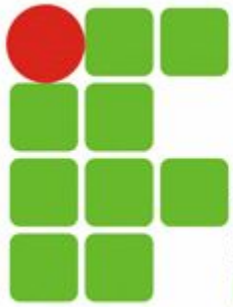


INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
FLUMINENSE

PROGRAMAÇÃO ORIENTADA A OBJETOS

Tratamento de Exceções

Prof. Roberta B Tôrres

Terminologia

- Exceção é a ocorrência de uma condição anormal durante a execução de um método;
- Falha é a inabilidade de um método cumprir a sua função;
- Erro é a presença de um método que não satisfaz sua especificação.

Em geral a existência de um erro gera uma falha que resulta em uma exceção !

https://slideplayer.com.br/slide/358200/

Três tipos de erros de tempo de execução

- 1. Erros de lógica de programação
  - Ex: limites do vetor ultrapassados, divisão por zero
  - Devem ser corrigidos pelo programador
- 2. Erros devido a condições do ambiente de execução
  - Ex: arquivo não encontrado, rede fora do ar, etc.
  - Fogem do controle do programador mas podem ser contornados em tempo de execução
- 3. Erros graves onde não adianta tentar recuperação
  - Ex: falta de memória, erro interno do JVM
  - Fogem do controle do programador e não podem ser contornados

Situação 1

Suponha uma classe **Conta e ContaCorrente**. Um depósito de 500,00 e um saque de 1000,00

```
public abstract class Conta {
        private float saldo;

        public void deposita(float valor)
        {   this.saldo = this.saldo + valor; }

        public void saque(float valor)
        {   this.saldo = this.saldo - valor; }

        public float getSaldo()
        {   return this.saldo; }

}
```

```
public class ContaCorrente extends Conta {

}
```

```
ContaCorrente cc = new ContaCorrente();
cc.deposita(500);

System.out.println("* Saldo da Conta = "+ cc.getSaldo());
cc.saque(1000);
System.out.println("* Saldo da Conta = "+ cc.getSaldo());
```

Temos aqui um **ERRO de Lógica**, considerando que não há tratamento para o valor a ser sacado.

Motivações para as Exceções

- *Exceções são*
  - *Erros de tempo de execução*
  - *Objetos criados a partir de classes especiais que são "lançados" quando ocorrem condições excepcionais*

Um método pode detectar uma falha mas não estar apto a resolver sua causa, devendo repassar essa função a quem saiba. As causas podem ser basicamente de três tipos:

- Erros de lógica de programação;
- Erros devido a condições do ambiente de execução (arquivo não encontrado, rede fora do ar, etc.);
- Erros irrecuperáveis (erro interno na JVM, etc);

Suporte as Exceções

As linguagens OO tipicamente dão suporte ao uso de exceções. Para usarmos exceções precisamos de:

- uma representação para a exceção;
- uma forma de lançar a exceção;
- uma forma de tratar a exceção.

Java suporta o uso de exceções:

- são representadas por classes;
- são lançadas pelo comando throw;
- são tratadas pela estrutura try-catch-finally.

Situação 2

Em Java, os métodos seguem um **contrato**. Caso, ao longo da execução, algum erro ocorra, o método envolvido na situação, deve avisar aos métodos executados antes dele.

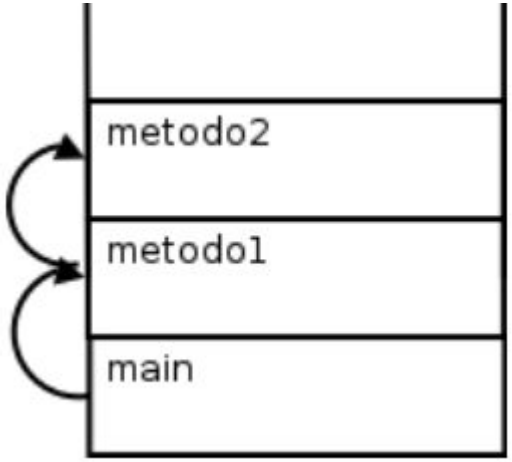
```java
// METODO MAIN
    public static void main(String[] args) {
        System.out.println("* INÍCIO DO MAIN");
         metodo1();
        System.out.println("* Fim do Main");
    }

// METODO 01
    static void metodo1() {
        System.out.println("-- Inicio do Metodo1");
        metodo2();
        System.out.println("-- Fim do Metodo1");
    }

// METODO 02
    static void metodo2() {
        System.out.println("## Inicio do Metodo2");
        ContaCorrente cc = new ContaCorrente();
        for (int i = 0; i <= 15; i++) {
            cc.deposita(1000);
            System.out.println(" * Saldo da Conta Corrente (i = "+ i + ") = " + cc.getSaldo());
            if (i == 3) {
                cc = null; // erro intencional
            }
        }
        System.out.println("## Fim do Metodo2");
    }
```

```
* INÍCIO DO MAIN
-- Inicio do Metodo1
## Inicio do Metodo2
 * Saldo da Conta Corrente (i = 0) = 1000.0
 * Saldo da Conta Corrente (i = 1) = 2000.0
 * Saldo da Conta Corrente (i = 2) = 3000.0
 * Saldo da Conta Corrente (i = 3) = 4000.0
Exception in thread "main" java.lang.NullPointerException
        at TesteExcecao.metodo2(TesteExcecao.java:40)
        at TesteExcecao.metodo1(TesteExcecao.java:31)
        at TesteExcecao.main(TesteExcecao.java:13)
```

Situação 2

Repare:

- Método **main** chamando **metodo1** e esse, por sua vez, chamando o **metodo2**. Cada um desses métodos pode ter suas próprias variáveis locais, isto é, o **metodo1** não enxerga as variáveis declaradas dentro do **main** e assim por diante.
- Em Java (e muitas outras linguagens), toda invocação de método é empilhada em uma estrutura de dados que isola a área de memória de cada um.
- Quando um método termina (retorna), ele volta para o método que o invocou. Ele descobre isso através da **pilha de execução** (***stack***).
- Propositalmente, o **metodo2** possui um problema, acessando uma referência nula quando o índice for igual a 4, gerando uma exceção!



Tratamento de Exceções

O sistema de exceções do Java funciona da seguinte maneira:

- quando a 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. Na situação 2, o **metodo2** não toma nenhuma medida diferente.
- Como o metodo2 não trata esse problema, a JVM pára 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 ***NullPointerException***?" "**Não**..."
  - Volta para o **main**, onde também não há proteção.
  - Então a JVM morre (na verdade, quem morre é apenas a **thread** corrente).

Tratamento de Exceções

Para entender o controle de fluxo de uma **Exception** em Java, veja o **metodo2** com o tratamento adequado (try...catch).

```
// METODO 02 com Tratamento de Exceção
static void metodo2() {
    System.out.println("## Inicio do Metodo2");
    ContaCorrente cc = new ContaCorrente();
    try {
        for (int i = 0; i <= 15; i++) {
            cc.deposita(i + 1000);
            System.out.println(" * Saldo da Conta Corrente = "+ cc.getSaldo());
            if (i == 3)
            { cc = null;   } // erro intencional
        }
     } catch (NullPointerException e) {
        System.out.println(" ---> Erro tratado: " + e);
     }
    System.out.println("## Fim do Metodo2");
}
```

Veja que o código que vai tentar (try) executar o bloco perigoso e, caso o problema seja do tipo **NullPointerException**, ele será pego (caught).

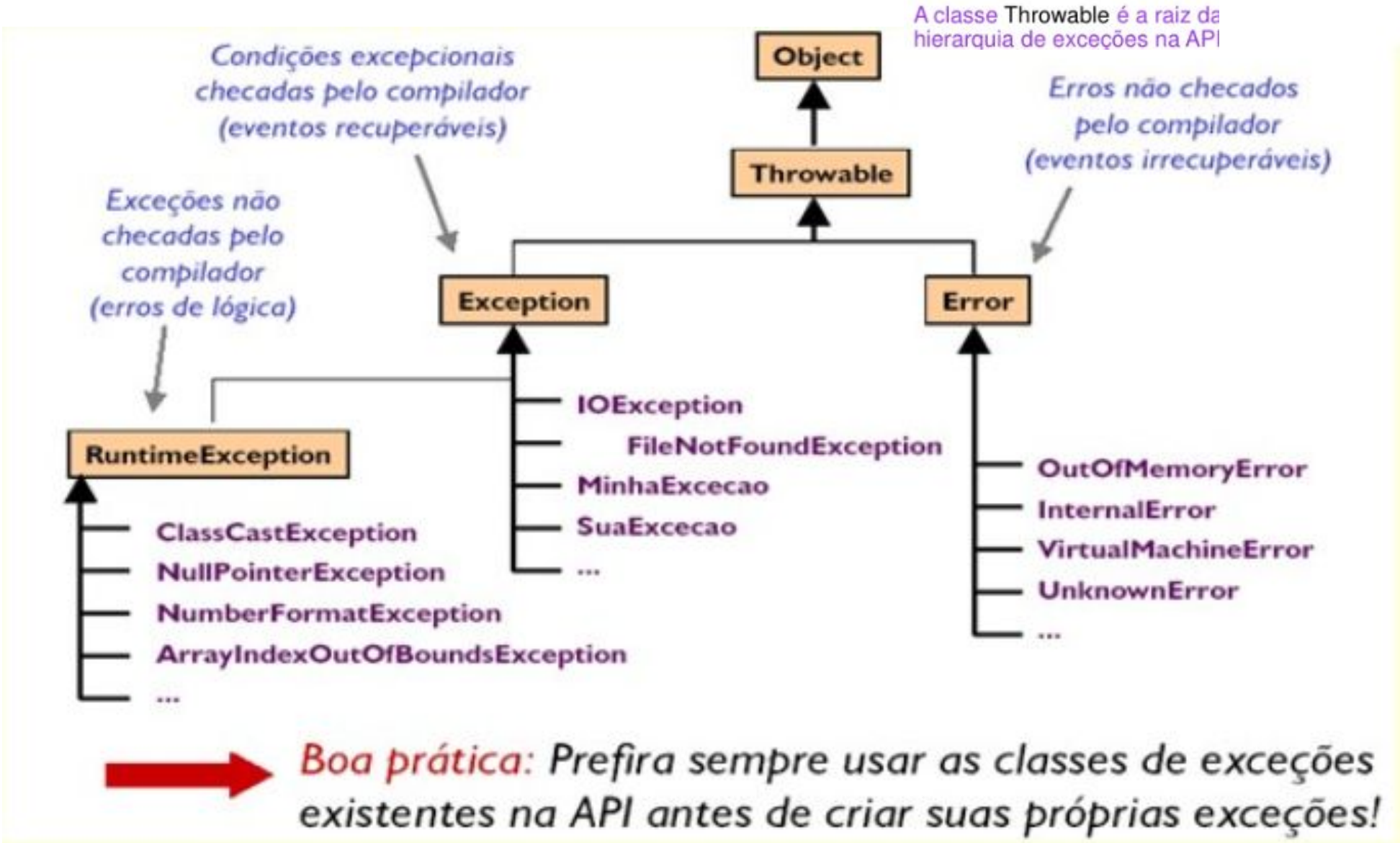
```
<terminated> TesteExcecao [Java Application] C:\Program Files\Java\j
-- Inicio do Metodo1
## Inicio do Metodo2
 * Saldo da Conta Corrente = 1000.0
 * Saldo da Conta Corrente = 2001.0
 * Saldo da Conta Corrente = 3003.0
 * Saldo da Conta Corrente = 4006.0
 ---> Erro tratado: java.lang.NullPointerException
## Fim do Metodo2
-- Fim do Metodo1
* Fim do Main
```

Tratamento de Exceções

Qual o efeito da exceção?

- *Uma exceção lançada interrompe o fluxo normal do programa*
  - *O fluxo do programa segue a exceção*
  - *Se o método onde ela ocorrer não a capturar, ela será propagada para o método que chamar esse método e assim por diante*
  - *Se ninguém capturar a exceção, ela irá causar o término da aplicação*
  - *Se em algum lugar ela for capturada, o controle pode ser recuperado*

Hierarquia de Exceções



Exceções

- *RuntimeException* e *Error*
  - *Exceções não verificadas em tempo de compilação*
  - *Subclasses de Error não devem ser capturadas (são situações graves em que a recuperação é impossível ou indesejável)*
  - *Subclasses de RuntimeException representam erros de lógica de programação que devem ser corrigidos (podem, mas não devem ser capturadas: erros devem ser corrigidos)*
- *Exception*
  - *Exceções verificadas em tempo de compilação (exceção à regra são as subclasses de RuntimeException)*
  - *Compilador exige que sejam ou capturadas ou declaradas pelo método que potencialmente as provoca*

Exceções

- A hierarquia Error descreve erros internos e problemas ocasionados pelo término de recursos dentro da máquina virtual Java.
- Não se deve lançar um objeto desse tipo. Apenas a JVM deve fazer uso das classes derivadas de Error.
- Pouco se pode fazer quando um erro desse tipo acontece. Essas situações, em geral, são bastante raras, portanto, não é exigido que um programa trate tais exceções.

Exceções

- Ao desenvolvermos aplicativos em Java, devemos utilizar as classes da hierarquia Exception.

- Esta também subdivide-se em duas, que são as exceções que derivam de RuntimeException e as demais subclasses que estendem diretamente a classe Exception (package java.lang).

Exceções

- Uma RuntimeException ocorre normalmente devido a erros de programação.
- Sendo assim, exceções desse tipo indicam condições que nunca devem acontecer se o programa estiver implementado adequadamente.
- Por essa razão, o compilador não exige que esse tipo de exceção seja tratada, logo, são chamadas de "*unchecked exceptions*".

Exceções - Boas Práticas

- Para evitar ClassCastException, use sempre **instanceof**:

```
public static Aluno converteParaAluno(Object obj) {
    Aluno a = null;
    if (obj instanceof Aluno){
        a = (Aluno)obj;
    }
    return a;
}
```

Exceções - Boas Práticas

- Para evitar NullPointerException, verifique sempre se a variável é diferente de null, antes de utilizá-la para acessar membros da classe:

```
public static void main (String[] args){
    Object o = new Object();
    Aluno a = converteParaAluno(o);
    if( a != null ){
        String nome = a.getNome();
        System.out.println(nome);
    } else {
        System.out.println("O argumento não é do tipo Aluno");
    }
}
```

Exceções - Boas Práticas

- Para evitar ArrayIndexOutOfBoundsException, use sempre **length**:

```
for (int i = 0; i < args.length; i++){
    String arg = args[i];
    System.out.println(arg + "\n");
}
```

Checked Exceptions

- As demais exceções, que são subclasses diretas de Exception, estão relacionadas a problemas que podem ocorrer com facilidade em aplicativos implementados corretamente, como por exemplo erros gerados por entrada incorreta de dados pelos usuários ou por problemas de leitura e escrita (I/O).

- O compilador exige que tais exceções sejam capturadas e tratadas, por isso são chamadas de "*checked exceptions*".

Lançando a Exceção

- Embora toda *checked exception* deva ser tratada, não é obrigatório tratá-la no mesmo escopo do método no qual ela foi gerada.
- Sendo assim, é possível propagar a exceção para um nível acima na pilha de invocações.
- Para tanto, o método que está deixando de capturar e tratar a exceção faz uso da palavra-chave throws na sua declaração.

Lançando a Exceções

- Existem vários métodos da API do Java que são declarados de forma a lançar exceções.
- Por exemplo, o construtor da classe FileReader é declarado como segue:

```
public FileReader(String filename)
                        throws FileNotFoundException
```

- Se um objeto da classe for criado usando um nome de arquivo inexistente, uma exceção do tipo FileNotFoundException é lançada.

Lançando a Exceções

```
// Podemos criar nossos próprios métodos e lançar as possíveis exceções geradas:

public static void copyFile(String sourceFile) throws FileNotFoundException,
                                                       IOException {
    File inputFile  = new File(sourceFile);
    File outputFile = new File("Output.txt");

    FileReader in  = new FileReader(inputFile);
    FileWriter out  = new FileWriter(outputFile);
    int c;

    while ((c = in.read()) != -1){
        out.write(c);
    }
    in.close();
    out.close();

    System.out.println("Cópia realizada com sucesso.");
}
```

Lançando a Exceções

```
public class RelatorioFinanceiro {
    public void metodoMau() throws ExcecaoContabil {
        if (!dadosCorretos) {
            throw new ExcecaoContabil("Dados Incorretos");
        }
    }
    public void metodoBom() {
        try {
            ... instruções ...
            metodoMau();
            ... instruções ...
        } catch (ExcecaoContabil ex) {
            System.out.println("Erro: " + ex.getMessage());
        }
        ... instruções ...
    }
}
```

instruções que sempre serão executadas

instruções serão executadas se exceção não ocorrer

instruções serão executadas se exceção não ocorrer ou se ocorrer e for capturada

Lançando a Exceções

- Para gerar uma condição de erro durante a execução de um método e lançar uma exceção específica, um objeto dessa classe deve ser criado e, através do comando throw, propagado para os métodos anteriores na pilha de execução.

Exceções

- As duas atividades associadas à manipulação de uma exceção são:
  - **geração:** a sinalização de que uma condição excepcional (por exemplo, um erro) ocorreu, e
  - **captura:** a manipulação (tratamento) da situação excepcional, onde as ações necessárias para a recuperação da situação de erro são definidas.

Capturando a Exceção Lançada

```
import java.io.*;

public class Copy {

    public static void main(String[] args) {
        try {
            copyFile("Copy.txt");
        } catch (FileNotFoundException e) {
            System.out.println("Arquivo não encontrado.");
        } catch (IOException e) {
            System.out.println("Ocorreu um erro de entrada e saída.");
        }
    }

    public static void copyFile(String file) throws FileNotFoundException,
                                                    IOException { …
}
```

Exceções

- Para cada exceção que pode ocorrer durante a execução do código, um bloco de ações de tratamento (conhecido como *exception handler*) deve ser especificado.
- O compilador Java verifica e força que toda exceção "não-trivial" tenha um bloco de tratamento associado.
  - As exceções desse tipo são chamadas de "*checked exceptions*".

Exceções - Sintaxe

- A sintaxe para captura e tratamento de exceção é:

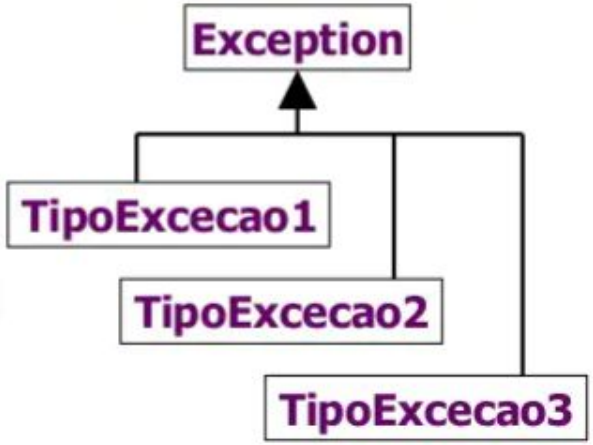
```
try {
    // bloco de código que inclui comandos que podem gerar uma exceção
} catch (XException ex) {
    // bloco de tratamento associado à condição de exceção
    // XException ou a qualquer uma de suas subclasses,
    // identificada aqui pelo objeto com referência ex
} catch (YException ey) {
    // bloco de tratamento para a situação de exceção
    // YException ou a qualquer uma de suas subclasses
} finally {
    // bloco de código que sempre será executado após
    // o bloco try, independentemente de sua conclusão
    // ter ocorrido normalmente ou ter sido interrompida
}
```

Toda vez que a estrutura **try** é utilizada, obrigatoriamente, em seu encerramento (na chave final), deve existir pelo menos um ***catch***.

Caso a estrutura ***try*** não tenha nenhum bloco ***catch***, o bloco ***finally*** torna-se obrigatório.

Exceções - Sintaxe

- *O bloco try "tenta" executar um bloco de código que pode causar exceção*
- *Deve ser seguido por*
  - *Um ou mais blocos catch(TipoDeExcecao ref)*
  - *E/ou um bloco finally*
- *Blocos catch recebem tipo de exceção como argumento*
  - *Se ocorrer uma exceção no try, ela irá descer pelos catch até encontrar um que declare capturar exceções de uma classe ou superclasse da exceção*
  - *Apenas um dos blocos catch é executado*




```
try {
    ... instruções ...
} catch (TipoExcecao1 ex) {
    ... faz alguma coisa ...
} catch (TipoExcecao2 ex) {
    ... faz alguma coisa ...
} catch (Exception ex) {
    ... faz alguma coisa ...
}

... continuação ...
```

Exceções - Exemplo

```
public static void main(String[] args) {
    try {
        FileWriter fileOut = new FileWriter("resultado.txt", true);
        for (int i = 0; i < args.length; i++){
            String arg = args[i];
            fileOut.write(arg + "\n");
        }
        fileOut.close();
    } catch (IOException e) {
        System.out.println("Ocorreu um erro de entrada e saída.");
    } catch (Exception e) {
        System.out.println("Ocorreu uma exceção genérica.");
    }
}
```

Exception: Métodos padrões

- *Construtores de Exception*
  - *Exception ()*
  - *Exception (String message)*
  - *Exception (String message, Throwable cause) [Java 1.4]*
- *Métodos de Exception*
  - *String getMessage()*
    - *Retorna mensagem passada pelo construtor*
  - *Throwable getCause()*
    - *Retorna exceção que causou esta exceção [Java 1.4]*
  - *String toString()*
    - *Retorna nome da exceção e mensagem*
  - *void printStackTrace()*
    - *Imprime detalhes (stack trace) sobre exceção*

Criando uma Classe de Exceção

- Exceções são classes. Assim, é possível que uma aplicação defina suas próprias exceções através do mecanismo de definição de classes.
- Para criar uma nova exceção deve-se estender a classe Exception ou uma de suas subclasses.
- Um classe de exceção pode conter tudo o que uma classe regular contém, isto é, atributos, construtores e métodos.

Criando uma Classe de Exceção NotaInvalidaException

- Veja o exemplo de uma nova classe de exceção que pode ser utilizada na classe Resultado.java:

```
public class NotaInvalidaException extends Exception {

    public NotaInvalidaException() {
        super();
    }

    public NotaInvalidaException(String message) {
        super(message);
    }
}
```

Criando uma Classe de Exceção
Classe Resultado

```
public class Resultado{
    private double nota;
    public double getNota() {
        return nota;
    }
    public void setNota(double nota) throws NotaInvalidaException {
        if(nota >= 0.0 && nota <= 10.0){
            this.nota = nota;
        } else {
            throw new NotaInvalidaException("Erro: A nota deve pertencer
                                             ao intervalo de 0 a 10.");
        }
    }
}
```

Repare que a classe Resultado apenas lança a exceção.

Criando uma Classe de Exceção Exemplo

```
public class TesteNotaInvalida {

    public static void main(String[] args) {

        resultado objResultado = new resultado();
        try {
            objResultado.setnota(5);
            System.out.println("Nota lançada: " + objResultado.getnota());
            objResultado.setnota(11);
            System.out.println("Nota lançada: " + objResultado.getnota());
        }
        catch (NotaInvalidaException e)
        {   System.out.println( e.getMessage() );}
    }
}
```

```
<terminated> TesteNotaInvalida [Java Application] C:\Program Files\Java
Nota lançada: 5.0
AVISO: Nota deve pertencer ao intervalo de 0 a 10.
```

A captura e tratamento da exceção é feita no método ***main()***.

Exceções - Bloco Finally

- O bloco finally, quando presente, é sempre executado.
- Em geral, ele inclui comandos que liberam recursos que eventualmente possam ter sido alocados durante o processamento do bloco try e que podem ser liberados, independentemente de a execução ter encerrado com sucesso ou ter sido interrompida por uma condição de exceção.
- A presença desse bloco é opcional.

ATENÇÃO: A estrutura *try* pode não conter nenhum bloco *catch*, porém o bloco *finally* torna-se obrigatório, nesta situação.

Exceções - Bloco Finally

```
private static Vector vector;
private static final int SIZE = 10;
public static void main(String[] args) {
    PrintWriter out = null;
    try {
        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Valor da posição: " + i + " = " + vector.elementAt(i));
        }
    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        } else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

Exceções Pré-definidas

Alguns exemplos de exceções já definidas no pacote `java.lang` incluem:

- `ArithmeticException`: indica situações de erros em processamento aritmético, tal como uma divisão inteira por 0. A divisão de um valor real por 0 não gera uma exceção (o resultado é o valor infinito);
- `NumberFormatException`: indica que tentou-se a conversão de uma string para um formato numérico, mas seu conteúdo não representava adequadamente um número para aquele formato. É uma subclasse de `IllegalArgumentException`;
- `IndexOutOfBounds`: indica a tentativa de acesso a um elemento de um agregado aquém ou além dos limites válidos. É a superclasse de `ArrayIndexOutOfBoundsException`, para arranjos, e de `StringIndexOutOfBounds`, para strings;
- `NullPointerException`: indica que a aplicação tentou usar uma referência a um objeto que não foi ainda definida;
- `ClassNotFoundException`: indica que a máquina virtual Java tentou carregar uma classe mas não foi possível encontrá-la durante a execução da aplicação.

Material Base

https://pt.slideshare.net/ludimila_monjardim/poo-23tratamento-deexcees

Introdução ao Java - Slides Prof. Ismael Santos
https://slideplayer.com.br/slide/358200/

Material Complementar

LEITURA RECOMENDADA

https://www.devmedia.com.br/tratando-excecoes-em-java/25514

https://www.caelum.com.br/apostila-java-orientacao-objetos/excecoes-e-controle-de-erros/

Material Consultado

- https://pt.slideshare.net/ludimila_monjardim/poo-23tratamento-deexcees
- https://slideplayer.com.br/slide/358200/
- https://pt.slideshare.net/denistuning/java-11-27818280
- https://pt.slideshare.net/regispires/java-13-excecoes-presentation

Outros exemplos na Apostila Caelum

https://www.caelum.com.br/apostila-java-orientacao-objetos/excecoes-e-controle-de-erros/#exerccio-para-comear-com-os-conceitos**)**