UNIVERSIDADE FEDERAL DO CEARÁ

TÓPICO 13

PROGRAMAÇÃO ORIENTADA A OBJETOS

Prof. Marcos Vinicius de Andrade Lima
E-mail: marcos.vinicius@ufc.br

Olá!

Sou Marcos Vinicius

No tópico passado nós aprendemos muita coisa com as classes abstratas e interfaces...

Neste tópico deixaremos nossas classes profissionais com o tratamento e aplicação de **exceções**!

> **Os homens erram, os grandes homens confessam que erraram** (Voltaire)

Introdução

INTRODUÇÃO

- Na execução de um programa, várias situações são passíveis de gerar erros
- As operações podem gerar diferentes tipos de erros devido a:

Ambiente de execução
✓ Ex: Acesso a arquivo, acesso à rede, falta de memória

Erros de programação
✓ Ex: Dividir por zero, acessar uma array fora dos limites de seu tamanho

Condições de negócio do programa
✓ Ex: Retirar dinheiro de uma conta sem saldo, diminuir o salário de um funcionário, atribuir uma nota inválida a um aluno.

INTRODUÇÃO

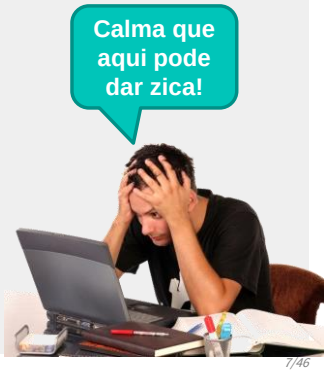
- Na **maioria das linguagens de programação**, o **tratamento de erros** é feito geralmente através de **testes condicionais**, usando if por exemplo, com auxílio de variáveis booleanas
- Vamos relembrar o tratamento de situações onde poderiam ocorrer erros durante a atribuição da nota de um estudante

```
void atribuirNota(int numProva, double nota) {
  if ( nota >= 0 && nota <= 10.0 )
    notas[numProva-1] = nota;
}
```

INTRODUÇÃO

- Se tentarmos atribuir uma nota inválida a uma prova, a atribuição não será realizada. Caso não se consiga efetuar a atribuição, como o método que chamou **atribuirNota** ficará sabendo da não realização da operação?

```
void processarNota(e, numProva, nota) {
  e.atribuirNota(numProva, nota);
}
```



- Vamos retornar, então, um valor booleano para indicar o sucesso ou não da operação

INTRODUÇÃO

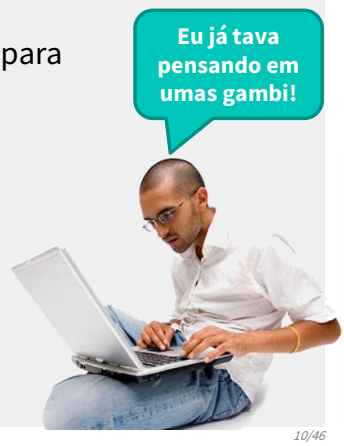
```
boolean atribuirNota(int numProva, double
nota){
  if ( nota >= 0 && nota <= 10.0 ) {
    notas[numProva-1] = nota;
    return true;
  } else return false;
}
```

- No nosso método **processarNota**, podemos agora tratar esse valor booleano

```
void processarNota(e, numProva,nota){
  if (!e.atribuirNota(numProva,nota)) {
  System.out.println("Nota Invalida!");
}
```

INTRODUÇÃO

- Para o sistema ser robusto, seria interessante ter uma maneira de forçar os métodos que chamam o método **atribuirNota()** a tratarem o erro
- Sistemas robustos requerem uma forma mais precisa para tratamento de erros
- Além disso, testes manuais excessivos podem gerar código incompreensível
- Java provê um mecanismo diferenciado, chamado de **Exceções (*Exceptions*)**, que estrutura o tratamento de erros e garante robustez aos sistemas



Exceções

Exceções

- Exceção em Java é um sinal que indica a ocorrência de alguma **condição excepcional**
- O compilador Java **obriga** o usuário da classe a tratar os possíveis erros que um método pode gerar (lembre-se que a linguagem é **fortemente tipada**)

Trabalhando com Exceções

- Para trabalhar com exceções, o seguinte **roteiro** pode ser seguido:

Passo 1
- Identificar a exceção

Passo 2
- Verificar se é preciso definir um novo tipo de exceção

Passo 3
- Identificar os métodos que podem gerar a exceção

Passo 4
- Disparar a exceção de acordo com sua natureza

Passo 5
- Prover o código para tratar a exceção, caso ela ocorra

Criando uma Classe de Exceções

- A primeira coisa a fazer, se for verificada a necessidade de definir um novo tipo de exceção, será especificar uma classe que represente essa exceção
- Essa nova classe de exceção deverá herdar da classe **Exception**
- **Exemplo:**

```
public class NotaInvalidaException extends
Exception
{   }
```

- **ATENÇÃO:** há um padrão para nomes de Exceções, onde os nomes terminam sempre com a palavra '*Exception*'

A CLÁUSULA *THROWS*

- Agora devemos definir os métodos que **podem** disparar essa exceção
- Isso é feito usando a cláusula **throws** (dispara)

```
void atribuirNota(int numProva, double nota) throws
NotaInvalidaException {
  if ( nota >= 0 && nota <= 10.0 ) {
    notas[numProva-1] = nota;
  }
}
```

- Assim, dizemos que o método **atribuirNota()** **pode** disparar uma exceção, possibilidade essa que deve ser convenientemente tratada pelos métodos que chamam o método **atribuirNota()**

A SENTENÇA *THROW*

- O próximo passo é disparar a exceção dentro do método **atribuirNota()**, quando ela ocorrer
- Isso é feito usando a sentença **throw** (disparar)

```
void atribuirNota(int numProva, double nota)
throws NotaInvalidaException {
  if ( nota >= 0 && nota <= 10.0 ) {
    notas[numProva-1] = nota;
  } else
    throw new NotaInvalidaException();
}
```

A Sentença *throw*

- Como exceções em Java são **classes**, antes de disparar uma exceção precisamos criar o objeto. Eis o porquê da construção abaixo:

```
throw new NotaInvalidaException();
```

Dispara a instância (exceção) recém-criada

Cria uma instância da classe NovaInvalidaException

- Agora, quem chamar o método **atribuirNota()** deverá tratar a possibilidade de ocorrer uma exceção durante sua execução!

Hey, Perceba a Diferença!

possibilidade

- **throws:** indica que o método **pode** disparar uma exceção

- **throw:** dispara a exceção

fato

O Bloco *try – catch*

- O bloco **try-catch** tem por finalidade sinalizar onde podem ocorrer exceções (**try**) e onde as exceções podem ser capturadas (**catch**), caso alguma exceção ocorra

```
void processarNota() {
  try {
    Estudante e = new Estudante();
    e.atribuirNota(1, 11.0);
  }
  catch (NotaInvalidaException e){
    System.out.println("Erro na
    atribuição de nota!");
  }
}
```

O Bloco try – catch – finally

- O bloco **try-catch** possui ainda uma terceira parte opcional chamada **finally**
- O bloco **finally**, quando existir, **sempre** será executado, ocorrendo ou não uma exceção
- O bloco **finally** é geralmente usado para fechar conexões com banco de dados, resetar variáveis, enfim, ações que devem ser tomadas caso erros ocorram ou não

Pay Attention!

```
try {
  Estudante e = new Estudante();
  e.atribuirNota(1, 11.0);
} catch (NotaInvalidaException e) {
  System.out.println("Erro na atribuição nota!");
} finally {
  System.out.println("Sempre passa aqui!");
}
```

Sempre é executado!

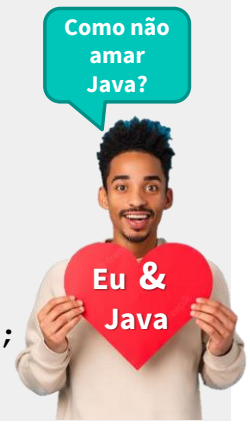
Disparando Várias Exceções

- Todas as exceções disparadas por um único método precisam ser listadas na **assinatura** do método

```
void atribuirNota(int numProva, double nota) throws
NotaInvalidaException, ProvaInvalidaException {
  if ( nota < 0.0 || nota > 10.0 )
    throw new NotaInvalidaException();
  else if ( numProva<1 || numProva>4 )
    throw new ProvaInvalidaException();
  else
    notas[numProva-1] = nota;
}
```


Olha Que Lindo!

```
try {
  Estudante e = new Estudante();
  e.atribuirNota(1, 11.0);
}
catch (NotaInvalidaException e) {
  System.out.println("Nota invalida");
}
catch (ProvaInvalidaException e) {
  System.out.println("Prova invalida!");
}
finally {
  System.out.println("Sempre passa aqui!");
}
```



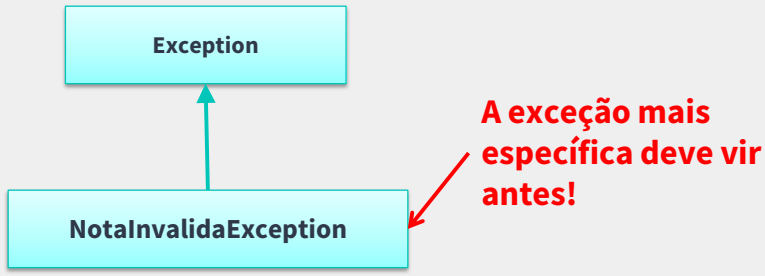
Java permite **vários blocos *catch***, cada um tratando uma exceção específica, **ou nenhum**!

O QUE VOCÊ ME DIZ DESSE CÓDIGO?


```
try {
  Estudante e = new Estudante();
  e.atribuirNota(1, 11.0);
}
catch (Exception e) {
  System.out.println("Erro desconhecido!");
}
catch (NotaInvalidaException e) {
  System.out.println("Erro na atribuição de nota!");
}
finally {
  System.out.println("Sempre passa aqui!");
}
```

VEJA BEM...

- No caso de tratar **várias** exceções usando vários blocos **`catches`**, as exceções **mais específicas** devem vir **antes** das mais genéricas, pois a primeira exceção que o sistema conseguir casar terá seu bloco **`catch`** executado



OLHANDO O TRY – CATCH MAIS DE PERTO

```
try {
  Estudante e = new Estudante();
  e.atribuirNota(1, 11.0);
}
catch (NotaInvalidaException e) {
  System.out.println("Erro na atribuição de nota!");
}

catch (Exception e) {
  System.out.println("Erro desconhecido!");
}

finally {
  System.out.println("Sempre passa aqui!");
}
```

Dois blocos catches, a mais específica vem primeiro

Então, Se Liga!

- Uma vez que ocorreu uma exceção dentro de um bloco **try**, o interpretador Java faz uma **espécie** de **switch**, comparando o objeto exceção gerado com os tipos de exceções existentes nas cláusulas **catch** até encontrar a primeira que se encaixe com seu tipo
- Feito isso, o sistema executa o bloco **catch** escolhido
- Se houver bloco **finally**, ele sempre será executado

Água Mole em Pedra Dura...

- Existem duas forma de tratar uma exceção localmente:
  1. **Capturando-a dentro de um bloco try-catch-finally**
  2. **Declarando que será tratada por quem chamar o método**

Tratando a Exceção Dentro de um Bloco try-catch-finally

```
void processarNota() {
  try {
    Estudante e = new Estudante();
    e.atribuirNota(1, 11.0);
  }
  catch (NotaInvalidaException e) {
  System.out.println("Erro na atribuição de nota!");
  }
}
```


Sendo Tratada por quem Chamar o método

```
void atribuirNota(int numProva, double nota)
throws NotaInvalidaException
{
  if ( nota >= 0 && nota <= 10.0 )
  {
    notas[numProva-1] = nota;
  }
  else
    throw new NotaInvalidaException();
}
```

Quem chamar o método atribuirNota()
deve tratar essa exceção ou repassá-la para ser tratada por quem o chamou, e assim por diante, **até chegar ao método main() e abortar o programa, se nunca for devidamente tratada**.

Tratando a Exceção no Lugar em que foi Gerada

```
void atribuirNota(int numProva, double nota)
{
  try {
    if ( nota >= 0 && nota <= 10.0 )
    {
      notas[numProva-1] = nota;
    }
    else
      throw new NotaInvalidaException();
  }
  catch(NotaInvalidaException e) { ... }
}
```


Exceções Nativas

- Java já disponibiliza uma séria de exceções. Portanto, antes de criar uma nova classe de exceção, é interessante verificar se já existe uma exceção igual ou similar à desejada
- Observe o código abaixo:

```
class TesteExcecao {
  public static void main(String argv[]) {
    int d1,d2;
    d1 = 10;
    d2 = 0;
    System.out.println(d1/d2);
  }
}
```

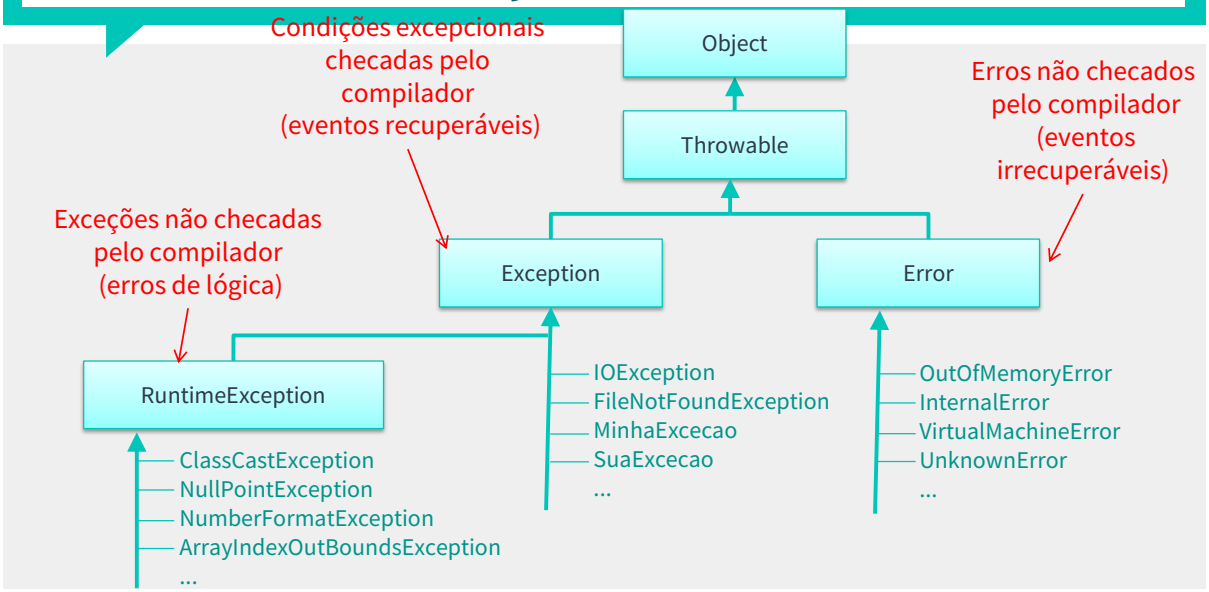

Uma Voadora Saindo...

- O código anterior vai resultar no seguinte erro quando executado:

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero at
TesteExececao.main(TesteExececao.java:6)
Press any key to continue...
```

- Isso ocorre porque tentou-se dividir um inteiro por zero, indicada na mensagem do sistema por **ArithmeticException**
- A exceção **ArithmeticException** já está disponível na linguagem Java, assim com uma série de outras exceções!

Hierarquia de Exceções



Analisando Melhor a Classe *Exception*

- A classe **Exception** representa as exceções que podem ser disparadas, capturadas e tratados pelos programas
- Cada exceção específica é subclasse de **Exception** e contém informações sobre o problema ocorrido
- Os principais métodos para exceções:

Exception()	Cria uma exceção
Exception(String msg)	Cria uma exceção com a mensagem de erro 'msg'
String getMessage()	Retorna a mensagem específica daquela exceção
void printStackTrace()	Imprime a pilha de execução no momento que ocorreu o erro

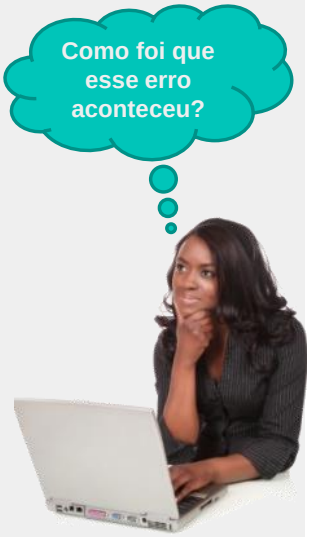
Olha Só Que Legal!

```
class TesteExcecao {
  public static void main(String argv[]) {
    try {
      int d1,d2;
      d1 = 10;
      d2 = 0;
      System.out.println(d1/d2);
    }
    catch (ArithmeticException e) {
     System.out.println ( "erro:" + e.getMessage() );
     e.printStackTrace();
    }
  }
}
```

```
erro:/ by zero
java.lang.ArithmeticException: / by zero
        at TesteExcecao.main(TesteExcecao.java:7)
```

Armazenando Informações nas Exceções

- Novos tipos de exceções podem ser criadas estendendo-se a classe **Exception** ou suas subclasses
- As novas exceções podem disponibilizar informações sobre o erro ocorrido
- Exceções são classes, portanto, podem definir variáveis e métodos, permitindo prover assim, mais informação sobre a causa e a solução para o problema ocorrido



Armazenando Informações nas Exceções

```
public class NotaInvalidaException extends Exception
{
  private double notaInvalida;
  public double getNotaInvalida() {
    return notaInvalida;
  }
  public NotaInvalidaException( double nota ){
    super("Nota Invalida");
    notaInvalida = nota;
  }
}
```

Agora será possível informar qual foi a nota inválida que causou a exceção!

Armazenando Informações nas Exceções

- O método que vai tratar a exceção pode fazer chamada aos métodos disponibilizados pela exceção como se estivesse usando um objeto comum

```
void processarNota() {
  try {
    Estudante e = new Estudante();
    e.atribuirNota(1, 11.0);
  }
  catch (NotaInvalidaException e) {
   System.out.println( e.getMessage() );
   System.out.println("A nota inválida é " +
   e.getNotaInvalida());
  }
}
```


Armazenando Informações nas Exceções

- Agora vamos usar nossa classe aperfeiçoada, criando uma exceção passando como parâmetro a nota que provocou o erro
- Essa informação vai ficar guardada no próprio objeto exceção gerado

```
void atribuirNota(int numProva, double nota)
throws NotaInvalidaException {
  if ( nota >= 0 && nota <= 10.0 )
  {
    notas[numProva-1] = nota;
  }
  else
    throw new NotaInvalidaException( nota );
}
```

Obrigado!

Mais alguma dúvida?

Acesse o **AME** para mais informações e treinamento do **NERDS**!

http://ame2.russas.ufc.br