizada de exceção.

```
Este índice não existe!
```

l.bertholdo@ifsp.edu.br

Throw

```java
public class Exemplo2 {
    public static void main(String[] args) {
        try {
            System.out.println(GetDiaSemana(8));
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
    }

    static String GetDiaSemana(int dia) {
        switch (dia) {
        case 1:
            return "Domingo";
        case 2:
            return "Segunda-feira";
        case 3:
            return "Terça-feira";
        case 4:
            return "Quarta-feira";
        case 5:
            return "Quinta-feira";
        case 6:
            return "Sexta-feira";
        case 7:
            return "Sábado";
        default:
            throw new IllegalArgumentException("Dia inexistente!");
        }
    }
}
```

```
Dia inexistente!
```

l.bertholdo@ifsp.edu.br

Throws

- Instrução usada em métodos que podem causar exceções monitoradas (*checked exceptions*).
- Este métodos devem capturar a exceção com **try/catch** ou ter em sua declaração a cláusula **throws**.

Throws

```java
public static void main(String[] args) {
    DataInputStream arq = null;
    try { // Lança a exceção.
        // Abre o arquivo texto.
        arq = new DataInputStream(new FileInputStream("C:\\teste\\dados.txt"));
        System.out.println("Arquivo aberto!");
    } catch (FileNotFoundException e) { // Captura a exceção.
        // Trata a exceção.
        System.out.println("A operação retornou o seguinte erro: ");
        System.out.println(e.getMessage());
    } finally { // Executa o código independentemente se houve ou não exceção.
        if (arq != null) {
            try {
                arq.close(); // Fecha o arquivo texto.
                System.out.println("Arquivo fechado!");
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else
            System.out.println("Erro na abertura do arquivo!");
    }
}
```

O método **main** usa o método **close()** da classe **FilterInputStream** (mãe da classe **DataInputStream**), o qual pode causar uma exceção monitorada do tipo **IOException**.

```java
public static void main(String[] args) throws IOException {
    DataInputStream arq = null;
    try { // Lança a exceção.
        // Abre o arquivo texto.
        arq = new DataInputStream(new FileInputStream("C:\\teste\\dados.txt"));
        System.out.println("Arquivo aberto!");
    } catch (FileNotFoundException e) { // Captura a exceção.
        // Trata a exceção.
        System.out.println("A operação retornou o seguinte erro: ");
        System.out.println(e.getMessage());
    } finally { // Executa o código independentemente se houve ou não exceção.
        if (arq != null) {
            arq.close(); // Fecha o arquivo texto.
            System.out.println("Arquivo fechado!");
        } else
            System.out.println("Erro na abertura do arquivo!");
    }
}
```

Referências

- Peter Jandl Junior; Java – Guia do Programador – 3ª Edição. São Paulo: Novatec Editora, 2015.
- https://docs.oracle.com/javase/8/docs/api/overview-tree.html