# Universidade Federal de Ouro Preto

## CSI032 - Programação de Computadores II

# Tratamento de Exceções

Professor: Dr. Rafael Frederico Alexandre
Contato: rfalexandre@decea.ufop.br
Colaboradores: Eduardo Matias Rodrigues
Contato: eduardo.matias@aluno.ufop.edu.br


# Tratamento de Exceções

# Tratamento de Exceções

# Exceção

## Acessando posição inválida de um Array

1. O que acontece se tentarmos acessar uma posição inválida de um Array?

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

    String[] paises = {"Brasil", "Peru", "Argentina"};

    System.out.println(paises[3]);
}
```

# Exceção

## ArrayIndexOutOfBoundsException

1. Teremos um problema, já que a posição 3 não existe:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at excecoes.Excecoes.main(Excecoes.java:21)
```

Figura: Saída do programa

1. O JAVA **lançou** um exceção;
2. A informação que aparece é chamada de **Stack Trace**;
3. Inclui o caminho que levou ao lançamento da exceção e qual exceção está sendo lançada;
   1. Exceção lançada: ArrayIndexOutOfBoundsException
   2. Lançada em: at excecoes.Excecoes.main(Excecoes.java:21)

# Introdução

## Exceção

1. Quando ocorre determinada situação que não deveria acontecer, o JAVA (e muitas outras linguagens) lançam uma exceção;
2. É uma forma de sinalizar e obrigar o programador a lidar com o código problemático.

# Exceção

## Definição

[Caelum, 2017]

Uma exceção representa uma situação que normalmente não ocorre e representa algo de estranho ou inesperado no sistema.

# Tratamento de Exceções

# Exceção

## Tratamento

1. No exemplo, o programa foi interrompido por um simples acesso a uma posição indevida do vetor;
   1. O JAVA (e outras linguagens) permite que tratemos uma exceção de forma a solucionar o mau funcionamento ou finalizar o programa de forma elegante para o usuário (ou programador).
   2. Para isso, vamos **tentar** (try {...}) executar o trecho de código que pode lançar a exceção e se ela for lançada, iremos **capturá-la** (catch (exceção lançada) {...})

# Exceção

## try, catch

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

    String[] paises = {"Brasil", "Peru", "Argentina"};

    try {
        System.out.println(paises[3]);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Posição inválida acessada\n");
    }

    System.out.println("Finalizando o programa...");
}
```

# Exceção

## try, catch

1. Tratando a exceção evitamos que o programa fosse encerrado pela JVM (Java Virtual Machine):

```
Posição inválida sendo acessada

Finalizando o programa...
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura: Saída do programa

# Exceção

## try, catch

1. "Sempre"que tivermos um trecho de código que pode lançar uma exceção, devemos colocá-lo dentro de um bloco **try** ;
2. Caso esse código lance uma exceção, a JVM irá verificar se há tratamento para aquele tipo de exceção e executará o código dentro do **catch()**;
    1. Caso não haja um tratamento para aquela exceção específica, o programa é interrompido.

# Exceção

## Divisão por Zero

1. Tomemos outro exemplo: um programa que lê dois números do teclado e executa a divisão.
2. O que acontece se o usuário informar o número zero como denominador?

# Exceção

## Divisão por Zero

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

    Scanner ler = new Scanner(System.in);

    System.out.print("Numerador: ");
    int numerador = ler.nextInt();
    System.out.print("Denominador: ");
    int denominador = ler.nextInt();

    int resultado = div(numerador, denominador);
    System.out.println(numerador + "/" + denominador + " = " + resultado)
}

public static int div(int numerador, int denominador) {
    return numerador/denominador;
}
```

# Exceção

## Divisão por Zero

1. A exceção **ArithmeticException: / by zero** é lançada;
2. Observe que a Stack Trace descreve a pilha de chamadas de métodos que levou ao lançamento da exceção. *Método main chamou o método div que lançou a exceção.*

```
Numerador: 10
Denominador: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at excecoes.Excecoes.div(Excecoes.java:33)
        at excecoes.Excecoes.main(Excecoes.java:28)
```

- Apenas para exemplo iremos incluir mais um método para realizar a divisão.

# Exceção

## Divisão por Zero

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

    Scanner ler = new Scanner(System.in);

    System.out.print("Numerador: ");
    int numerador = ler.nextInt();
    System.out.print("Denominador: ");
    int denominador = ler.nextInt();

    int resultado = div1(numerador, denominador);
    System.out.println(numerador + "/" + denominador + " = " + resultado);
}

public static int div1(int numerador, int denominador) {
    return div2(numerador, denominador);
}

public static int div2(int numerador, int denominador) {
    return numerador/denominador;
}
```

# Exceção

## Divisão por Zero

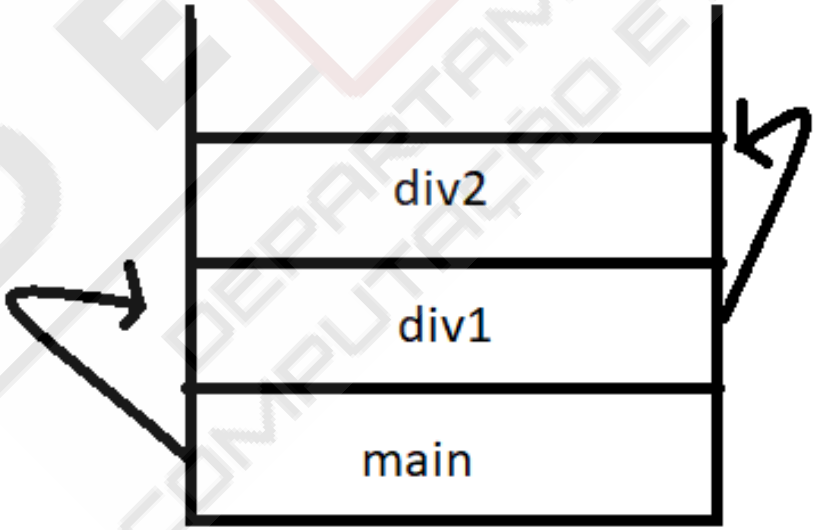
```
Numerador: 20
Denominador: 0
Exception in thread "main" java.lang.ArithmeticException: / by zer
        at excecoes.Excecoes.div2(Excecoes.java:37)
        at excecoes.Excecoes.div1(Excecoes.java:33)
        at excecoes.Excecoes.main(Excecoes.java:28)
```

1. Podemos ver o rastreio das chamadas dos métodos que levaram ao lançamento da exceção, *main* chama *div1* que chama o *div2* que lança uma exceção;
2. Como isso acontece?

# Exceção

## Divisão por Zero

1. Quando um método é chamado, ele é empilhado em uma estrutura de dados que isola a área de memória de cada método;
2. Quando o método termina sua execução, ele é desempilhado e o programa volta para o método que faz a chamada;



# Exceção

## Divisão por Zero

1. A Stack Trace de uma exceção é muito importante, ela é a primeira a ser analisada para identificar o problema;

# Exceção

## Stack Trace

1. Sistema de exceção:
   1. Quando o método *div2* tenta fazer a divisão por zero e a exceção ArithmeticException é lançada, a JVM verifica se essa exceção é tratada no método *div2*, como não é, a execução desse método é interropida, ele é desempilhado e o programa volta ao método *div1*;
   2. É verificado se o método *div1* possui tratamento para a exceção ArithmeticException, como não possui, sua execução também é interropida e a execução volta ao método *main*;
   3. É verificado se o método *main* trata a exceção, como ele também não tomou as precauções, o programa é interrompido.

# Exceção

## Tratando a exceção

1. Podemos tratar a ArithmeticException em quaisquer um dos 3 métodos: *main*, *div1*, *div2*;

# Exceção

## Tratando a exceção na main

```
public static void main(String[] args) {
    Scanner ler = new Scanner(System.in);

    System.out.print("Numerador: ");
    int numerador = ler.nextInt();
    System.out.print("Denominador: ");
    int denominador = ler.nextInt();

    try {
        int resultado = div1(numerador, denominador);
        System.out.println(numerador + "/" + denominador + " = " + resultado);
    } catch (ArithmeticException e) {
        System.out.println("Exceção capturada na main");
    }
}

public static int div1(int numerador, int denominador) {
    return div2(numerador, denominador);
}

public static int div2(int numerador, int denominador) {
    return numerador / denominador;
```

# Exceção

## Tratando a exceção na main

```
run:
Numerador: 10
Denominador: 0
Exceção capturada na main
BUILD SUCCESSFUL (total time: 3 seconds)
```

# Exceção

## Tratando a exceção no método div2

```
public static void main(String[] args) {
    Scanner ler = new Scanner(System.in);

    System.out.print("Numerador: ");
    int numerador = ler.nextInt();
    System.out.print("Denominador: ");
    int denominador = ler.nextInt();

    int resultado = div1(numerador, denominador);
    System.out.println(numerador + "/" + denominador + " = " + resultado);
}

public static int div1(int numerador, int denominador) {
    return div2(numerador, denominador);
}

public static int div2(int numerador, int denominador) {
    try {
        return numerador / denominador;
    } catch (ArithmeticException e) {
        System.out.println("Exceção capturada no div2");
        return 0;
    }
}
```

# Exceção

## Tratando a exceção no método div2

```
run:
Numerador: 10
Denominador: 0
Exceção capturada no div2
10/0 = 0
BUILD SUCCESSFUL (total time: 3 seconds)
```

# Exceção

## Tratando a exceção

1. Implemente o programa e realize o tratamento da exceção no método *div1*;
2. Quando a exceção foi tratada no método *div2* tivemos que adicionar um **return 0** ao **catch**, o que alterou a saída do programa;
    1. Não há uma regra definitiva para o local onde a exceção deve ser tratada, o programador deve avaliar cada caso!

# Exceção

## InputMismatchException

1. E se o usuário errar na digitação e informar um caracter que não seja um número?

# Exceção

## InputMismatchException

```
public static void main(String[] args) {
    Scanner ler = new Scanner(System.in);

    System.out.print("Numerador: ");
    int numerador = ler.nextInt();

    System.out.print("Denominador: ");
    int denominador = ler.nextInt();

    int resultado = div(numerador, denominador);
    System.out.println("Resultado: " + resultado);
}

public static int div(int numerador, int denominador) {
    return numerador / denominador;
}
```

# Exceção

## InputMismatchException

- Uma exceção do tipo **InputMismatchException** é lançada;

```
Numerador: 10
Denominador: k
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at excecoes.Excecoes.main(Excecoes.java:26)
```

# Exceção

## Tratando várias exceções

1. Além do usuário errar e solicitar uma divisão por zero, ele também pode digitar um caracter diferente de um número inteiro e como o método *nextInt* da classe **Scanner** espera apenas um inteiro, temos um problema;
2. Felizmente, podemos tratar tantas exceções quanto forem necessárias,

# Exceção

## InputMismatchException e ArithmeticException

```
public static void main(String[] args) {
    Scanner ler = new Scanner(System.in);

    try {
        System.out.print("Numerador: ");
        int numerador = ler.nextInt();

        System.out.print("Denominador: ");
        int denominador = ler.nextInt();

        int resultado = div(numerador, denominador);
        System.out.println("Resultado: " + resultado);

    } catch (InputMismatchException e) {
        System.out.println("Entrada inválida");
    } catch (ArithmeticException e) {
        System.out.println("Divisão por zero");
    }
```

# Observação

## try

1. Observe que todo código que pode lançar um exceção está contido dentro do bloco **try {...}**;
2. Os métodos *nextInt* podem lançar a exceção InputMismatchException caso o usuário não digite um número inteiro;
3. O método *div* pode lançar a exceção ArithmeticException caso o usuário tente fazer divisão por 0;

# Exceção

## InputMismatchException e ArithmeticException

```
run:
Numerador: sadfsd
Entrada inválida
BUILD SUCCESSFUL (total time: 1 second)
```

# Exceção

## InputMismatchException e ArithmeticException

```
run:
Numerador: 9
Denominador: 0
Divisão por zero
BUILD SUCCESSFUL (total time: 4 seconds)
```

# Exceção

## **catch**: Forma resumida

```
public static void main(String[] args) {
    Scanner ler = new Scanner(System.in);

    try {
        System.out.print("Numerador: ");
        int numerador = ler.nextInt();

        System.out.print("Denominador: ");
        int denominador = ler.nextInt();

        int resultado = div(numerador, denominador);
        System.out.println("Resultado: " + resultado);

    } catch (InputMismatchException | ArithmeticException e) {
        System.out.println("Operação inválida");
    }
}
```

# Exceção

## catch

1. InputMismatchException, ArithmeticException e qualquer exceção que o java pode lançar são classes;
   1. o **catch** recebe uma instância da classe da exceção lançada;
   2. com esse objeto podemos obter algumas informações úteis;

# Exceção

## getMessage, printStackTrace, getStackTrace

1. **getMessage**: retorna uma string descritiva armazenada na exceção;
2. **printStackTrace**: exibe toda a StackTrace;
3. **getStackTrace**: retorna um array com elementos que contem as informações da StackTrace.

# Exceção

## getMessage

```
public static void main(String[] args) {
    Scanner ler = new Scanner(System.in);

    try {
        System.out.print("Numerador: ");
        int numerador = ler.nextInt();

        System.out.print("Denominador: ");
        int denominador = ler.nextInt();

        int resultado = div(numerador, denominador);
        System.out.println("Resultado: " + resultado);

    } catch (InputMismatchException | ArithmeticException e) {
        System.out.println(e.getMessage());
    }
}
```

# Exceção

## getMessage

```
run:
Numerador: 44
Denominador: 0
/ by zero
BUILD SUCCESSFUL (total time: 3 seconds)
```

# Exceção

## printStackTrace

```
public static void main(String[] args) {
    Scanner ler = new Scanner(System.in);

    try {
        System.out.print("Numerador: ");
        int numerador = ler.nextInt();

        System.out.print("Denominador: ");
        int denominador = ler.nextInt();

        int resultado = div(numerador, denominador);
        System.out.println("Resultado: " + resultado);

    } catch (InputMismatchException | ArithmeticException e) {
        e.printStackTrace();
    }
}
```

# Exceção

## printStackTrace

```
run:
Numerador: asdasd
java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at excecoes.Excecoes.main(Excecoes.java:25)
BUILD SUCCESSFUL (total time: 2 seconds)
```

# Exceção

## getStackTrace

1. O método *getStackTrace* retorna um vetor do tipo **StackTraceElement** onde cada elemento contem informações do programa naquele ponto da StackTrace;
2. Vejamos um exemplo para a exceção de divisão por zero;

# Exceção

## getStackTrace

```
public static void main(String[] args) {
    Scanner ler = new Scanner(System.in);

    try {
        System.out.print("Numerador: ");
        int numerador = ler.nextInt();

        System.out.print("Denominador: ");
        int denominador = ler.nextInt();

        int resultado = div(numerador, denominador);
        System.out.println("Resultado: " + resultado);

    } catch (InputMismatchException | ArithmeticException e) {
        StackTraceElement[] elementos = e.getStackTrace();

        for (StackTraceElement ele : elementos) {
            System.out.println("\nClasse: " + ele.getClassName());
            System.out.println("Arquivo: " + ele.getFileName());
            System.out.println("Método: " + ele.getMethodName());
            System.out.println("Linha: " + ele.getLineNumber());
        }
    }
}
```

# Exceção

## getStackTrace

```
run:
Numerador: 4
Denominador: 0

Classe: excecoes.Excecoes
Arquivo: Excecoes.java
Método: div
Linha: 46

Classe: excecoes.Excecoes
Arquivo: Excecoes.java
Método: main
Linha: 30
BUILD SUCCESSFUL (total time: 2 seconds)
```

# Exceção

## Recuperação

1. Tratar uma exceção permite que o programa se recupere de forma automática e continue executando;
2. Tomemos o exemplo da divisão entre os números, e se quisermos que o programa seja encerrado apenas quando a operação for realizada?

# Exceção

## Recuperação

```
public static void main(String[] args) {
    Scanner ler;

    boolean loop = true;

    while (loop) {
        ler = new Scanner(System.in);
        try {
            System.out.print("Numerador: ");
            int numerador = ler.nextInt();
            System.out.print("Denominador: ");
            int denominador = ler.nextInt();
            int resultado = div(numerador, denominador);
            System.out.println(numerador+"/"+denominador+"="+resultado);
            loop = false;
        } catch (InputMismatchException | ArithmeticException e) {
            System.out.println("Entrada inválida!\n");
        }
    }
}
```

# Exceção

## Recuperação: saída

```
run:
Numerador: 8
Denominador: 0
Entrada inválida!

Numerador: 7
Denominador: asdasdasv
Entrada inválida!

Numerador: ç
Entrada inválida!

Numerador: 8
Denominador: 4
8/4=2
BUILD SUCCESSFUL (total time: 12 seconds)
```

# Tratamento de Exceções

# Exceções

## Hierarquia

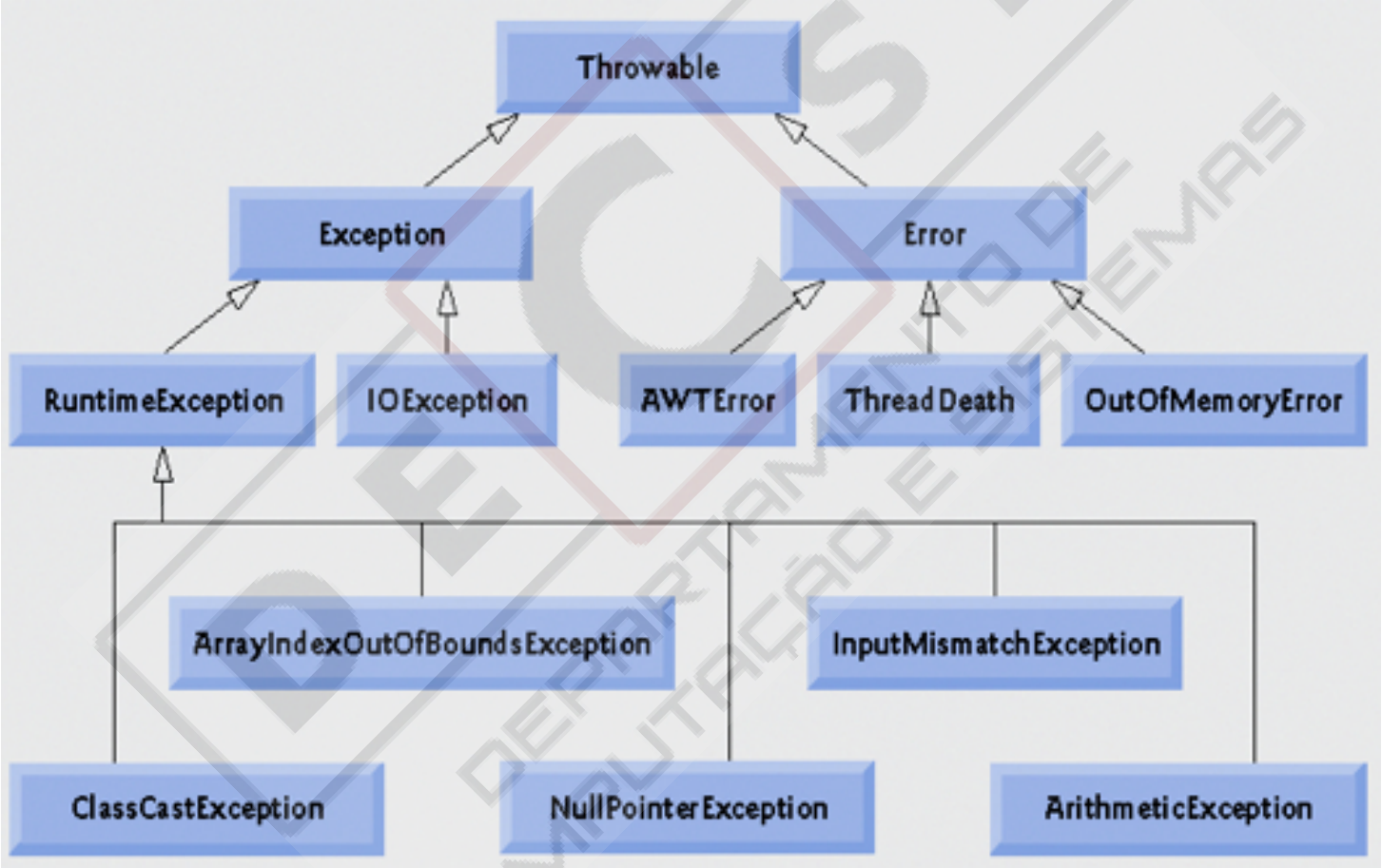



Figura: https://www.devmedia.com.br/trabalhando-com-excecoes-em-java/27601

# Exceções

## Hierarquia

1. Somente objetos **Throwable** podem ser utilizados com o mecanismo de tratamento de exceção;
2. **Throwable** tem duas filhas diretas:
   1. **Error** e suas subclasses representam situações anormais que acontecem na JVM, um programa não deve tentar capturar esses erros e normalmente não há como se recuperar deles;
   2. **Exception** podem ser capturadas pelo programa;
3. Os métodos getMessage, printStackTrace e getStackTrace são implementados na classe **Throwable**;

# Exceções não verificadas

## RunTimeException

1. Em nossos exemplos até agora, utilizamos as classe InputMismatchException e ArithmeticException que são filhas de **RunTimeException**;
2. Toda classe filha de **RunTimeException** lançam exceções não verificadas *(unchecked)*, ou seja, o compilador não exige que elas sejam capturadas ou **declaradas**;

# Exceções verificadas

## Abrindo arquivo

1. Um exemplo de exceção checada (checked: compilador nos obriga a "lidar"com ela, caso contrário um erro de compilação acontece) é a FileNotFoundException, lançada quando tentamos abrir um arquivo que não existe;
2. Se tentarmos executar um programa que abre um arquivo mas não trata ou declara a FileNotFoundException, um erro irá ocorrer.

```
public static void main(String[] args) {
    abreArquivo();

}

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

# Exceções verificadas

## catch-or-declare

1. Uma vez que o arquivo que queremos abrir pode não existir, somos obrigados a capturar a exceção no método *abreArquivo* ou declara-la (sinalizar que o método pode lançar uma exceção e quem chamá-lo, direta ou indiretamente ficará incubido de tratar essa exceção);
2. isso é conhecido como requisito "capture ou declare" (catch-or-declare);
3. Para declarar uma exceção basta colocar a palavra reservada **throws** antes do corpor do método, seguida das exceções que o método pode lançar.

# Declarando exceção

## throws

```
public static void main(String[] args) {
    try {
        abreArquivo();
    } catch (FileNotFoundException ex) {
        System.out.println("Arquivo inexistente");
    }

}

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

# Declarando exceção

## throws

1. Ao declarar a exceção FileNotFoundException no método abreArquivo(), estamos indicando a exceção que esse método pode lançar e a responsabilidade de tratá-la será de outro método que o chamou (mesmo de forma indireta);
2. também podemos capturar a exceção dentro do método abreArquivo(), em vez de ter feito na main;

# Capturando exceção

## try, catch

```
public static void main(String[] args) {
    abreArquivo();

}

public static void abreArquivo() {
    try {
        new java.io.FileInputStream("arquivo.txt");
    } catch (FileNotFoundException ex) {
        System.out.println("Arquivo inexistente");
    }
}
```

# Exceção

## catch-or-declare

1. Execute os código do exemplo anterior e observe a saída;
2. Podemos declarar uma exceção **unchecked** ainda que o compilador não nos obrigue a fazer isso;
   1. Pode melhorar a legibilidade e aparece no JAVADOC.

# Declarando exceção

## ArithmeticException

```
public static void main(String[] args) {
    Scanner ler = new Scanner(System.in);

    System.out.print("Numerador: ");
    int numerador = ler.nextInt();
    System.out.print("Denominador: ");
    int denominador = ler.nextInt();

    int resultado = div(numerador, denominador);
    System.out.println("Resultado: " + resultado);
}

public static int div(int numerador, int denominador)
        throws ArithmeticException {

    return numerador/denominador;
}
```

# Declarando exceção

## ArithmeticException

1. Apenas indicamos que o método pode lançar uma exceção, mas não tratamos ela caso seja lançada:

```
Numerador: 8
Denominador: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at excecoes.Excecoes.div(Excecoes.java:36)
        at excecoes.Excecoes.main(Excecoes.java:31)
```

# Tratamento de Exceções

# Exceção

## finnaly

1. Programas que alocam recursos devem liberá-los ainda que algo dê errado na execução:
   1. Um arquivo deve ser fechado mesmo que o programa não possa continuar;
   2. Uma conexão com o banco de dados sempre deve ser liberada após seu uso.
2. Se um recurso não for liberado, ele talvez não vai estar disponível para outro programa.
3. Para evitar isso temos o bloco **finnaly**;

# Exceção

## finnaly

1. Um bloco **finnaly** deve ser colocado após o último block catch, se não houver blocos catch, ele deve ser colocado após o try;
2. O bloco finnaly só não será executado se o programa for abortado utilizando o **System.exit**.

# Exceção

## finnaly

```
public static void main(String[] args) {
    Scanner ler = new Scanner(System.in);

    System.out.print("Numerador: ");
    int numerador = ler.nextInt();
    System.out.print("Denominador: ");
    int denominador = ler.nextInt();

    try {
        int resultado = div(numerador, denominador);
        System.out.println("Resultado: " + resultado);
    } catch (ArithmeticException e) {
        System.out.println("Exceção: " + e.getMessage());
    } finally {
        System.out.println("**Bloco finally**");
    }
}
```

# Exceção

## Execução normal

```
run:
Numerador: 10
Denominador: 5
Resultado: 2
**Bloco finally**
BUILD SUCCESSFUL (total time: 2 seconds)
```

# Exceção

## Exceção ArithmeticException lançada

```
run:
Numerador: 9
Denominador: 0
Exceção: / by zero
**Bloco finally**
BUILD SUCCESSFUL (total time: 4 seconds)
```

# Tratamento de Exceções

# Exceção

## Novas exceções

1. Na maioria das vezes utilizamos as classes existentes na API do JAVA, cujos métodos lançam as exceções apropriadas;
2. Se formos construir uma classe para outros programadores utilizarem é adequado declarar nossas próprias classes de exceção;
3. Uma nova classe de exceção deve herdar de uma classe de exceção existente para o mecanismo de tratamento de exceção funcionar;

# Exemplo

## [Caelum, 2017] Adap.

1. Suponha um sistema de contas bancárias. De uma conta devemos saber o nome do titular, o número da conta e o saldo. Deve ser possível realizar saques e depósitos de uma conta, caso o usuário tente sacar um quantia não disponível o sistema deve lançar uma exceção.

# Exemplo

## [Caelum, 2017] Adap.

```
public class SaldoInsuficienteException extends RuntimeException {

    public SaldoInsuficienteException(String mensagem) {
        super(mensagem);
    }
}
```

# Exemplo

[Caelum, 2017] Adap.

1. Fizemos a classe SaldoInsuficienteException herdar de RuntimeException, lembre-se: subclasses da RuntimeException lançam exceções unchecked, ou seja, o compilador não irá nos obrigar a tratá-la ou declará-la;
2. Se quisermos que ela seja uma exceção checked, basta herdar da classe Exception.

# Exemplo

## [Caelum, 2017] Adap.

```
public class Conta {

    private int id;
    private String titular;
    private double saldo;

    public Conta(int id, String titular) {
        this.id = id;
        this.titular = titular;
        this.saldo = 0;
    }
```

# Exemplo

## [Caelum, 2017] Adap.

```
public void sacar(double valor) {
    if (valor > saldo) {
        SaldoInsuficienteException e = new SaldoInsuficienteException("Saldo insuficiente, " +
                                                                      "tente um quantia menor");
        throw e; // lança a exceção
    }
    this.saldo -= valor;
}

public void depositar(double valor) {
    this.saldo += valor;
}
```

# Lançando uma exceção

## throw

1. Para lançar qualquer exceção utilizamos a palavra reservada **throw** seguida do objeto a ser lançado;
2. Não confunda com a palavra **throws**, utilizada para declarar que um método pode lançar um determinada exceção.

# Exemplo

[Caelum, 2017] Adap.

```
public static void main(String[] args) {
    Conta c1 = new Conta(1, "Eduardo");

    c1.depositar(100);
    c1.sacar(200);
}
```

1. Saída do programa: Exception in thread "main"excecoes.SaldoInsuficienteException: Saldo insuficiente, tente um quantia menor.

# Tratando a exceção

[Caelum, 2017] Adap.

```
public static void main(String[] args) {
    Conta c1 = new Conta(1, "Eduardo");

    c1.depositar(100);

    try {
        c1.sacar(200);
    } catch (SaldoInsuficienteException e) {
        System.out.println("Exceção: " + e.getMessage());
    }
}
```

# Saída do programa

[Caelum, 2017] Adap.

```
run:
Exceção: Saldo insuficiente, tente um quantia menor
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Exceção

## Alternativa

1. Sempre que puder utilize exceções já existentes, crie suas próprias exceções se você não tiver escolha;
2. No exemplo anterior poderiamos lançar a exceção *IllegalArgumentException* em vez de ter criado a SaldoInsuficienteException.

# Método sacar

## IllegalArgumentException

```
public void sacar(double valor) {
    if (valor > saldo) {
        IllegalArgumentException e = new IllegalArgumentException("Saldo insuficiente, " +
                                                                  "tente um quantia menor");
        throw e; // lança a exceção
    }
    this.saldo -= valor;
}
```

# main

## IllegalArgumentException

1. Devemos capturar a IllegalArgumentException:

```
public static void main(String[] args) {
    Conta c1 = new Conta(1, "Eduardo");

    c1.depositar(100);

    try {
        c1.sacar(200);
    } catch (IllegalArgumentException e) {
        System.out.println("Exceção: " + e.getMessage());
    }
}
```

# Saída do programa

## IllegalArgumentException

```
run:
Exceção: Saldo insuficiente, tente um quantia menor
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Tratamento de Exceções

# Exercícios

## exceção

1. Crie uma classe que crie um vetor de inteiros de 10 posições. Feito isso, permita que o usuário digite valores inteiros a fim de preencher este vetor. Não implemente nenhum tipo controle referente ao tamanho do vetor, deixe que o usuário digite valores até que a entrada 0 seja digitada.
    1. Uma vez digitado o valor 0, o mesmo deve ser inserido no vetor e a digitação de novos elementos deve ser interrompida. Feita toda a coleta dos dados, exiba-os em tela.
    2. Sua classe deve tratar as seguintes exceções:
        1. *ArrayIndexOutOfBoundsException*: quando o usuário informar mais que 10 valores.
        2. *InputMismatchException*: quando o usuário informar um valor que não é numérico.

# Exercícios

## exceção

1. Qual é a saída do código?

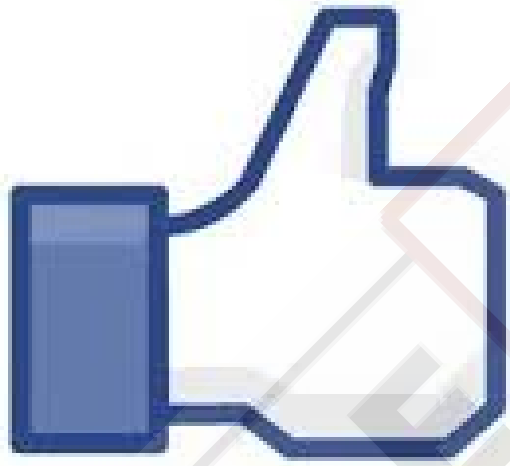
```
public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
        try {
            System.out.println("divisão: " + 10/i);
        } catch (ArithmeticException e) {
            System.out.println("erro na iteração " + i);
        } finally {
            System.out.println("Continua");
        }
    }
}
```

# Exercícios

## exceção

1. Qual é a saída do código?

```
public static void main(String[] args) {
    int i = 0;
    try {
        for (i = -1; i < 5; i++) {
            System.out.println("divisão: " + (10/(i-1)));
        }
    } catch (ArithmeticException e) {
        System.out.println("erro na iteração " + i);
    } finally {
        System.out.println("continua...");
    }
}
```



**MUITO OBRIGADO!!!**

# Referências Bibliográficas I

Caelum (2017).
Java e orientação a objeto.
https://www.caelum.com.br/download/caelum-java-objetos-fj11.pdf.