



Computação Orientada a Objetos

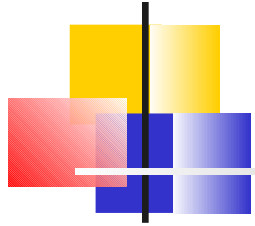
Tratamento de Exceções

Este material é baseado em *slides* da Profa. Patrícia R. Oliveira



Exceções

- Uma exceção é uma indicação de um problema que ocorre durante a execução de um programa
- O tratamento de exceções permite aos programadores criar aplicativos que podem resolver (tratar) exceções
- Está relacionado tanto ao tratamento de erros irre recuperáveis do sistema quanto de situações alternativas à seqüência típica de eventos;



Visão Geral

- Os programas costumam testar condições para determinar como a execução de um programa deve prosseguir

Realize uma tarefa

Se a tarefa anterior não tiver sido executada corretamente

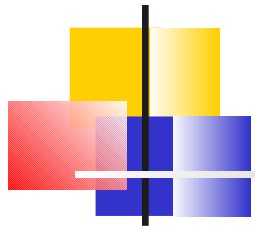
Realize processamento de erro

Realize a próxima tarefa

Se a tarefa anterior não tiver sido executada corretamente

Realize processamento de erro

...



Visão Geral

- Embora funcione, mesclar a lógica do programa com o tratamento de erros pode dificultar a leitura, modificação, manutenção e a depuração dos programas

Realize uma tarefa

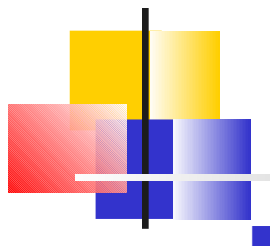
Se a tarefa anterior não tiver sido executada corretamente

Realize processamento de erro

Realize a próxima tarefa

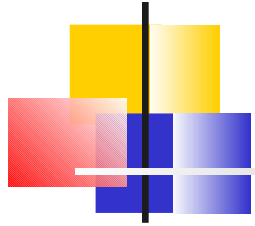
Se a tarefa anterior não tiver sido executada corretamente

Realize processamento de erro



Visão Geral

- O tratamento de exceções permite aos programadores remover da “linha principal” de execução do programa o código do tratamento de erros
 - aprimora a clareza do programa
- Os programadores podem escolher quais exceções serão tratadas:
 - todas as exceções
 - todas as exceções de um certo tipo
 - todas as exceções de um grupo de tipos relacionados (hierarquia de herança)



Visão Geral

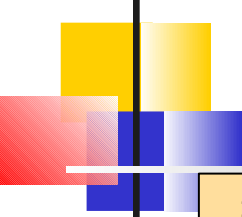
- O tratamento de exceções reduz a probabilidade de que erros sejam negligenciados
- Resultado: produtos de software mais robustos e tolerantes a falhas!



Conceitos básicos

- Diz-se que uma exceção é lançada (isto é, a exceção ocorre) quando um método detecta um problema e é incapaz de tratá-lo
- Quando uma exceção ocorre dentro de um método, o método cria um objeto do tipo exceção:
 - contém informação a respeito do evento, incluindo seu tipo e o estado do programa no momento da ocorrência.

Ex: Divisão por zero *SEM* tratamento de exceções



```
import java.util.Scanner;
public class DivideByZeroNoExceptionHandling
{
    // demonstra o lançamento de uma exceção quando ocorre uma divisão por zero
    public static int quociente( int numerador, int denominador )
    {
        return numerador / denominador; // possivel divisao por zero
    } // fim de método quociente

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

        System.out.print( " Entre com um numerador inteiro: " );
        int numerador = scanner.nextInt();
        System.out.print( " Entre com um denominador inteiro: " );
        int denominador = scanner.nextInt();

        int result = quociente( numerador, denominador );
        System.out.printf(
            "\nResult: %d / %d = %d\n", numerador, denominador, result );
    } // fim de main
} // fim da classe DivideByZeroNoExceptionHandling
```


Ex: Divisão por zero *SEM*

tratamento de exceções

- Execução 1: Divisão bem sucedida!

Entre com um numerador inteiro: 100

Entre com um denominador inteiro: 7

Result: $100/7 = 14$

Ex: Divisão por zero *SEM*

tratamento de exceções

- Execução 2: Usuário insere o valor 0 como denominador...

```
Entre com um numerador inteiro: 100
```

```
Entre com um denominador inteiro: 0
```

```
Exception in thread "main" java.lang.ArithmeticException: /by zero
```

```
at DivideByZeroNoExceptionHandling.quociente(DivideByZeroNoExceptionHandling.  
java:8)
```

```
at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.  
java:20)
```

Ex: Divisão por zero *SEM*

tratamento de exceções

- Execução 2: Usuário insere o valor 0 como denominador...

Entre com um numerador inteiro: 100

Entre com um denominador inteiro: 0

```
Exception in thread "main" java.lang.ArithmeticException: /by zero
at DivideByZeroNoExceptionHandling.quociente(DivideByZeroNoExceptionHandling.
    java:8)
at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.
    java:20)
```

Rastreamento de pilha: mensagem que inclui

- 2) nome da exceção (**`java.lang.ArithmeticException`**)
- 3) o problema que ocorreu (**`/by zero`**)
- 4) o caminho de execução que resultou na exceção, método por método

Ex: Divisão por zero *SEM*

tratamento de exceções

- Execução 2: Usuário insere o valor 0 como denominador...

Entre com um numerador inteiro: 100

Entre com um denominador inteiro: 0

```
Exception in thread "main" java.lang.ArithmeticException: /by zero
at DivideByZeroNoExceptionHandling.quociente(DivideByZeroNoExceptionHandling.
    java:8)
at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.
    java:20)
```

- A exceção foi detectada na linha 20 do método **main**
- Cada linha contém o nome da classe e o método seguido pelo nome do arquivo e da linha
- Subindo a pilha, vê-se que a exceção ocorre na linha 8, no método **quociente**

Ex: Divisão por zero *SEM*

tratamento de exceções

- Execução 2: Usuário insere o valor 0 como denominador...

Entre com um numerador inteiro: 100

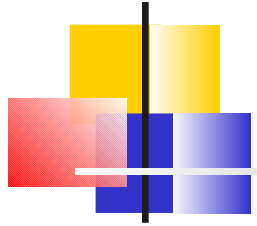
Entre com um denominador inteiro: 0

```
Exception in thread "main" java.lang.ArithmeticException: /by zero
at DivideByZeroNoExceptionHandling.quociente(DivideByZeroNoExceptionHandling.
    java:8)
at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.
    java:20)
```

A linha superior da cadeia de chamadas indica o ponto de lançamento – ponto inicial onde a exceção ocorre

Está na linha 8 do método **quociente**

Ex: Divisão por zero *SEM*



tratamento de exceções

- Execução 3: Usuário insere a string “Hello” como denominador...

```
Entre com um numerador inteiro: 100
Entre com um denominador inteiro: Hello
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Unknow Source)
at java.util.Scanner.next(Unknow Source)
at java.util.Scanner.nextInt(Unknow Source)
at DivideByZeroNoExceptionHandler.main(DivideByZeroNoExceptionHandler.
java:18)
```

Ex: Divisão por zero *SEM*

tratamento de exceções

- Execução 3: Usuário insere a string “Hello” como denominador...

Entre com um numerador inteiro: 100

Entre com um denominador inteiro: Hello

```
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Unknow Source)
at java.util.Scanner.next(Unknow Source)
at java.util.Scanner.nextInt(Unknow Source)
at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling. java:18)
```

Informa a ocorrência de uma **`InputMismatchException`** (pacote **`java.util`**)

A exceção foi detectada na linha 18 do método **`main`**.

Ex: Divisão por zero *SEM*

tratamento de exceções

- Execução 3: Usuário insere a string “Hello” como denominador...

Entre com um numerador inteiro: 100

Entre com um denominador inteiro: 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 DivideByZeroNoExceptionHandler.main(DivideByZeroNoExceptionHandler.java:18)
```

Subindo a pilha, nota-se que a exceção ocorre no método **nextInt** (pacote **java.util**)

No lugar do nome do arquivo e linha, aparece o texto **Unknown Source**

A JVM não tem acesso ao código-fonte no local da exceção

Ex: Divisão por zero *SEM*

tratamento de exceções

- Nas execuções 2 e 3, quando as exceções ocorrem e os rastreamentos são exibidos, o programa também se fecha.
- Com tratamento de exceções: o programa pode continuar mesmo que uma exceção tenha ocorrido!



Incluindo código em um bloco *try*

- O bloco *try* inclui:
 - o código que pode lançar (*throw*) uma exceção
 - e o código que não deve ser executado se ocorrer uma exceção (ie, que deve ser pulado se uma exceção for lançada)
- Um bloco *try* consiste na palavra-chave *try* seguida por uma sequência de código entre chaves {}
- Este código, ou os métodos nele invocados, podem criar objetos derivados do tipo (classe) *Exception* sinalizando condições de exceção;



Exemplo – Bloco *try*

```
try // lê dois números e calcula o quociente
{
    System.out.print( "Entre com um numerador inteiro: " );
    int numerador = scanner.nextInt();
    System.out.print( " Entre com um denominador inteiro: " );
    int denominator = scanner.nextInt();

    int result = quocient( numerador, denominador );
    System.out.printf( "\nResult: %d / %d = %d\n", numerador,
                      denominator, result );
} // fim de try
```

O método **nextInt** lança uma exceção **InputMismatchException** se o valor lido não for um inteiro válido



Capturando exceções

- Um bloco *catch* (também chamado de *handler* de exceção) captura (ie, recebe) e trata uma exceção
- Um bloco *catch* inicia-se com a palavra-chave *catch* e é seguido por um parâmetro entre parênteses e um bloco de código entre chaves { }
- Pelo menos um bloco *catch* ou um bloco *finally* (discutido depois) deve se seguir imediatamente a um bloco *try*



Exemplo – Bloco *catch*

```
catch ( InputMismatchException inputMismatchException )
{
    System.err.printf( "\nException: %s\n", inputMismatchException );

    scanner.nextLine(); // descarta entrada para o usuário tentar novamente
    System.out.println("Deve-se entrar com numeros inteiros. Tente de novo.\n" );
} // fim de catch
catch ( ArithmeticException arithmeticException )
{
    System.err.printf( "\nException: %s\n", arithmeticException );
    System.out.println("Zero é um denominador inválido. Tente de novo.\n" );
} // fim de catch
```

O primeiro bloco trata uma exceção **InputMismatchException**

O segundo bloco trata uma exceção **ArithmeticException**



Capturando exceções

```
try{  
    código que pode gerar exceções  
}  
  
catch (TipoDeExceção ref) {  
    código de tratamento da exceção  
}
```

- Todo bloco *catch* especifica entre parênteses um parâmetro de exceção que identifica o tipo (classe) de exceção que o *handler* pode processar



Capturando exceções

- Quando ocorrer uma exceção em um bloco *try*, o bloco *catch* que será executado é aquele cujo tipo de parâmetro corresponde à exceção que ocorreu
- O nome do parâmetro de exceção permite ao bloco *catch* interagir com um objeto de exceção capturado
 - Ex: invocar o método **toString** da exceção capturada, que exibe informações básicas sobre a exceção



Erros comuns

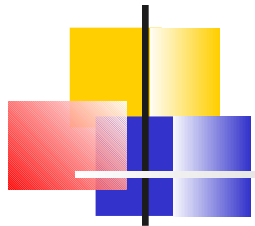
- É um erro de sintaxe colocar código entre um bloco *try* e seus blocos *catch* correspondentes
- Cada bloco *catch* pode ter apenas um parâmetro
 - Especificar uma lista de parâmetros de exceção separados por vírgula é um erro de sintaxe
- É um erro de compilação capturar o mesmo tipo de exceção em dois blocos *catch* diferentes em uma única cláusula *try*



Fluxo de controle

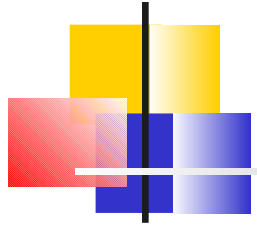
- Se ocorrer uma exceção em um bloco *try*, este termina imediatamente e o controle do programa é transferido para o primeiro dos blocos *catch* seguintes em que o tipo do parâmetro de exceção corresponda ao da exceção lançada no bloco *try*

```
try{  
    código que pode gerar exceções  
}  
  
catch (TipoDeExceção ref) {  
    código de tratamento da exceção  
}
```



Fluxo de controle

- Após a exceção ser tratada, o controle do programa não retorna ao ponto de lançamento porque o bloco *try* expirou
 - as variáveis locais do bloco também foram perdidas
- Em vez disso, o controle é retomado depois do último bloco *catch*
 - Isso é conhecido como modelo de terminação de tratamento de exceções

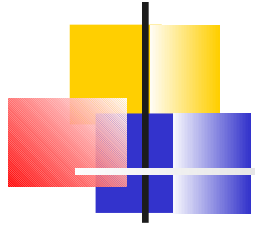


Dica de prevenção de erros

- Com o tratamento de exceções, um programa pode continuar executando (em vez de encerrar) depois de lidar com o problema
- Isso ajuda a assegurar o tipo de aplicativos robustos que colaboram para o que é chamado de computação de missão crítica

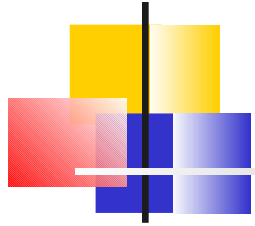


- | Ex: | <u>tipo</u> | <u>nome</u> |
|-----|------------------------|------------------------|
| | InputMismatchException | inputMismatchException |
| | ArithmeticException | arithmeticException |



Nomes de parâmetros

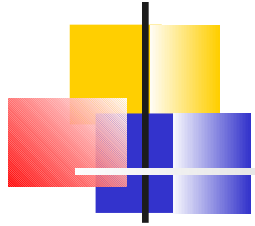
- Usar um nome de parâmetro de exceção que reflita o seu tipo promove a clareza do programa
 - lembra ao programador o tipo da exceção em tratamento



Utilizando a cláusula *throws*

- A parte da declaração de método localizada na linha 2 é conhecida como uma cláusula *throws*

```
1. public static int quociente( int numerador, int  
   denominador )  
2. throws ArithmeticException  
3. {  
4. return numerador / denominador; // possível divisão por  
   zero  
5. } // fim de método quociente
```



Utilizando a cláusula *throws*

- Uma cláusula *throws* especifica as exceções que um método lança
- Essa cláusula aparece depois da lista de parâmetros e antes do corpo do método

```
public static int quociente( int numerador, int denominador )  
    throws ArithmeticException  
{  
    return numerador / denominador; // possível divisão por  
    zero  
} // fim de método quociente
```



Utilizando a cláusula *throws*

- A cláusula *throws* contém uma lista de exceções separadas por vírgulas que o método lançará se ocorrer algum problema
- Essas exceções podem ser lançadas por instruções no corpo do método ou por métodos chamados no corpo

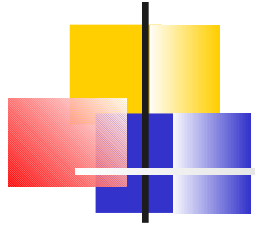
```
public static int quociente( int numerador, int denominador )  
    throws ArithmeticException  
{  
    return numerador / denominador; // possível divisão por zero  
} // fim de método quociente
```




Utilizando a cláusula *throws*

- Um método pode lançar exceções das classes listadas em sua cláusula *throws* ou de suas subclasses
- Ex: adicionamos a cláusula *throws* a esse aplicativo para indicar ao resto do programa que o método **quociente** pode lançar uma **ArithmeticException**

```
public static int quociente( int numerador, int denominador )  
    throws ArithmeticException  
{  
    return numerador / denominador; // possível divisão por zero  
} // fim de método quociente
```



Utilizando a cláusula *throws*

- Os clientes do método **quociente** são informados de que o método pode lançar uma **ArithmeticException** e de que a exceção deve ser capturada

```
public static int quociente( int numerador, int denominador )
    throws ArithmeticException
{
    return numerador / denominador; // possível divisão por
    zero
} // fim de método quociente
```



Ex:Tratando `ArithmeticException` e `InputMismatchException`

- Utilizando o tratamento de exceções para processar quaisquer `ArithmeticException` e `InputMismatchException` que possam surgir no programa
- Se o usuário cometer um erro, o programa captura e trata (lida com) a exceção
 - Permite ao usuário tentar inserir a entrada novamente



Ex:Tratando ArithmeticException e InputMismatchException

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

public class DivideByZeroWithExceptionHandling
{
    // demonstra o lançamento de uma exceção quando ocorre uma divisão
    // por zero
    public static int quociente( int numerador, int denominador )
        throws ArithmeticException
    {
        return numerador / denominador; // possível divisão por zero
    } // fim de método quociente
//continua...
```



Ex:Tratando ArithmeticException e InputMismatchException

```
public static void main( String args[] )
{
    Scanner scanner = new Scanner( System.in ); // scanner para entrada
    boolean continueLoop = true; // determina se mais tentativas são
    necessárias
    do
    {
        try { // lê dois números e calcula o quociente
            System.out.print( "Entre com um numerador inteiro: " );
            int numerador = scanner.nextInt();
            System.out.print( " Entre com um denominador inteiro: " );
            int denominador = scanner.nextInt();
            int result = quociente( numerador, denominador );
            System.out.printf( "\nResult: %d / %d = %d\n", numerador,
                               denominador, result );
            continueLoop = false; // entrada bem-sucedida; fim de loop
        } // fim de try
    } // continua...
```



Ex:Tratando ArithmeticException e InputMismatchException

```
catch ( InputMismatchException inputMismatchException )
{
    System.err.printf( "\nException: %s\n", inputMismatchException );

    scanner.nextLine(); // descarta entrada para o usuário tentar
    novamente
    System.out.println("Deve-se entrar com numeros inteiros. Tente de
    novo.\n");
} // fim de catch
catch ( ArithmeticException arithmeticException )
{
    System.err.printf( "\nException: %s\n", arithmeticException );
    System.out.println("Zero e um denominador invalido. Tente de novo.\n"
);
} // fim de catch
} while ( continueLoop ); // fim de do...while
} // fim de main
} // fim da classe DivideByZeroWithExceptionHandling
```



Ex:Tratando `ArithmeticException` e `InputMismatchException`

- Execução 2: Usuário insere o valor 0 como denominador...

Entre com um numerador inteiro: 100

Entre com um denominador inteiro: 0

Exception: `java.lang.ArithmeticException: /by zero`
Zero e um denominador invalido. Tente de novo.

Entre com um numerador inteiro: 100

Entre com um denominador inteiro: 7

Result: $100/7 = 14$



Ex:Tratando ArithmeticException e InputMismatchException

- Execução 3: Usuário insere a string “Hello” como denominador...

```
Entre com um numerador inteiro: 100  
Entre com um denominador inteiro: Hello
```

```
Exception: java.util.InputMismatchException  
Deve-se entrar com numeros inteiros. Tente de novo.
```

```
Entre com um numerador inteiro: 100  
Entre com um denominador inteiro: 7
```

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




Usos típicos

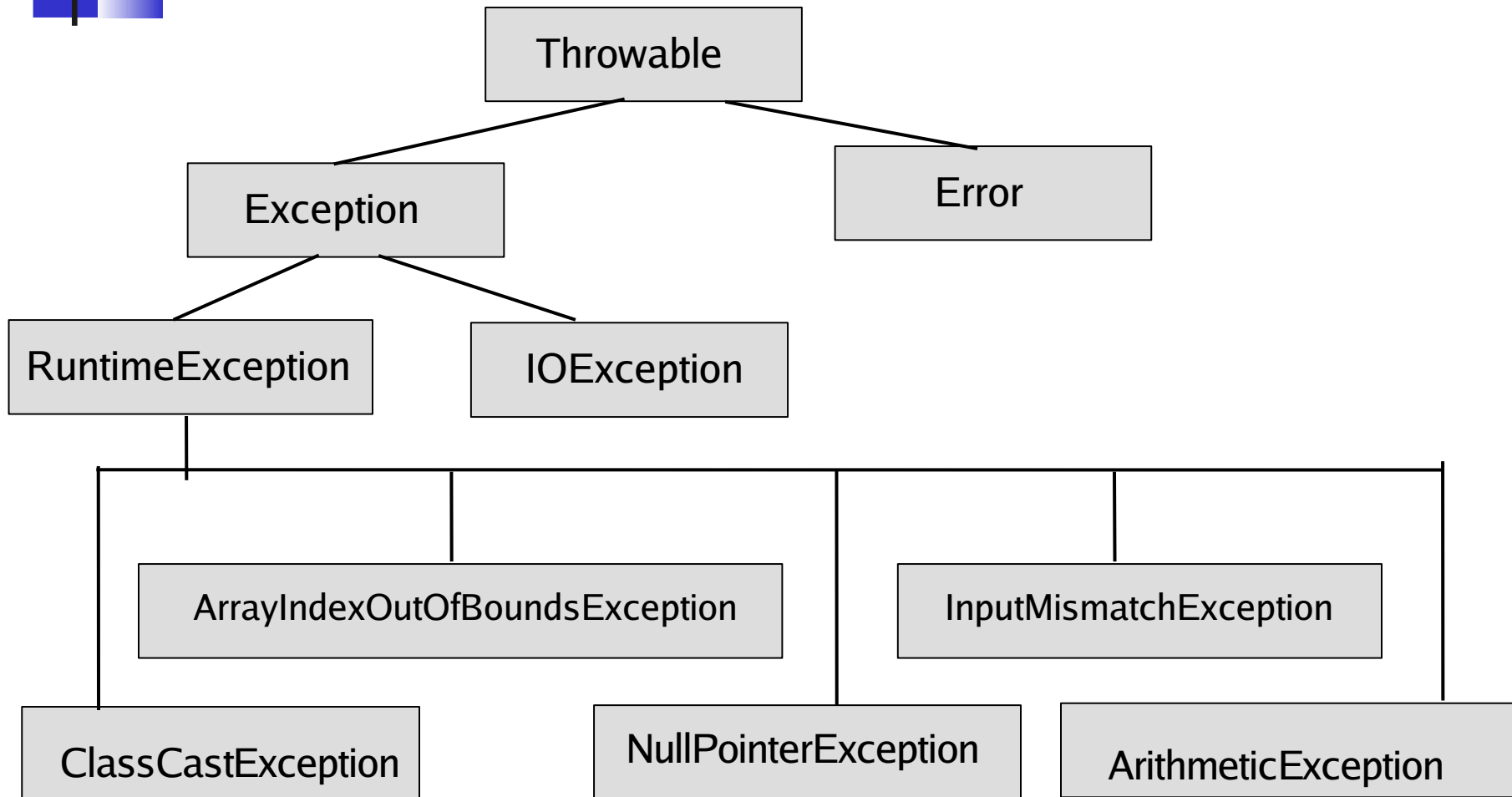
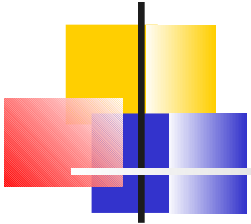
- Quando o sistema pode se recuperar do erro: o tratador de erro implementa o procedimento de recuperação
- Quando o sistema não pode se recuperar do erro mas desejamos encerrar o programa de forma “limpa”
- Em projetos grandes que exijam tratamento de erros uniforme

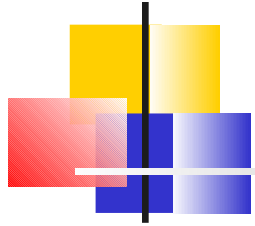


Hierarquia de exceções em Java

- Todas as classes de exceção de Java herdam, direta ou indiretamente, da classe **Exception**, formando uma hierarquia de herança
- Os programadores podem estender essa hierarquia para criar suas próprias classes de exceção

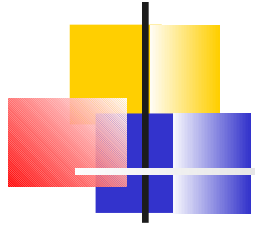
Parte da hierarquia de herança da classe Throwable





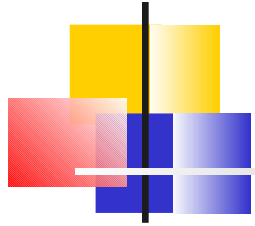
Hierarquia de exceções em Java

- Somente objetos **Throwable** podem ser usados com o mecanismo de tratamento de exceções
- A classe **Throwable** tem duas subclasses: **Exception** e **Error**



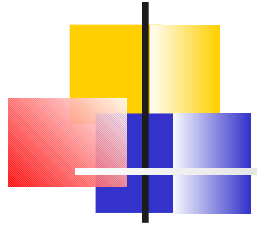
Hierarquia de exceções em Java

- A classe **Exception** e suas subclasses representam situações excepcionais que podem ocorrer em um programa e que podem ser capturadas por um aplicativo
- Ex:
 - subclasse **RuntimeException** (pacote **java.lang**)
 - subclasse **IOException** (pacote **java.io**)



Hierarquia de exceções em Java

- A classe **Error** e suas subclasses (ex, **OutOfMemoryError**) representam situações anormais que podem acontecer na JVM
- Exceções **Error** acontecem raramente e não devem ser capturadas por aplicativos
 - normalmente não é possível que os aplicativos se recuperem de exceções **Error**

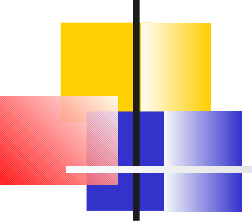


Hierarquia de exceções em Java

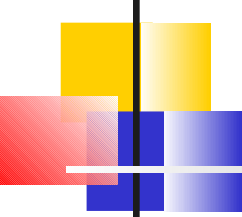
- A hierarquia de exceções Java é enorme, contendo centenas de classes
- A documentação sobre a classe **Throwable** pode ser encontrada em:

java.sun.com/javase/6/docs/api/java/lang/Throwable.html

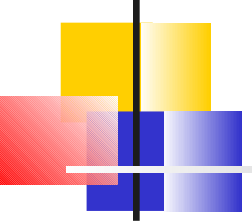
Exceções verificadas e não verificadas

- 
- Java faz distinção entre duas categorias de exceção:
 - verificadas
 - não verificadas
 - O compilador Java impõe um requisito *catch-or-declare* (capture ou declare) às exceções verificadas

Exceções verificadas e não verificadas

- 
- O tipo de uma exceção determina se ela é verificada ou não verificadas
 - Todos os tipos de exceção que são subclasses da classe **RuntimeException** são exceções não verificadas
 - Ex:
 - subclasse **ArrayIndexOutOfBoundsException**
 - subclasse **ArithmeticException**

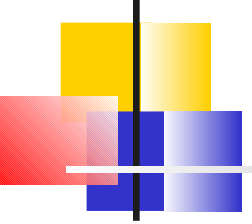
Exceções verificadas e não verificadas

- 
- Todas as classes que herdam da classe **Exception** mas não da classe **RuntimeException** são exceções verificadas
 - Ex:
 - subclasse **IOException**
 - As classes que herdam da classe **Error** são consideradas não verificadas

Exceções verificadas e não verificadas

- O compilador verifica cada chamada de método e declaração de método para determinar se o método lança exceções verificadas
- Se lançar, o compilador assegura que a exceção verificada é capturada (via blocos **try/catch**) ou declarada em uma cláusula **throws**

Exceções verificadas e não verificadas

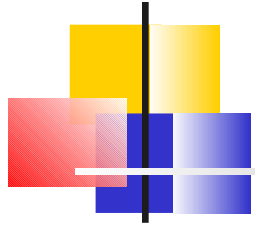
- 
- Ou seja, para satisfazer o requisito *catch-or-declare*, uma das duas atitudes deve ser tomada:
 - 1) *catch* (capture): incluir o código que gera a exceção em um bloco **try** e fornecer um handler **catch** para tratar o tipo da exceção verificada
 - 2) *declare* (declare): adicionar, depois da lista de parâmetros e antes do corpo do método que lança a exceção, uma cláusula **throws** contendo o tipo da exceção verificada

Exceções verificadas e não verificadas

- Se o requisito *catch-or-declare* não for satisfeito, o compilador emitirá uma mensagem de erro indicando que a exceção deve ser capturada ou declarada
- Isso força os programadores a pensarem nos problemas que podem ocorrer quando um método que lança exceções verificadas for chamado

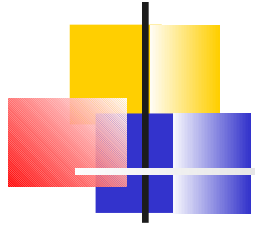
Exceções verificadas e não verificadas

- Ao contrário das exceções verificadas, o compilador Java não verifica o código para determinar se uma exceção não verificada é capturada ou declarada
- Não é necessário que as exceções não verificadas sejam listadas na cláusula **throws** de um método
 - mesmo se forem, essas exceções não precisam ser capturadas por um aplicativo



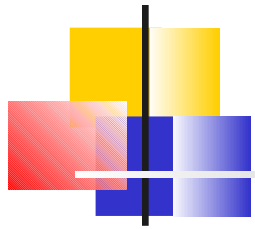
Erro comum de programação

- Colocar um bloco **catch** para um tipo de exceção de superclasse antes de outros blocos **catch** que capturam tipos de exceção de subclasse impede que esses blocos executem
- Então, ocorre um erro de compilação



Bloco *finally*

- Os programas que obtém certos tipos de recurso devem retorná-los ao sistema explicitamente para evitar os supostos vazamentos de recurso
- Exemplos de recursos:
 - arquivos
 - conexões com bancos de dados
 - conexões de rede



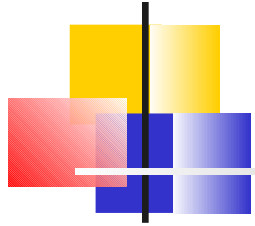
Bloco **finally**

- O bloco **finally** é opcional e consiste na palavra-chave **finally** seguida pelo código entre chaves **{}**
- Se estiver presente, esse bloco é colocado depois do último bloco **catch**



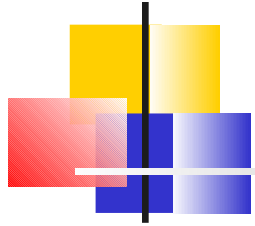
Bloco finally

```
try {  
    instruções  
    instruções de aquisição de recursos  
}//fim de try  
catch(UmTipoDeExceção exception1)  
{  
    instruções de tratamento de exceção  
}//fim de catch  
...  
catch(OutroTipoDeExceção exception2)  
{  
    instruções de tratamento de exceção  
}//fim de catch  
finally {  
    instruções  
    instruções de liberação de recursos  
}//fim de finally
```



Bloco **finally**

- O bloco **finally** quase sempre será executado, independentemente de ter ocorrido uma exceção ou de esta ter sido tratada ou não
- O bloco **finally** não será executado somente se o aplicativo fechar antes de um bloco **try** chamando o método **System.exit**
 - Esse método fecha imediatamente um aplicativo



Bloco **finally**

- Como bloco **finally** quase sempre será executado, ele em geral contém o código de liberação de recursos
- O bloco **finally** não será executado somente se o aplicativo fechar antes de um bloco **try** chamando o método **System.exit**
 - Esse método fecha imediatamente um aplicativo



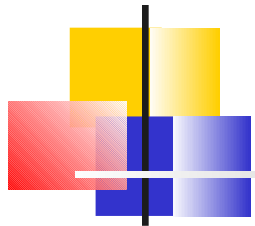
Lançando exceções com **throw**

- Os programadores podem lançar exceções utilizando a instrução **throw**
- A instrução **throw** é executada para sinalizar a ocorrência de uma exceção
- Assim como as exceções lançadas pelos métodos da API Java, isso indica para os aplicativos clientes que ocorreu um erro
- O operando de **throw** pode ser de qualquer classe derivada de **Throwable**



Lançando exceções com **throw**

```
public static void throwException() throws Exception
{
    try // lança uma exceção e imediatamente a captura
    {
        System.out.println( "Metodo throwException" );
        throw new Exception(); // gera a exceção
    } // fim de try
    catch ( Exception exception ) // captura exceção lançada em try
    {
        System.err.println("Excecao tratada no metodo throwException");
    } // fim de catch
    finally // executa independentemente do que ocorre em try...catch
    {
        System.err.println( "Finally executado em throwException" );
    } // fim de finally
} // fim de método throwException
```



Relançando exceções

- As exceções são relançadas quando um bloco **catch**, ao receber uma exceção, decide que não pode processar essa exceção ou que só pode processá-la parcialmente
- Relançar uma exceção adia o tratamento de exceções (ou parte dele) para um outro bloco **catch** associado com uma instrução **try** externa
- Uma exceção é relançada utilizando a palavra-chave **throw** seguida por uma referência ao objeto que acabou de ser capturado



Relançando exceções

```
public static void throwException() throws Exception
{
    try { // lança uma exceção e imediatamente a captura
        System.out.println( "Metodo throwException" );
        throw new Exception(); // gera a exceção
    } // fim de try
    catch ( Exception exception ) // captura exceção lançada em try
    {
        System.err.println("Excecao tratada no metodo throwException" );
        throw exception; // lança novamente para processamento adicional
    } // fim de catch
    finally // executa independentemente do que ocorre em try...catch
    {
        System.err.println( "Finally executado in throwException" );
    } // fim de finally
} // fim de método throwException
```




Relançando exceções

```
public static void main( String args[] )
{
    try
    {
        ➡ throwException(); // chama método throwException
    } // fim de try
    catch ( Exception exception ) // exceção lançada por
    throwException
    {
        System.err.println( "Exception handled in main" );
    } // fim de catch

} // fim de main
```



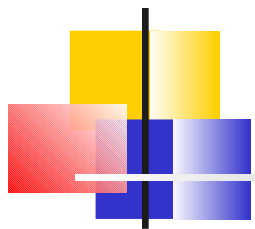
Exceções aninhadas

- A captura e tratamento de exceções pode ser aninhada em vários níveis de *try/catch*:

```
try{  
    try{  
        throw Exceção2  
    }  
    catch ( Exceção1 ){...}  
}  
catch( Exceção2 ){...}
```

Responsabilidade de tratamento de exceções

- Quando um método lança uma exceção, o ambiente Java tenta encontrar algum código capaz de tratá-la;
- Em alguns casos é conveniente que o próprio método que gerou a exceção faça seu tratamento;
- Em outros, é mais adequado propagá-la ao método que o chamou.



Pilha de execução

- O código para tratamento da exceção pode estar no próprio método que a provocou, ou em algum método superior na *pilha de execução*.
- A pilha de execução é a lista ordenada de métodos que foram chamados até chegar ao método que gerou a exceção



Onde está o *catch* ?

- O ambiente Java pesquisa a pilha de execução em busca de um tratamento para a exceção que foi gerada;
- Quando um tratamento adequado (i.e., para o tipo de exceção em questão) for encontrado, este assume o controle do programa;
- Neste caso diz-se que o tratador de exceção “captura” (*catch*) o evento;
- Caso nenhum tratador seja encontrado, o controle chega de volta até *main()* e o programa termina.



Exemplo

```
// throwException lança exceção que não é capturada nesse método
public static void throwException() throws Exception
{
    try { // lança uma exceção e a captura em main
        System.out.println( "Metodo throwException" );
        throw new Exception(); // gera a exceção
    } // fim de try
    catch ( RuntimeException runtimeException ) { // captura tipo incorreto
        System.err.println("Excecao tratada no metodo throwException" );
    } // fim de catch
    finally { // o bloco finally sempre executa
        System.err.println( "Finally e sempre executado" );
    } // fim de finally
} // fim de método throwException
```



Exemplo

```
public static void main( String args[] )
{
    try { // chama throwException para demonstrar o
        desempilhamento
        throwException();
    } // fim de try
    catch ( Exception exception ) // exceção lançada em
        throwException
    {
        System.err.println( "Exceção tratada em main" );
    } // fim de catch
} // fim de main
```



Rastreamento de pilha

- As informações de rastreamento de pilha incluem:
 - O nome da exceção (ex, **java.lang.ArithmeticException**) em uma mensagem descritiva que indica o problema que ocorreu
 - O caminho de execução (pilha de chamadas de métodos) que resultou na exceção, método por método



printStackTrace, getStackTrace e getMessage

- A classe **Throwable** oferece um método chamado **printStackTrace** que envia para o fluxo de erro padrão o rastreamento da pilha
 - Útil para o processo de teste e depuração



printStackTrace, getStackTrace e getMessage

- A classe **Throwable** também fornece o método **getStackTrace** que recupera informações sobre o rastreamento da pilha que podem ser impressas por **printStackTrace**
- O método **getMessage** da classe **Throwable** retorna a string descritiva armazenada em uma exceção

```
public class UsingExceptions
{
    public static void main( String args[] )
    {
        try
        {
            method1(); // chama method1
        } // fim de try
        catch ( Exception exception ) // captura a exceção lançada em method1
        {
            System.err.printf( "%s\n\n", exception.getMessage());
            exception.printStackTrace(); // imprime rastreamento de pilha da exceção

            // obtém informações de rastreamento de pilha
            StackTraceElement[] traceElements = exception.getStackTrace();

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

            // faz um loop por traceElements para obter a descrição da exceção
            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());
            } // for final
        } // fim de catch
    } // fim de main
}
```

```

public class UsingExceptions
{
    public static void main( String args[] )
    {
        try
        {
            method1(); // chama method1
        } // fim de try
        catch ( Exception exception ) // captura a
        {
            System.err.printf( "%s\n\n", exception.getMessage());
            exception.printStackTrace(); // imprime rastreamento de pilha da exceção

            // obtém informações de rastreamento de pilha
            StackTraceElement[] traceElements=exception.getStackTrace();

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

            // faz um loop por traceElements para obter a descrição da exceção
            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());
            } // for final
        } // fim de catch
    } // fim de main
}

```

**Invoca o método
getMessage da exceção
para obter a sua descrição**

```
public class UsingExceptions
```

```
{
```

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

```
    {
```

```
        try
```

```
        {
```

```
            method1(); // chama method1
```

```
        } // fim de try
```

```
        catch ( Exception exception ) //
```

```
        {
```

```
            System.err.printf( "%s\n\n", exception.getMessage());
```

```
            exception.printStackTrace(); // imprime rastreamento de pilha da exceção
```

```
            // obtém informações de rastreamento de pilha
```

```
            StackTraceElement[] traceElements=exception.getStackTrace();
```

```
            System.out.println( "\nStack trace from getStackTrace:" );
```

```
            System.out.println( "Class\t\tFile\t\t\tLine\tMethod" );
```

```
            // faz um loop por traceElements para obter a descrição da exceção
```

```
            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());
```

```
            } // for final
```

```
        } // fim de catch
```

```
    } // fim de main
```

**Invoca o método
printStackTrace para gerar a saída
do rastreamento de pilha
que indica onde ocorreu a exceção**

```

public class UsingExceptions
{
    public static void main( String args[] )
    {
        try
        {
            method1(); // chama method1
        } // fim de try
        catch ( Exception exception
        {
            System.err.printf( "%s\n",
            exception.printStackTrace() );
        }
    }
}

```

As informações de rastreamento são colocadas em um array de objetos `StackTraceElement`

```

// obtém informações de rastreamento de pilha
StackTraceElement[] traceElements=exception.getStackTrace();

```

```

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

```

```

// faz um loop por traceElements para obter a descrição da exceção
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());
} // for final

```

```

} // fim de catch

```

```

} // fim de main

```

```
public class UsingExceptions
{
    public static void main( String args[] )
    {
        try
        {
            method1(); // chama method1
        } // fim de try
        catch ( Exception exception ) // captura a exceção lançada em method1
        {
            System.err.printf( "%s\n\n", exception.getMessage());
            exception.printStackTrace(); // imprime rastreamento de pilha da exceção

            // obtém informações de rastreamento de
            StackTraceElement[] traceElements=exception.getStackTrace();

            System.out.println( "\nStack trace from " + exception );
            System.out.println( "Class\t\tFile\t\t\tLine Number" );

            // faz um loop por traceElements para obter a descrição da exceção
            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());
            } // for final
        } // fim de catch
    } // fim de main
}
```

Instrução for aprimorada: itera pelos elementos de um array sem utilizar um contador

```

public class UsingExceptions
{
    public static void main( String args[] )
    {
        try
        {
            method1(); // chama method1
        } // fim de try
        catch ( Exception exception ) // captura a exceção lançada em method1
        {
            System.err.printf( "%s\n\n", exception.getMessage());
            exception.printStackTrace(); // imprime rastreamento de pilha da exceção

            // obtém informações de rastreamento
            StackTraceElement[] traceElements = exception.getStackTrace();

            System.out.println( "\nStack Trace:" );
            System.out.println( "Class\tFile\tLine\tMethod" );

            // faz um loop por traceElements para obter a descrição da exceção
            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());
            } // for final
        } // fim de catch
    } // fim de main
}

```

Sintaxe da instrução for aprimorada:
(tipo identificador : nome do array)
- O tipo do parâmetro deve corresponder ao tipo dos elementos do array


```

public class UsingExceptions
{
    public static void main( String args[] )
    {
        try
        {
            method1(); // chama method1
        } // fim de try
        catch ( Exception exception ) // captura a exceção lançada em method1
        {
            System.err.printf( "%s\n\n", exception.getMessage());
            exception.printStackTrace(); // imprime rastreamento de pilha da exceção

            // obtém informações de rastreamento de pilha
            StackTraceElement[] elements = exception.getStackTrace();

            System.out.printf( "%s\n", elements.length );
            System.out.println( "-----" );

            // faz o loop para cada elemento da pilha
            for ( StackTraceElement element : elements )
            {
                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());
            } // for final
        } // fim de catch
    } // fim de main
}

```

**Obtém cada elemento do array e invoca
seus métodos para recuperar o nome da classe,
o nome do arquivo, o número da linha e o nome do método
para esses elementos**

```
public class UsingExceptions
{
    public static void main( String args[] )
    {
        try
        {
            method1(); // chama method1
        } // fim de try
        catch ( Exception exception ) // captura a exceção lançada em method1
        {
            System.err.printf( "%s\n\n", exception.getMessage());
            exception.printStackTrace(); // imprime rastreamento de pilha da exceção

            // obtém informações de rastreamento de pilha
            StackTraceElement[] traceElements = exception.getStackTrace();

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

            // faz um loop por traceElements para obter a descrição da exceção
            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());
            } // for final
        } // fim de catch
    } // fim de main
}
```

```
1. public static void method1()throws Exception
2. {
3.     method2();
4. } // fim de método method1
```

```
6. public static void method2()throws Exception
7. {
8.     method3();
9. } // fim de método method2
```

```
11. public static void method3()throws Exception
12. {
13.     throw new Exception( "Exception thrown in method3" );
14. } // fim de método method3
15.} // fim da classe UsingExceptions
```

ponto de
lançamento da
exceção

■ Saída do programa:

Exception thrown in method3

java.lang.Exception:Exception thrown in method3

at UsingExceptions.method3(UsingExceptions.java:40)

at UsingExceptions.method2(UsingExceptions.java:36)

at UsingExceptions.method1(UsingExceptions.java:32)

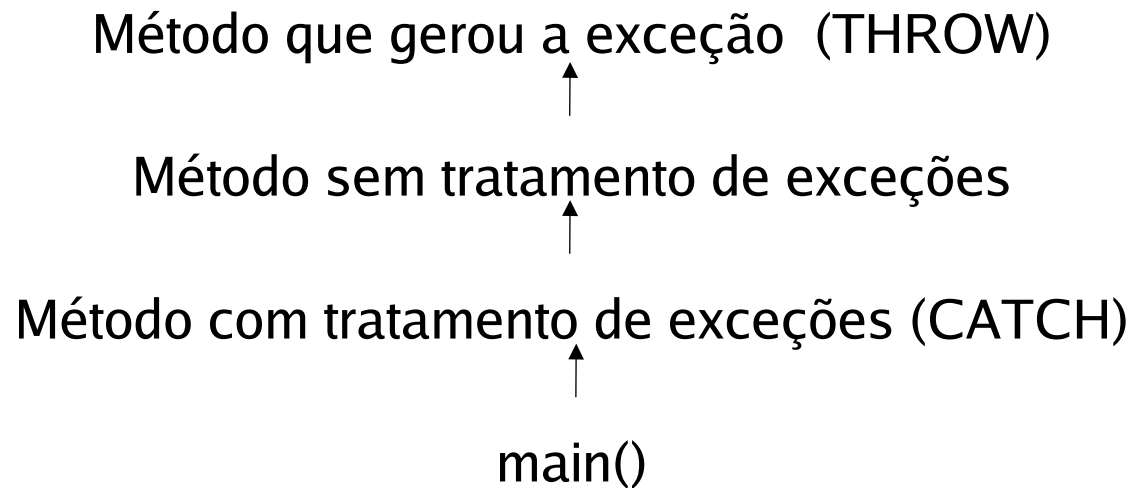
at UsingExceptions.main(UsingExceptions.java:7)

Stack trace from getStackTrace:

Class	File	Line	Method
UsingExceptions	UsingExceptions.java	40	method3
UsingExceptions	UsingExceptions.java	36	method2
UsingExceptions	UsingExceptions.java	32	method1
UsingExceptions	UsingExceptions.java	7	main



Throw/Catch e a pilha de execução





Declarando novos tipos de exceção

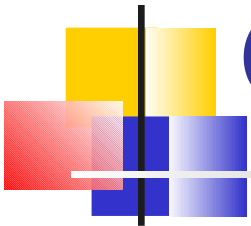
- Programadores podem achar útil declarar suas próprias classes de exceção
 - específicas aos problemas que podem ocorrer quando outro programador empregar suas classes reutilizáveis
- Uma nova classe de exceção deve estender uma classe de exceção existente
 - assegura que a classe pode ser utilizada com o mecanismo de tratamento de exceções

Declarando novos tipos de exceção

- Exceções são derivadas da classe *Exception*
- O construtor da exceção armazena no objeto criado informações sobre o evento (e.g., a mensagem de erro a ser exibida etc)
- Em geral uma classe de exceção possuirá dois construtores:
 - Um construtor default (i.e., sem argumentos) criando uma mensagem de erro padrão
 - Um construtor que recebe uma mensagem de exceção personalizada

Declarando novos tipos de exceção

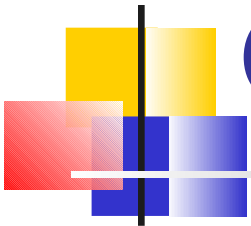
- A string da mensagem é armazenada em uma variável do objeto **exceção** criado;
- Essa string pode ser recuperada pelo método **getMessage** da classe **Exception**;
- O próprio nome da exceção pode ser obtido com **toString(exceção)**



Gerando e tratando exceções

- O código em **try** (ou nos métodos por ele invocados) pode conter comandos **throw** para lançar uma nova exceção

```
try{  
    if (condição) throw new MinhaExceção();  
}  
catch (MinhaExceção x) {  
    System.out.println(x.getMessage());  
}
```



Gerando e tratando exceções

- No tratamento da exceção (bloco **catch**) a mensagem criada pelo construtor da exceção pode ser obtida pelo método **getMessage** da classe **Exception**

```
try{
    if (condição) throw new MinhaExceção();
}
catch (MinhaExceção x) {
    System.out.println(x.getMessage());
}
```



Exemplo: divisão por zero

- Problema: o usuário entra com dois inteiros para divisão, e desejamos capturar erros de divisão por zero;
- Em **java.lang** não há uma exceção específica para divisão por zero
 - o mais próxima é a **ArithmeticException**
- Então estendemos e criamos nossa própria subclasse de exceção, que será chamada **ExceçãoDivisãoPorZero**



Exemplo: divisão por zero

```
// ExcecaoDivisaoPorZero.java
public class ExceçãoDivisãoPorZero
    extends Exception {
    public ExceçãoDivisãoPorZero() {
        super("Tentativa de divisão por zero");
    }

    public ExceçãoDivisãoPorZero(String msg) {
        super(msg);
    }
}
```



Exemplo: divisão por zero

```
69      try {
73          result = divisão( n1, n2 );
74          System.out.println(result);
75      }
82      catch (ExceçãoDivisãoPorZero e ) {
83          System.out.println(e.toString().e.getMessage());
86      }
87  }
```

ExceçãoDivisãoPorZero:
Tentativa de divisão
por zero

- Se for gerada alguma exceção dentro do bloco *try*, o bloco inteiro é encerrado, e a execução é desviada para a cláusula *catch* correspondente;
- Não ocorrendo uma exceção, o código em *catch* é ignorado.



Exemplo: divisão por zero

- Uma exceção é lançada pelo comando **throw**

```
// em algum lugar dentro do método divisão...
94  {
95      if ( denominador == 0 )
96          throw new ExceçãoDivisãoPorZero();
97
98      return  numerador / denominador;
99  }
100
```

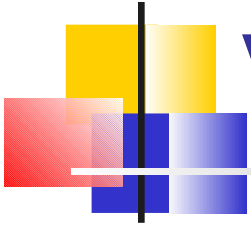
Quando criar uma classe do tipo exceção ?

- Quando for preciso usar um tipo de exceção não definido na plataforma Java
- Quando for útil a distinção entre suas exceções e as geradas por outros programadores
- Quando o código gera várias exceções relacionadas



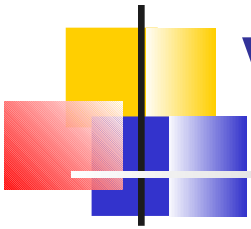
Lembre-se

- O tratamento de exceções pode fazer muito mais do que simplesmente exibir uma mensagem de erro:
 - Recuperação de erros;
 - Solicitar ao usuário orientação sobre como proceder;
 - Propagar o erro até um gerenciador de exceções de alto nível.



Vantagens em tratar exceções

1. Separação entre o código principal (e.g., da seqüência típica de eventos) do código de tratamento de erros;
2. Propagação de erros ao topo da pilha de execução;
 - Erros só precisam ser tratados por métodos que estão interessados neles;
 - Aumento da coesão de classes.



Vantagens em tratar exceções

3. Agrupamento e diferenciação de tipos de exceções em uma hierarquia:
 - O tratamento de exceções pode ser tão genérico ou tão específico quanto desejado;
 - Em geral procura-se definir exceções tão específicas quanto possível.

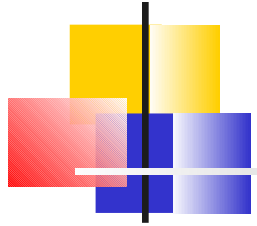


Resumo

- Foi apresentado o mecanismo de tratamento de exceções da linguagem Java e as estruturas e comandos associados:
 - `try ... catch; finally; throw; throws; exception; error; etc.`

Referências

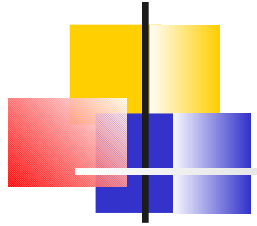
- H.M. Deitel & P. J. Deitel, Java – como programar, 6a. Edição, Pearson Prentice-Hall, São Paulo, 2005. Capítulo 13.



Exercício 1

Este código está correto ?

```
try      {  
        }  
finally {  
        }
```

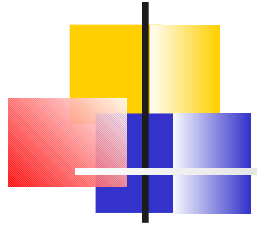


Exercício 2

Que tipos de exceções podem ser capturadas pelo código abaixo ?

Há alguma contra-indicação no seu uso ?

```
try {  
}  
catch (Exception e)      {  
}
```



Exercício 3

Há algo errado com o código abaixo ?

Ele vai compilar ?

```
try {  
} catch (Exception e) {  
} catch (ArithmeticException a) {  
}
```

Exercício 4: Qual a saída deste programa ?

```
class Exemplo
{public static void main (String[] a)
{ try    { teste(); }
  catch ( Exceção x)
        { System.out.println("Tratamento 3"); }
}

static void teste() throws Exceção
{ try    {
        try {throw new Exceção(); }
        catch ( Exceção x)
            { System.out.println("Tratamento 1"); }
            throw new Exceção(); } }
  catch (Exceção x)
    { System.out.println("Tratamento 2");
      throw new Exceção(); }
}
}

class Exceção extends Exception {}
```


Referências

- H.M. Deitel & P. J. Deitel, Java – como programar, 6a. Edição, Pearson Prentice-Hall, São Paulo, 2005. Capítulo 13.