



BCC221

# Programação Orientada a Objetos

Prof. Marco Antonio M. Carvalho

2014/2



UFOP

# Endereços Importantes

- Site da disciplina:  
<http://www.decom.ufop.br/marco/>
- Moodle:  
[www.decom.ufop.br/moodle](http://www.decom.ufop.br/moodle)
- Lista de e-mails:  
[bcc221-decom@googlegroups.com](mailto:bcc221-decom@googlegroups.com)
- Para solicitar acesso:  
<http://groups.google.com/group/bcc221-decom>



UFOP



# Avisos



# Avisos

UFOP

- A matéria da última prova começa hoje.



# Na aula passada

UFOP

- Caixas de Diálogo
- Componentes Swing
- Swing vs. AWT
- Componentes Leves e Pesados
- Exibindo Textos e Imagens em uma Janela
- Especificando o Leiaute
- Criando Labels
- Campos de Texto e Introdução à Manipulação de Eventos
- Tipos Comuns de Eventos e Listener Interfaces
- Botões
  - Botões que Mantêm seu Estado
- Caixas de Marcação
- Botões de Opção
- Caixas de Combinação
- Listas
  - Listas de Seleções Múltiplas
- Manipulação de Eventos do Mouse
- Classes Adaptadoras
- Subclasse JPanel para Desenhar com o Mouse
- Manipulação de Eventos do Teclado
- Gerenciadores de Leiaute
- Áreas de Texto



# Na aula de hoje

UFOP

- Tratamento de Exceções
- *try e catch*
- Modelo de Terminação
- Cláusula *throws*
- Quando Utilizar Exceções?
- Hierarquia de Exceções Java
- Blocos *finally*
- *throw*
- Desfazendo a Pilha
- *printStackTrace, getStackTrace e getMessage*
- Exceções Encadeadas
- Declarando Novos Tipos de Exceções
- Pré-Condições e Pós-Condições
- Asserções



# Tratamento de Exceções

UFOP

- Uma exceção é uma indicação de um problema que ocorre durante a **execução** de um programa
  - Tratar as exceções permite que um programa continue executando como se não houvesse ocorrido um erro;
  - Programas **robustos** e **tolerantes a falhas**.
- O estilo e os detalhes do tratamento de exceções em Java é baseado parcialmente do encontrado em C++.



# Tratamento de Exceções

UFOP

- O exemplo a seguir apresenta um problema comum
  - Divisão por zero.
- Exceções são disparadas, e o programa é incapaz de tratá-las.



# DivideByZeroNoExceptionHandling.java

UFOP

```
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling
{
    //demonstra o disparo de uma exceção quando ocorre uma divisão por zero
    public static int quotient( int numerator, int denominator )
    {
        return numerator / denominator; // possível divisão por zero
    }

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

        System.out.print( "Please enter an integer numerator: " );
        int numerator = scanner.nextInt();
        System.out.print( "Please enter an integer denominator: " );
        int denominator = scanner.nextInt();

        int result = quotient( numerator, denominator );
        System.out.printf("\nResult: %d / %d = %d\n", numerator, denominator, result);
    }
}
```



# Entrada e Saída

UFOP

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main"
java.lang.ArithmetricException: / by zero
at DivideByZeroNoExceptionHandling.quotient(
    DivideByZeroNoExceptionHandling.java:10)
at DivideByZeroNoExceptionHandling.main(
    DivideByZeroNoExceptionHandling.java:22)
```



# Tratamento de Exceções

UFOP

- Quando o denominador é nulo, várias linhas de informação são exibidas em resposta à entrada inválida
  - Esta informação é chamada de ***Stack Trace***;
  - Inclui o nome da exceção em uma mensagem descritiva que indica o problema ocorrido e também a **cadeia de chamadas aos métodos (*method-call stack*)** no momento em que ocorreu o erro;
  - O *stack trace* inclui o caminho da execução que levou a exceção método por método;
- A segunda linha indica que ocorreu uma exceção ***ArithmeticException***
  - Pode ser lançada por diferentes problemas aritméticos;
  - A próxima linha indica que essa exceção ocorreu como resultado de uma divisão por zero.



# Tratamento de Exceções

UFOP

- A partir da última linha do *stack trace*, vemos que a exceção foi detectada inicialmente na linha 22 do *main*
  - Cada linha contém o nome da classe e o método, seguidos pelo nome do arquivo e o número da linha;
- Na linha acima, a exceção ocorre na linha 10, no método *quotient*;
- A linha do topo da cadeia de chamadas indica o ponto de disparo
  - O ponto inicial em que ocorreu a exceção;
  - Linha 10 do método *quotient*.



# Entrada e Saída

UFOP

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception in thread "main"
```

**java.util.InputMismatchException**

```
at java.util.Scanner.throwFor(Unknown Source)
at java.util.Scanner.next(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at DivideByZeroNoExceptionHandling.main(
    DivideByZeroNoExceptionHandling.java:20)
```



# Tratamento de Exceções

- Ao informarmos um tipo diferente do esperado, uma exceção ***InputMismatchException*** é lançada;
- A partir do final do *stack trace*, vemos que a exceção foi detectada na linha 20 do *main*;
- Na linha superior, a exceção ocorre no método *nextInt*
  - Ao invés de aparecer o nome do arquivo e número da linha, aparece o texto “*Unknown Source*”;
  - Significa que a JVM não possui acesso ao código fonte em que ocorreu a exceção.
- Nestes exemplos, a execução do programa foi interrompida pelas exceções
  - Nem sempre este é o caso.

*try e catch*



# *try e catch*

UFOP

- O exemplo a seguir utiliza o tratamento de exceções para processar quaisquer exceções *ArithmeticException* e *InputMismatchException*
  - Se o usuário cometer um erro, o programa **captura e trata** a exceção;
  - Neste caso, permite que o usuário informe os dados novamente.



# DivideByZeroWithExceptionHandling.java

UFOP

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
    //demonstra o disparo de uma exceção quando ocorre uma divisão por zero
    public static int quotient( int numerator, int denominator ) throws
        ArithmeticException
    {
        return numerator / denominator; // possivel divisao por zero
    }

    public static void main( String args[] )
    {
        Scanner scanner = new Scanner( System.in );
        boolean continueLoop = true; // determina se e necessario mais entrada
```



# DivideByZeroWithExceptionHandling.java

UFOP

```
do
{
    try // le dois numeros e calcula a divisao
    {
        System.out.print( "Please enter an integer numerator: " );
        int numerator = scanner.nextInt();
        System.out.print( "Please enter an integer denominator: " );
        int denominator = scanner.nextInt();

        int result = quotient( numerator, denominator );
        System.out.printf( "\nResult: %d / %d = %d\n", numerator, denominator, result );
        continueLoop = false; // entrada ok, fim do loop
    }
}
```



# DivideByZeroWithExceptionHandling.java

UFOP

```
catch ( InputMismatchException inputMismatchException )
{
    System.err.printf( "\nException: %s\n", inputMismatchException );
    scanner.nextLine(); // descarta a entrada para o usuario tentar de novo
    System.out.println("You must enter integers. Please try again.\n" );
}
catch ( ArithmeticException arithmeticException )
{
    System.err.printf( "\nException: %s\n", arithmeticException );
    System.out.println("Zero is an invalid denominator. Please try again.\n" );
}
} while ( continueLoop );
}
```



# Entrada e Saída

UFOP

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
```

```
Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7
```

```
Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
```

```
Exception: java.util.InputMismatchException
You must enter integers. Please try again.
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7
```

```
Result: 100 / 7 = 14
```



# *try e catch*

UFOP

- A classe ***ArithmeticException*** não precisa ser importada porque está localizada no pacote *java.lang*
  - A classe ***InputMismatchException*** precisa ser importada.
- O bloco ***try*** neste exemplo é seguido de dois blocos ***catch***
  - Um para cada tipo de exceção.



# try e catch

UFOP

- Um bloco *try* engloba o código que possa disparar uma exceção
  - Consiste da palavra *try* e de um bloco de código delimitado por { e };
  - No exemplo, o método *nextInt* lança uma exceção *InputMismatchException* se o valor lido não for um inteiro;
  - No método *quotient*, a JVM lança uma exceção *AirthmeticException* caso o denominador seja nulo.



# *try e catch*

- Um *bloco catch* captura e trata uma exceção
  - Começa com a palavra *catch*, seguido por um único parâmetro entre parênteses e um bloco de código entre { e };
  - O parâmetro especifica o tipo da exceção a ser capturada.
- Pelo menos um *bloco catch* ou um *bloco finally* devem seguir imediatamente um *bloco try*
  - O bloco cujo objeto parâmetro seja do mesmo tipo ao da exceção lançada será executado;
  - O bloco executado pode interagir com o objeto da exceção
    - Como no exemplo, invocar o método *toString*.



# try e catch

UFOP

- Uma **exceção não capturada** é uma exceção que ocorre e não há bloco *catch* correspondente
  - Java utiliza um modelo de diversas linhas de execução para programas (*multithread*);
  - Se uma destas linhas de execução (*thread*) lançar uma exceção não capturada, somente ela será suspensa;
  - Pode haver cascateamento da suspensão ou comportamento inesperado do programa.

# Modelo de Terminação



# Modelo de Terminação

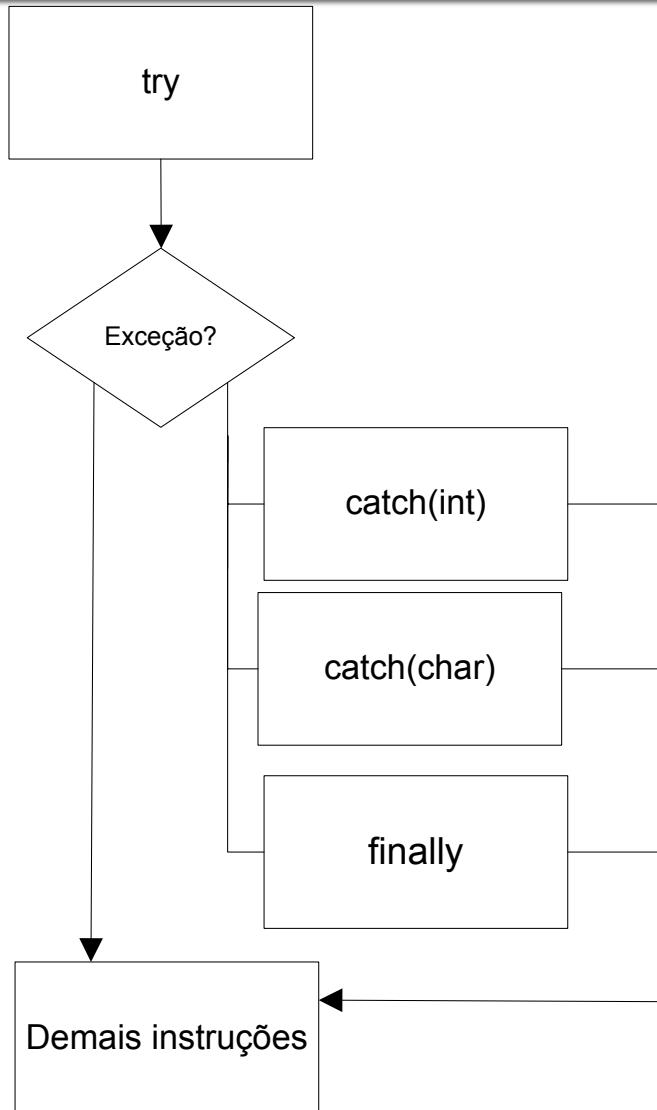
UFOP

- Se uma exceção ocorre em um bloco *try*, o mesmo é interrompido e o controle do programa é transferido para o bloco *catch* adequado ou para o bloco *finally*, se disponível;
- Depois do tratamento da exceção, o controle do programa não retorna ao ponto de ocorrência da exceção
  - O bloco *try* expirou, inclusive perdendo as variáveis locais.
- O controle passa para a instrução seguinte ao último bloco *catch/finally*
  - Chamado de **Modelo de Terminação**.
- Se nenhuma exceção ocorrer, todos os blocos *catch* e o *finally* são ignorados.



# Modelo de Terminação

UFOP



# Cláusula *throws*



# Cláusula *throws*

UFOP

- Uma cláusula *throws* especifica as exceções que um método lança
  - Aparece entre a lista de parâmetros e o corpo do método;
  - Contém uma lista de valores separados por vírgula, em que cada valor é um tipo de exceção;
  - As exceções podem ser lançadas explicitamente dentro do próprio método ou por outros métodos chamados dentro do primeiro.
- Se sabemos que um método pode lançar exceções, devemos incluir o código de tratamento de exceções adequado
  - O próprio método não tratará a exceção;
  - A documentação do método deve fornecer maiores detalhes sobre as causas do lançamento de exceções.

# Quando Utilizar Exceções?



# Quando Utilizar Exceções?

UFOP

- Erros síncronos (na execução de uma instrução)
  - Índice de vetor fora dos limites;
  - *Overflow* aritmético (valor fora dos limites do tipo);
  - Divisão por zero;
  - Parâmetros inválidos;
  - Alocação de memória excessiva ou indisponível.
- Exceções não devem ser utilizadas para erros assíncronos (paralelos à execução do programa)
  - Erros de I/O;
  - Cliques de mouse e pressionamento de teclas.
- Obviamente, exceções não tratam erros em **tempo de compilação**.

# Hierarquia da Exceções Java



# Hierarquia da Exceções Java

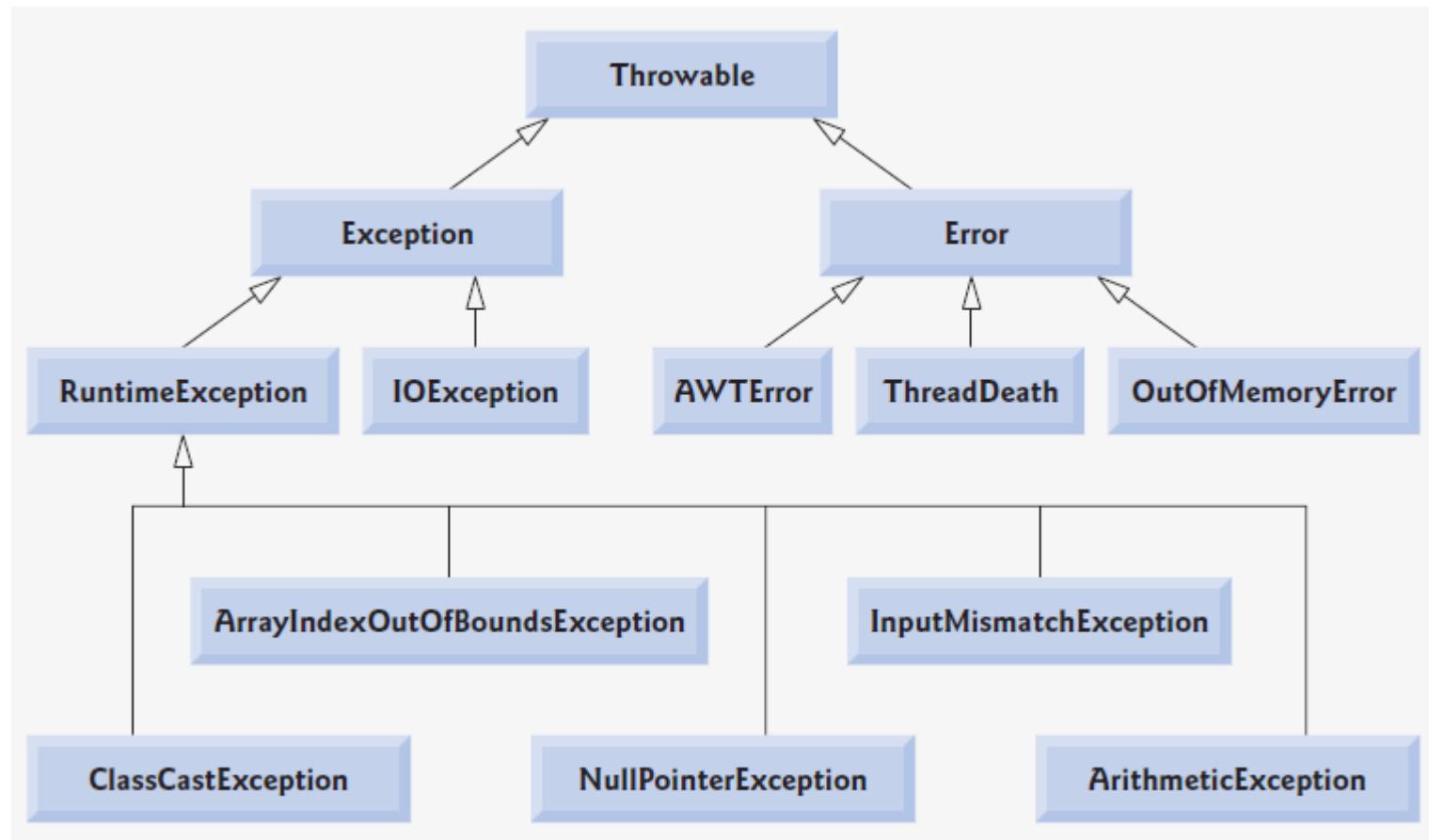
UFOP

- Todas as classes de exceção Java herdam direta ou indiretamente da classe ***Exception***
  - É possível estender esta hierarquia para criar nossas próprias classes de exceção;
  - A hierarquia específica é iniciada pela classe ***Throwable*** (uma subclasse de *Object*)
    - Somente objetos *Throwable* podem ser utilizados com o mecanismo de tratamento de exceções.
  - A classe ***Exception*** e suas subclasses representam situações excepcionais que ocorrem em um programa e que podem ser capturadas por ele próprio;
  - A classe ***Error*** e suas subclasses representam situações anormais que podem acontecer na JVM.



# Hierarquia da Exceções Java

UFOP



*catch-or-declare*



# *catch-or-declare*

UFOP

- Códigos “válidos” em Java deve honrar o requisito *catch-or-declare*
  - Ou (*catch-or-specify*).
- Códigos que possam lançar certas exceções devem cumprir com uma das opções abaixo:
  - Possuir uma estrutura *try/catch* que manipule a exceção;
  - Declarar que o método correspondente pode lançar exceções, através de uma cláusula *throws*.



# *catch-or-declare*

UFOP

- Códigos que não honrem o *catch-or-declare* não compilam
  - Nem todas exceções estão sujeitas ao *catch-or-declare*.
- Java distingue entre três categorias de exceções:
  - **Exceções verificadas**
    - Precisam ser listadas na cláusula *throws*.
  - **Erros ou Exceções não verificadas**
    - Não precisam ser listadas na cláusula *throws*.
  - **Exceções em tempo de execução**
    - Não precisam ser listadas na cláusula *throws*.



# *catch-or-declare*

UFOP

## ■ **Exceções verificadas**

- Um código bem escrito pode antecipá-las e se recuperar.

## ■ **Erros ou Exceções não verificadas**

- Condições excepcionais externas ao código;
- Não é possível antecipá-las ou recuperar-se.

## ■ **Exceções em tempo de execução**

- Condições excepcionais internas ao código.
- *Bugs.*
- Devem ser corrigidas no próprio código para evitar a exceção.



# *catch-or-declare*

UFOP

## Não Verificadas

- `ArrayIndexOutOfBoundsException`
- `ClassCastException`
- `IllegalArgumentException`
- `IllegalStateException`
- `NullPointerException`
- `NumberFormatException`
- `AssertionError`
- `ExceptionInInitializerError`
- `StackOverflowError`
- `NoClassDefFoundError`

## Verificadas

- `Exception`
- `IOException`
- `FileNotFoundException`
- `ParseException`
- `ClassNotFoundException`
- `CloneNotSupportedException`
- `InstantiationException`
- `InterruptedException`
- `NoSuchMethodException`
- `NoSuchFieldException`



# *catch-or-declare*

UFOP

- Esta distinção é importante, porque o compilador Java força o *catch-or-declare* para exceções verificadas
- O tipo da exceção determina quando uma exceção é verificada ou não
  - Todas as exceções que herdam direta ou indiretamente da classe *RuntimeException* são exceções não verificadas;
  - Todas as exceções que herdam direta ou indiretamente da classe *Exception* mas não da classe *RuntimeException* são exceções verificadas.



# catch-or-declare

UFOP

- O compilador verifica cada declaração e chamada a método para determinar se ele lança alguma exceção
  - Caso positivo, o compilador confere se a exceção verificada é capturada ou declarada na cláusula *throws*;
  - Se o requerimento *catch-or-declare* não for satisfeito, o compilador gerará um erro.
- Se a exceção é não verificada, não ocorre esta verificação.

**Blocos *finally***



# Blocos *finally*

UFOP

- Programas que obtêm certos tipos de recursos devem devolvê-los aos sistema explicitamente para evitar a perda dos mesmos (*resource leaks*);
- O ***bloco finally*** é opcional, e se presente, é colocado depois do último bloco *catch*
  - Consiste da palavra *finally* seguida por um bloco de comandos entre { e }.
- Se houver um bloco *finally*, Java garante que ele será executado
  - Independentemente de qualquer exceção ser ou não disparada no bloco *try* correspondente;
  - Também será executado se um bloco *try* for encerrado com uma instrução *return*, *break* ou *continue*;
  - Porém, não executará se o método ***System.exit*** for invocado.



# Blocos *finally*

UFOP

- Justamente por quase sempre se executado, um bloco *finally* contém códigos de liberação de recursos
  - Por exemplo, fechar conexões de rede, arquivos, etc.
- O exemplo a seguir demonstra a execução de um bloco *finally* mesmo uma exceção não sendo lançada no bloco *try* correspondente.



# UsingExceptions.java

UFOP

```
public class UsingExceptions
{
    public static void main( String args[] )
    {
        try
        {
            throwException();
        }
        catch ( Exception exception ) // excecao lancada por throwException
        {
            System.err.println( "Exception handled in main" );
        }

        doesNotThrowException();
    }
}
```



# UsingExceptions.java

UFOP

```
// demonstra try...catch...finally
public static void throwException() throws Exception
{
    try // lança uma exceção e imediatamente a captura
    {
        System.out.println( "Method throwException" );
        throw new Exception(); // gera a exceção
    }
    catch ( Exception exception ) // captura a exceção lançada no try
    {
        System.err.println("Exception handled in method throwException" );
        throw exception; // lança novamente para mais processamento
        // qualquer código aqui seria inatingível
    }
    finally // executa independentemente do que ocorre no try...catch
    {
        System.err.println( "Finally executed in throwException" );
    }
    // qualquer código aqui seria inatingível
}
```



# UsingExceptions.java

UFOP

```
// demonstra o finally quando nao ocorre excecao
public static void doesNotThrowException()
{
    try // o bloco try nao lancia excecoes
    {
        System.out.println( "Method doesNotThrowException" );
    }
    catch ( Exception exception ) // nao e executado
    {
        System.err.println( exception );
    }
    finally // executa independentemente do que ocorre no try...catch
    {
        System.err.println( "Finally executed in doesNotThrowException" );
    }

    System.out.println( "End of method doesNotThrowException" );
}
```



# Saída

UFOP

Method throwException

Exception handled in method throwException

Finally executed in throwException

Exception handled in main

Method doesNotThrowException

Finally executed in doesNotThrowException

End of method doesNotThrowException



# Blocos *finally*

UFOP

- Note o uso de ***System.err*** para exibir os dados
  - Direciona o conteúdo para a fluxo padrão de erros;
  - Se não for redirecionado, os dados serão exibidos no *prompt* de comando.

*throw*



# throw

UFOP

- A instrução **throw** é executada para indicar que ocorreu uma exceção
  - Lança uma exceção.
- Até aqui tratamos exceções lançadas por outros métodos
  - Podemos lançar as nossas próprias;
  - Deve ser especificado um objeto a ser lançado
    - De qualquer classe derivada da classe *Throwable*.
- Exceções podem ser relançadas
  - Quando um bloco *catch* recebe uma exceção, mas é incapaz de processá-la totalmente, ele pode relançá-la para outro bloco *try-catch* mais externo;
  - Blocos *finally* não podem relançar exceções.

# Desfazendo a Pilha



# Desfazendo a Pilha

UFOP

- Quando uma exceção é lançada mas não capturada em um determinado escopo, a pilha de chamadas de métodos é desfeita passo a passo
  - A cada passo, tenta-se capturar a exceção;
  - Este processo é chamado de ***stack unwinding***.
- O exemplo a seguir demonstra este processo.



# UsingExceptions.java

UFOP

```
public class UsingExceptions
{
    public static void main( String args[] )
    {
        try
        {
            throwException();
        }
        catch ( Exception exception ) // exceção lançada em throwException
        {
            System.err.println( "Exception handled in main" );
        }
    }
}
```



# UsingExceptions.java

UFOP

```
// throwException lanza uma excecao que nao e capturada neste metodo
public static void throwException() throws Exception
{
    try // lanza uma excecao e a captura no main
    {
        System.out.println( "Method throwException" );
        throw new Exception(); // gera a excecao
    }
    catch ( RuntimeException runtimeException ) // captura o tipo incorreto
    {
        System.err.println( "Exception handled in method throwException" );
    }
    finally // sempre sera executado
    {
        System.err.println( "Finally is always executed" );
    }
}
```



# Saída

UFOP

Method throwException  
Finally is always executed  
Exception handled in main



# Desfazendo a Pilha

UFOP

- O método *main* invoca o método *throwException* dentro de um bloco *try...catch*
  - Por sua vez, o método lança uma exceção em seu próprio bloco *try...catch*;
  - No entanto, o *catch* não captura a exceção, pois o tipo não é adequado;
  - A pilha é desfeita, volta-se ao *main* e então o bloco *catch* captura a exceção.

*printStackTrace,  
getStackTrace e getMessage*



# *printStackTrace*, *getStackTrace* e *getMessage*

UFOP

- A classe **Throwable** fornece três métodos para obtermos informações sobre exceções:
  - *printStackTrace*: exibe a *stack trace* no fluxo de erro padrão;
  - *getStackTrace*: retorna os dados que serão exibidos pelo método anterior;
  - *getMessage*: retorna uma *string* descritiva armazenada na exceção.
- O exemplo a seguir demonstra a utilização destes métodos.



# UsingExceptions.java

UFOP

```
public class UsingExceptions
{
    public static void main( String args[] )
    {
        try
        {
            method1();
        }
        catch ( Exception exception ) // captura a excecao lancada emmethod1
        {
            System.err.printf( "%s\n\n", exception.getMessage() );
            exception.printStackTrace(); // imprime o stack trace

            // obtém a informacao do stack trace
            StackTraceElement[] traceElements = exception.getStackTrace();

            System.out.println( "\nStack trace from getStackTrace:" );
            System.out.println( "Class\t\tFile\t\tLine\tMethod" );
        }
    }
}
```



# UsingExceptions.java

UFOP

```
// itera pelos elementos para obter a descricao da excecao
for ( StackTraceElement element : traceElements )
{
    System.out.printf( "%s\t", element.getClassName() );
    System.out.printf( "%s\t", element.getFileName() );
    System.out.printf( "%s\t", element.getLineNumber() );
    System.out.printf( "%s\n", element.getMethodName() );
}
```



# UsingExceptions.java

UFOP

```
// lança a exceção de volta para o main
public static void method1() throws Exception
{
    method2();
}

// lança a exceção de volta para o method1
public static void method2() throws Exception
{
    method3();
}

// lança a exceção de volta para o method2
public static void method3() throws Exception
{
    throw new Exception( "Exception thrown in method3" );
}
```



# Saída

UFOP

Exception thrown in method3

```
java.lang.Exception: Exception thrown in method3
    at UsingExceptions.method3(UsingExceptions.java:49)
    at UsingExceptions.method2(UsingExceptions.java:43)
    at UsingExceptions.method1(UsingExceptions.java:37)
    at UsingExceptions.main(UsingExceptions.java:10)
```

Stack trace from getStackTrace:

Class	File	Line	Method
UsingExceptions	UsingExceptions.java	49	method3
UsingExceptions	UsingExceptions.java	43	method2
UsingExceptions	UsingExceptions.java	37	method1
UsingExceptions	UsingExceptions.java	10	main



# *printStackTrace, getStackTrace e getMessage*

UFOP

- Objetos da classe ***StackTraceElement*** armazenam informações da *stack trace* sobre a exceção;
- Possuem os métodos:
  - ***getClassName***: retorna o nome da classe;
  - ***getFileName***: retorna o nome do arquivo;
  - ***getLineNumber***: retorna o número da linha;
  - ***getMethodName***: retorna o nome do método.

# Exceções Encadeadas



# Exceções Encadeadas

UFOP

- Às vezes um bloco *catch* captura um tipo de exceção e então lança uma nova exceção de outro tipo
  - Para indicar que uma exceção específica do programa ocorreu;
- Nas versões mais antigas do Java não havia um mecanismo que juntasse a informação da primeira exceção com a segunda
  - Fornecendo assim a informação completa sobre a exceção;
  - Como um *stack trace* completo.
- As **exceções encadeadas** permitem que o objeto de uma exceção mantenha toda a informação;
- O exemplo a seguir demonstra este recurso.



# UsingChainedExceptions.java

UFOP

```
public class UsingChainedExceptions
{
    public static void main( String args[] )
    {
        try
        {
            method1();
        }
        catch ( Exception exception ) // excecao lancada por method1
        {
            exception.printStackTrace();
        }
    }

    // lancia uma excecao de volta ao main
    public static void method1() throws Exception
    {
        try
        {
            method2();
        }
        catch ( Exception exception ) // excecao lancada por method2
        {
            throw new Exception( "Exception thrown in method1", exception );
        }
    }
}
```



# UsingChainedExceptions.java

UFOP

```
// lança uma exceção de volta ao method1
public static void method2() throws Exception
{
    try
    {
        method3();
    }
    catch ( Exception exception ) // exceção lançada por method3
    {
        throw new Exception( "Exception thrown in method2", exception );
    }
}

// lança uma exceção de volta ao method2
public static void method3() throws Exception
{
    throw new Exception( "Exception thrown in method3" );
}
```



# Saída

UFOP

```
java.lang.Exception: Exception thrown in method1
at UsingChainedExceptions.method1 (UsingChainedExceptions.java:27)
at UsingChainedExceptions.main (UsingChainedExceptions.java:10)
```

Caused by: java.lang.Exception: Exception thrown in method2  
at UsingChainedExceptions.method2 (UsingChainedExceptions.java:40)  
at UsingChainedExceptions.method1 (UsingChainedExceptions.java:23)  
... 1 more

Caused by: java.lang.Exception: Exception thrown in method3  
at UsingChainedExceptions.method3 (UsingChainedExceptions.java:47)  
at UsingChainedExceptions.method2 (UsingChainedExceptions.java:36)  
... 2 more



# Exceções Encadeadas

UFOP

- O programa consiste em 4 métodos
  - Cada um com um bloco *try* em que invoca o próximo método em ordem crescente;
  - Exceto *method3*, que lança uma exceção;
  - À medida em que a pilha de chamadas dos métodos é desfeita, cada bloco *catch* captura a exceção e lança uma nova;
  - Um dos construtores da classe ***Exception*** possui dois argumentos
    - Uma mensagem personalizada;
    - Um objeto *Throwable*, que identifica a causa da exceção.
  - No exemplo, a causa da exceção anterior é utilizada como parâmetro para o construtor.

# Declarando Novos Tipos de Exceções

# Declarando Novos Tipos de Exceções



UFOP

- Normalmente, os programadores Java utilizam as classes da API Java e de terceiros
  - Tipicamente os métodos destas classes lançam as exceções apropriadas quando ocorre um erro.
- Quando escrevemos classes que serão distribuídas, é útil declarar nossas próprias classes de exceções que podem ocorrer
  - Caso as exceções não sejam contempladas na API Java.
- Uma nova classe de exceções deve estender uma classe de exceções já existente
  - Para garantir que ela funcionará com o mecanismo de tratamento de exceções.

# Declarando Novos Tipos de Exceções



UFOP

- Como qualquer outra classe, uma classe de exceções contém atributos e métodos
  - Porém, tipicamente contém apenas dois construtores
    - Um que não possui argumentos e informa uma mensagem padrão ao construtor da superclasse;
    - Um que possui receber uma *string* com uma mensagem personalizada como argumento e a repassa ao construtor da superclasse.
- Antes de criar uma nova classe, é necessário analisar a API Java para decidir qual classe deve ser utilizada como superclasse
  - É uma boa prática que seja uma classe relacionada com a natureza da exceção.

# Pré-Condições e Pós-Condições



# Pré-Condições e Pós-Condições

UFOP

- Programadores gastam um bom tempo realizando a manutenção e a depuração de códigos;
- Para facilitar estas tarefas, podemos especificar os estados esperados antes e depois da execução de um método
  - Estes estados são chamados de **pré-condições** e **pós-condições**.
- Uma pré-condição deve ser verdadeira quando um método é invocado
  - Restrições quanto aos parâmetros e outras expectativas quanto ao estado atual do programa;
  - Se as pré-condições não são atendidas, o comportamento do método é indefinido.



# Pré-Condições e Pós-Condições

UFOP

- Uma pós-condição deve ser verdadeira após o retorno de um método
  - Descreve restrições quanto ao valor de retorno e outros efeitos possíveis;
  - Quando invocamos um método, assumimos que ele atende todas as pós-condições.
- Nossos códigos devem documentar todas as pós-condições
  - Assim, os usuários saberão o que esperar de uma execução de cada método;
  - Também ajuda a desenvolver o próprio código.
- Quando pré-condições e pós-condições não são atendidas, os métodos tipicamente lançam exceções.



# Pré-Condições e Pós-Condições

UFOP

- Por exemplo, o método *charAt* da classe *String*, que recebe um índice como argumento
  - **Pré-condição:** o argumento deve ser maior ou igual a zero, e menor que o comprimento da *string*;
  - **Pós-condição:** retornar o caractere no índice indicado;
  - Caso contrário, o método lançará a exceção *IndexOutOfBoundsException*.
- Acreditamos que o método atenderá sua pós-condição se garantirmos a pré-condição
  - Sem nos preocuparmos com os detalhes do método.

# Asserções



# Asserções

UFOP

- Quando implementamos e depuramos uma classe, é útil criarmos certas condições que devem ser verdadeiras em determinados pontos do código
  - Estas condições são chamadas de **asserções**;
  - Nos ajudam a capturar eventuais *bugs* e a identificar possíveis erros de lógica;
  - Pré-condições e pós-condições são asserções relativas aos métodos.
- Java inclui duas versões da instrução ***assert*** para validação de asserções.



# Asserções

UFOP

- A instrução *assert* avalia uma expressão booleana e determina se a mesma é verdadeira ou falsa;
- A primeira forma é

`assert expressao;`

- Uma exceção ***AssertionError*** é lançada caso a expressão seja falsa;
- A segunda forma é

`assert expressao1 : expressao2;`

- Uma exceção ***AssertionError*** é lançada caso a primeira expressão seja falsa, com a segunda expressão como mensagem de erro.



# Asserções

UFOP

- Podemos utilizar asserções para implementar pré-condições e pós-condições
  - Ou para verificar quaisquer estados intermediários que nos ajudem a garantir que o código funciona corretamente.
- O exemplo a seguir demonstra a utilização de asserções.



# AssertTest.java

UFOP

```
import java.util.Scanner;

public class AssertTest
{
    public static void main( String args[] )
    {
        Scanner input = new Scanner( System.in );

        System.out.print( "Enter a number between 0 and 10: " );
        int number = input.nextInt();

        // assertao para verificar que o valor absoluto esta entre 0 e 10
        assert ( number >= 0 && number <= 10 ) : "bad number: " + number;

        System.out.printf( "You entered %d\n", number );
    }
}
```



# Saída

UFOP

Enter a number between 0 and 10: **5**

You entered 5

Enter a number between 0 and 10: **50**

Exception in thread "main" java.lang.AssertionError:  
bad number: 50  
at AssertTest.main(AssertTest.java:15)



# Asserções

UFOP

- Por padrão, as asserções são desabilitadas quando o programa é executado
  - Reduzem a performance e são inúteis para o usuário final.
- Para habilitar as asserções, é necessário utilizar a opção **-ea** na linha de comando  
***java -ea AssertTest***
- Em uma execução normal, não podem ser encontrados *AssertionErrors*
  - Nunca devem ser tratados, porém, corrigidos.



UFOP



# Perguntas?



# Na próxima aula

UFOP

- Genéricos



UFOP



**FIM**