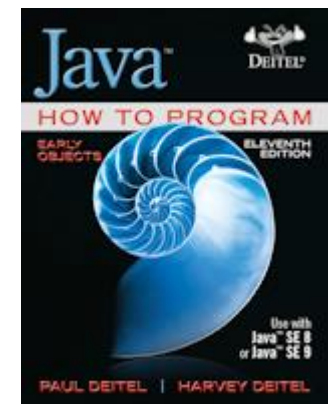


Java – Aula 8

Exceções

Notas de Aula
Prof. André Bernardi
andrebernardi@unifei.edu.br





Exceções

- Uma ***exceção*** indica um problema que ocorre quando um programa é executado.
- O nome "***exceção***" sugere que o problema ocorre com pouca frequência;
- O tratamento de exceção ajuda a criar programas tolerantes a falhas.
- O tratamento de exceção processa erros síncronos, que ocorrem quando uma instrução é executada.



Tratamento de Exceções

Utilização:

- Existem algumas falhas que decorrem da má utilização do usuário.
- Existem falhas que podem ocorrer fora do escopo atual.

O tratamento de exceção não é projetado para processar problemas associados com eventos assíncronos, que ocorrem paralelamente com o fluxo do programa de controle e independentemente dele.

```

1 // Figura 7.8: StudentPoll.java
2 // Programa de análise de enquete.
3
4 public class StudentPoll
5 {
6     public static void main(String[] args)
7     {
8         // array das respostas dos alunos (mais tipicamente, inserido em tempo de execução)
9         int[] responses = { 1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3,
10                             2, 3, 3, 2, 14 };
11         int[] frequency = new int[6]; // array de contadores de frequência
12
13         // para cada resposta, seleciona elemento de respostas e utiliza esse valor
14         // como índice de frequência para determinar elemento a incrementar
15         for (int answer = 0; answer < responses.length; answer++)
16         {
17             try
18             {
19                 ++frequency[responses[answer]];
20             }
21             catch (ArrayIndexOutOfBoundsException e)
22             {
23                 System.out.println(e); // invoca o método toString
24                 System.out.printf(" responses[%d] = %d\n\n",
25                                     answer, responses[answer]);
26             }
27         }
28     }
29 }

```

```
29         System.out.printf("%s%10s%n", "Rating", "Frequency");
30
31         // gera saída do valor de cada elemento do array
32         for (int rating = 1; rating < frequency.length; rating++)
33             System.out.printf("%6d%10d%n", rating, frequency[rating]);
34     }
35 } // fim da classe StudentPoll
```

```
java.lang.ArrayIndexOutOfBoundsException: 14
    responses[19] = 14
```

Rating	Frequency
1	3
2	4
3	8
4	2
5	2



Tratamento de Exceções

- Para lidar com uma exceção, coloque qualquer código que pode lançar uma exceção em uma instrução ***try***.
- O bloco ***try*** contém o código que pode lançar uma exceção, e o bloco ***catch*** contém o código que manipula a exceção se uma ocorrer.
- Pode haver muitos blocos ***catch*** para tratar com diferentes tipos de exceções que podem ser lançadas no bloco ***try*** correspondente.



Tratamento de Exceções

- Quando um bloco ***try*** termina, todas as variáveis declaradas no bloco ***try*** saem de escopo.
- Um bloco ***catch*** declara um tipo e um parâmetro de exceção. Dentro do bloco catch, você pode usar o identificador do parâmetro para interagir com um objeto que capturou a exceção.
- O método ***toString*** de um objeto de exceção retorna uma mensagem de erro da exceção.



Sintaxe

```
try
{
    // esta parte do código será executada, e caso haja alguma exceção ...
}
catch (ClasseDaExceção1 instância1)
{
    // esta parte do código será executada caso a exceção seja da classe especificada
}
catch (ClasseDaExceção2 instância2)
{
    // esta parte do código será executada caso a exceção seja da classe especificada
}
finally
{
    // esta parte do código será executada independente da ocorrência de exceções
}
```

É um erro de sintaxe colocar código entre um bloco try e seus blocos catch correspondentes.



Exemplos

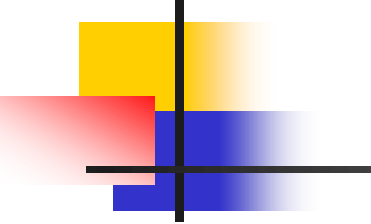
- divisão por zero sem tratamento de exceção;
- tratando *ArithmeticExceptions* e *InputMismatchExceptions* ;

O tratamento de exceção fornece uma técnica única e uniforme para documentar, detectar e recuperar-se de erros. Isso ajuda os programadores que trabalham em grandes projetos a entender o código de processamento de erro uns dos outros.

```

1 // Figura 11.2: DivideByZeroNoExceptionHandling.java
2 // Divisão de inteiro sem tratamento de exceção.
3 import java.util.Scanner;
4
5 public class DivideByZeroNoExceptionHandling
6 {
7     // demonstra o lançamento de uma exceção quando ocorre uma divisão por zero
8     public static int quotient(int numerator, int denominator)
9     {
10         return numerator / denominator; // possível divisão por zero
11     }
12
13     public static void main(String[] args)
14     {
15         Scanner scanner = new Scanner(System.in);
16
17         System.out.print("Please enter an integer numerator: ");
18         int numerator = scanner.nextInt();
19         System.out.print("Please enter an integer denominator: ");
20         int denominator = scanner.nextInt();
21
22         int result = quotient(numerator, denominator);
23         System.out.printf(
24             "%nResult: %d / %d = %d%n", numerator, denominator, result);
25     }
26 } // fim da classe DivideByZeroNoExceptionHandling

```



```
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: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at DivideByZeroNoExceptionHandling.quotient(
        DivideByZeroNoExceptionHandling.java:10)
    at DivideByZeroNoExceptionHandling.main(
        DivideByZeroNoExceptionHandling.java:22)
```

```
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)
```

```

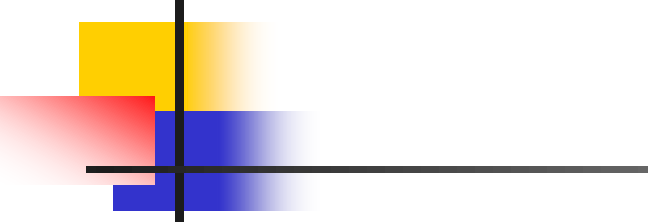
1 // Figura 11.3: DivideByZeroWithExceptionHandling.java
2 // Tratando ArithmeticExceptions e InputMismatchExceptions.
3 import java.util.InputMismatchException;
4 import java.util.Scanner;
5
6 public class DivideByZeroWithExceptionHandling
7 {
8     // demonstra o lançamento de uma exceção quando ocorre uma divisão por zero
9     public static int quotient(int numerator, int denominator)
10         throws ArithmeticException
11     {
12         return numerator / denominator; // possível divisão por zero
13     }
14
15     public static void main(String[] args)
16     {
17         Scanner scanner = new Scanner(System.in);
18         boolean continueLoop = true; // determina se mais entradas são necessárias
19
20         do
21         {
22             try // lê dois números e calcula o quociente
23             {
24                 System.out.print("Please enter an integer numerator: ");
25                 int numerator = scanner.nextInt();

```

```

26         System.out.print("Please enter an integer denominator: ");
27         int denominator = scanner.nextInt();
28
29         int result = quotient(numerator, denominator);
30         System.out.printf("%nResult: %d / %d = %d%n", numerator,
31                             denominator, result);
32         continueLoop = false; // entrada bem-sucedida; fim do loop
33     }
34     catch (InputMismatchException inputMismatchException)
35     {
36         System.err.printf("%nException: %s%n", inputMismatchException);
37         scanner.nextLine(); // descarta entrada para poder tentar de novo
38         System.out.printf(
39             "You must enter integers. Please try again.%n%n");
40     }
41
42     catch (ArithmeticException arithmeticException)
43     {
44         System.err.printf("%nException: %s%n", arithmeticException);
45         System.out.printf(
46             "Zero is an invalid denominator. Please try again.%n%n");
47     }
48 } while (continueLoop);
49 }
50 } // fim da classe DivideByZeroWithExceptionHandling

```



```
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: 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
```



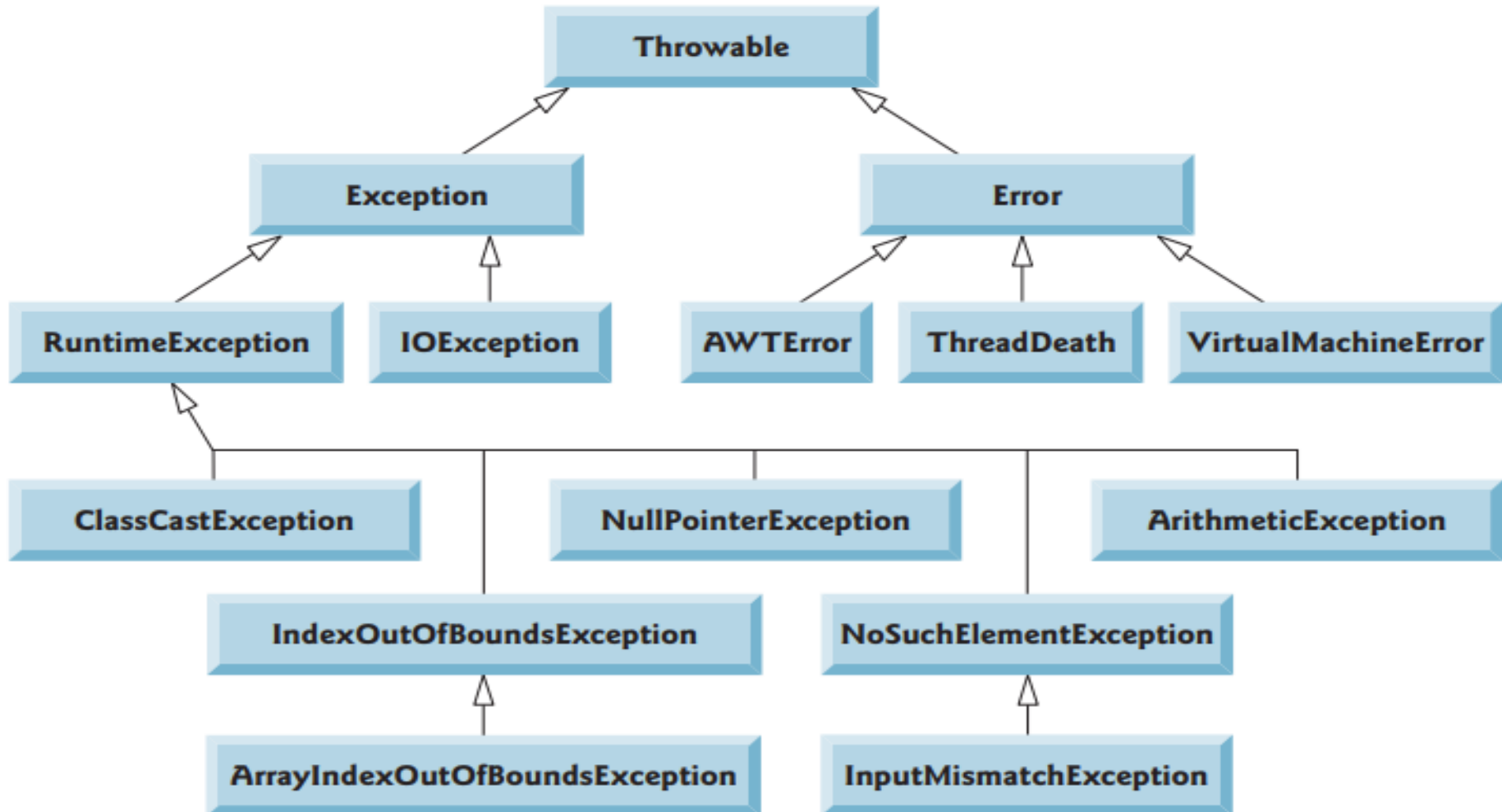
Multi-catch

- É relativamente comum que um bloco **try** seja seguido por vários blocos **catch** para tratar vários tipos de exceção. Se os corpos dos vários blocos **catch** forem idênticos, use o recurso **multi-catch** (introduzido no Java SE 7) para capturar esses tipos de exceção em uma única rotina de tratamento catch e realizar a mesma tarefa. A sintaxe para um **multi-catch** é:

catch (Tipo1 | Tipo2 | Tipo3 e)

- Cada tipo de exceção é separado do seguinte por uma barra vertical (**|**). A linha de código anterior indica que qualquer um dos tipos (ou suas subclasses) pode ser capturado na rotina de tratamento de exceção. Quaisquer tipos **Throwable** podem ser especificados em um **multi-catch**.

Hierarquia de exceção Java





Tipos de Exceções em Java

Exceções verificadas *versus* não verificadas

- Verificadas (Obrigatórias)
 - Alguns objetos devem ser criados e processados dentro de blocos try/catch
 - Derivam diretamente de ***Exception***;
- Não Verificadas (Opcionais)
 - O programador decide se utiliza **if/else** ou bloco **try/cath**
 - Derivam de ***RunTimeException***



Exceções Verificadas (Obrigatórias)

- Você deve lidar com **exceções verificadas**. Isso resulta em código mais robusto do que aquele que seria criado se você fosse capaz de simplesmente ignorar as exceções.
- Se seu método chamar outros métodos que lançam **exceções verificadas**, essas exceções devem ser ***capturadas*** ou ***declaradas***. Se uma exceção pode ser significativamente tratada em um método, o método deve capturar a exceção em vez de declará-la.



Exceções Verificadas

- Se um método de ***subclasse*** sobrescreve um método de ***superclasse***, é um **erro** o método de *subclasse* **listar** **mais** exceções em sua cláusula ***throws*** do que o método da *superclasse* lista. Entretanto, a cláusula ***throws*** de uma *subclasse* pode conter um **subconjunto** da cláusula ***throws*** de uma *superclasse*.



Exceções não Verificadas (Opcionais)

- O compilador Java não examina o código para determinar se uma exceção **não verificada** é *capturada* ou *declarada*. Em geral, pode-se impedir a ocorrência de exceções não verificadas com codificação adequada (***if-else***).
- 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.
- Embora o compilador não imponha o requisito *capture* ou *declare* para as exceções não verificadas, ele fornece o código de tratamento de exceção adequado quando se sabe que tais exceções são possíveis.



Capturando exceções de subclasse

- Se uma rotina de tratamento **catch** for escrita para capturar objetos de exceção de **superclasse**, ele também pode capturar todos os objetos de **subclasses** dessa classe. Isso permite que **catch** trate exceções relacionadas polimorficamente. Você pode capturar cada **subclasse** individualmente se essas exceções exigirem processamento diferente.
- Colocar um bloco **catch** para um tipo de exceção de superclasse antes de outros blocos catch que capturam tipos de exceção de subclasse impediria que esses blocos executem, então ocorre um erro de compilação.



Apenas a primeira **catch** que corresponde é executada

Se múltiplos blocos **catch** correspondem a um tipo particular de exceção, somente o **primeiro** bloco **catch** correspondente executará na ocorrência de uma exceção desse tipo. É um **erro de compilação** capturar *exatamente* o mesmo tipo em dois blocos **catch** diferentes associados com um bloco **try** particular.



Apenas a primeira **catch** que corresponde é executada

Entretanto, pode haver vários blocos **catch** que correspondam a uma exceção — isto é, vários blocos **catch** cujos tipos forem os mesmos que o tipo de exceção ou uma *superclasse* desse tipo. Por exemplo, poderíamos seguir um bloco **catch** para o tipo *ArithmeticException* com um bloco **catch** para o tipo *Exception* — ambos corresponderiam às *ArithmeticExceptions*, mas somente o primeiro bloco **catch** correspondente executaria.



Capturando exceções

- A captura de tipos de *subclasse* individualmente está sujeita a erro se você se esquecer de testar um ou mais dos tipos de *subclasse* explicitamente;
- Capturar a *superclasse* garante que os objetos de **todas** as *subclasses* serão capturados. Posicionar um bloco **catch** para o tipo de *superclasse* depois de todos os outros blocos **catch** de *subclasse* garante que **todas** as exceções de *subclasses* são por fim capturadas.



Bloco *finally*

- O bloco *finally* (que consiste na palavra-chave *finally*, seguida pelo código entre chaves), às vezes referido como a cláusula *finally*, é **opcional**. Se estiver presente, ele é colocado depois do último bloco *catch*. Se não houver blocos *catch*, o bloco *finally*, se presente, segue imediatamente o bloco *try*.



Quando o bloco **finally** é executado

- O **finally** será executado se uma exceção **for** ou **não** lançada no bloco **try** correspondente.
- O bloco **finally** também será executado se um bloco **try** for fechado usando uma instrução **return**, **break** ou **continue** ou simplesmente quando alcança a chave de fechamento direita.



Quando o bloco **finally** é executado

- O caso em que o bloco **finally** **não** executará, é se o aplicativo sair precocemente do bloco **try** chamando o método ***System.exit()***.
- O bloco **finally** é um lugar ideal para liberar os recursos adquiridos em um bloco **try** (como arquivos abertos), o que ajuda a eliminar vazamentos de recurso.



Demonstrando o bloco *finally*

```
1 // Figura 11.5: UsingExceptions.java
2 // mecanismo de tratamento de exceção try...catch...finally.
3
4 public class UsingExceptions
5 {
6     public static void main(String[] args)
7     {
8         try
9         {
10             throwException();
11         }
12         catch (Exception exception) // exceção lançada por throwException
13         {
14             System.err.println("Exception handled in main");
15         }
16
17         doesNotThrowException();
18     }
19
```

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

```
20 // demonstra try...catch...finally
21 public static void throwException() throws Exception
22 {
23     try // lança uma exceção e imediatamente a captura
24     {
25         System.out.println("Method throwException");
26         throw new Exception(); // gera a exceção
27     }
28     catch (Exception exception) // captura exceção lançada em try
29     {
30         System.err.println(
31             "Exception handled in method throwException");
32         throw exception; // lança novamente para processamento adicional
33
34         // o código aqui não seria alcançado; poderia causar erros de compilação
35
36     }
37     finally // executa independentemente do que ocorre em try...catch
38     {
39         System.err.println("Finally executed in throwException");
40     }
41
42     // o código aqui não seria alcançado; poderia causar erros de compilação
43
44 }
45
```

```

45
46 // demonstra finally quando nenhuma exceção ocorrer
47 public static void doesNotThrowException()
48 {
49     try // bloco try não lança uma exceção
50     {
51         System.out.println("Method doesNotThrowException");
52     }
53     catch (Exception exception) // não executa
54     {
55         System.err.println(exception);
56     }
57     finally // executa independentemente do que ocorre em try...catch
58     {
59         System.err.println(
60             "Finally executed in doesNotThrowException");
61     }
62
63     System.out.println("End of method doesNotThrowException");
64 }
65 } // fim da classe UsingExceptions

```

```

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

```



Lançando uma exceção

- Métodos que lançam exceções obrigatórias **devem** citar em seu protótipo a palavra reservada ***throws*** seguida do tipo de exceção que é lançada em seu interior.
- Utiliza-se a palavra reservada ***throw*** para lançar uma exceção.
- Uma instrução ***throw*** especifica um objeto a ser lançado. O operando de um ***throw*** pode ser de qualquer classe derivada da classe ***Throwable***.



Lançando uma exceção

- Lance exceções de construtores para indicar que os parâmetros de construtor não são válidos — isso evita que um objeto seja criado em um estado inválido.
- Se uma exceção não tiver sido capturada quando o controle entrar em um bloco *finally* e esse bloco lançar uma exceção que não será capturada por ele, a primeira exceção será perdida e a exceção do bloco será retornada ao método chamador.



Lançando uma exceção

- Evite inserir em um bloco ***finally*** código que pode usar ***throw*** para lançar uma exceção. Se esse código for necessário, inclua o código em um ***try...catch*** dentro do bloco ***finally***.
- *Supor que* uma exceção lançada de um bloco ***catch*** será processada por esse bloco ***catch*** ou qualquer outro bloco ***catch*** associado com a mesma instrução ***try*** pode resultar em erros de lógica.



printStackTrace e getMessage

- Métodos utilizados para apresentar as informações sobre a exceção que foi gerada.
- `StackTraceElement[] traceElements = exception.printStackTrace();`
 - `element.getClassName()`
 - `element.getFileName()`
 - `element.getLineNumber()`
 - `element.getMethodName()`;

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

```

1 // Figura 11.6: UsingExceptions.java
2 // Desempilhando e obtendo dados a partir de um objeto de exceção.
3
4 public class UsingExceptions
5 {
6     public static void main(String[] args)
7     {
8         try
9         {
10             method1();
11         }
12         catch (Exception exception) // captura a exceção lançada em method1
13         {
14             System.err.printf("%s%n%n", exception.getMessage());
15             exception.printStackTrace();
16
17             // obtém informações de rastreamento de pilha
18             StackTraceElement[] traceElements = exception.getStackTrace();
19
20             System.out.printf("%nStack trace from getStackTrace:%n");
21             System.out.println("Class\t\tFile\t\tLine\tMethod");
22
23             // faz um loop por traceElements para obter a descrição da exceção
24             for (StackTraceElement element : traceElements)
25             {
26                 System.out.printf("%s\t", element.getClassName());
27                 System.out.printf("%s\t", element.getFileName());

```

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)

```

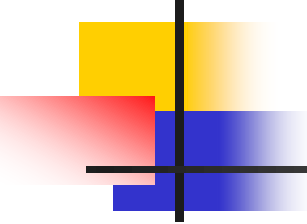
Stack trace from getStackTrace:

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

```

28         System.out.printf("%s\t", element.getLineNumber());
29         System.out.printf("%s%n", element.getMethodName());
30     }
31 }
32 } // fim de main
33
34 // chama method2; lança exceções de volta para main
35 public static void method1() throws Exception
36 {
37     method2();
38 }
39
40 // chama method3; lança exceções de volta para method1
41 public static void method2() throws Exception
42 {
43     method3();
44 }
45
46 // lança Exception de volta para method2
47 public static void method3() throws Exception
48 {
49     throw new Exception("Exception thrown in method3");
50 }
51 } // fim da classe UsingExceptions

```

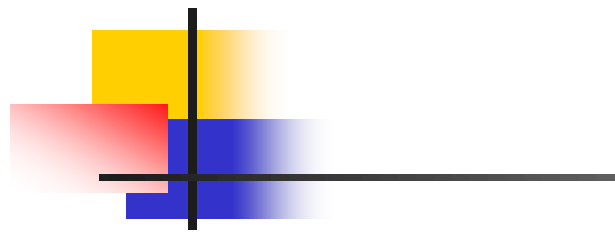


Exception thrown in method3

```
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



Compile e execute o programa mostrado na listagem ao lado.

Verifique os resultados que o programa dará para a execução do programa como:

```
java DemoExcecoes 0 1 2
java DemoExcecoes a b c d
java DemoExcecoes 0 0 0 0
java DemoExcecoes 0 1 0 1
java DemoExcecoes
```

```
1  /* Classe que corrige erros em potencial em um programa exemplo */
2  class DemoExcecoes
3  {
4      /* Método que permite a execução da classe */
5      public static void main(String args[])
6      { // tentaremos executar o bloco de comandos abaixo
7          try
8          { // Supomos que quatro argumentos serão passados para o
9            // programa, e podemos transformar estes argumentos em valores int
10              int a1 = Integer.parseInt(args[0]);
11              int a2 = Integer.parseInt(args[1]);
12              int b1 = Integer.parseInt(args[2]);
13              int b2 = Integer.parseInt(args[3]);
14              // Efetuamos uma operação simples com estes valores.
15              int resultado = a1/a2 + b1/b2;
16              System.out.println("O resultado é " + resultado);
17          }
18          // se alguma exceção foi jogada, ela será pega por este bloco
19          catch (Exception e)
20          { // é necessário passar uma instância de uma classe de exceções !
21              System.out.println("Uma exceção ocorreu: " + e);
22          }
23      } // fim do método main
24  } // fim da classe DemoExcecoes
```



Assertions

- Assertivas ajudam a capturar potenciais bugs e identificar possíveis erros de lógica.
- A instrução ***assert*** permite validar as afirmações de forma programática.
- Para permitir assertivas em tempo de execução, utilize a switch ***-ea*** ao executar o comando *java*



Assertions

- `assert expression;`
 - This statement evaluates *expression* and throws an ***AssertionError*** if the expression is false.
- `assert expression1 : expression2;`
 - This statement evaluates *expression1* and throws an ***AssertionError*** with *expression2* as the error message if *expression1* is false




Assertion

- You can use assertions to programmatically implement **preconditions** and **postconditions** or to verify any other intermediate states that help you ensure your code is working correctly.
- **java -ea AssertTest**

```
1 // Fig. 13.9: AssertTest.java
2 // Demonstrates the assert statement
3 import java.util.Scanner;
4
5 public class AssertTest
6 {
7     public static void main( String args[] )
8     {
9         Scanner input = new Scanner( System.in );
10
11         System.out.print( "Enter a number between 0 and 10: " );
12         int number = input.nextInt();
13
14         // assert that the absolute value is >= 0
15         assert ( number >= 0 && number <= 10 ) : "bad number: " + number;
16
17         System.out.printf( "You entered %d\n", number );
18     } // end main
19 } // end class AssertTest
```

Enter a number between 0 and 10: 5
You entered 5



```
1 // Fig. 13.9: AssertTest.java
2 // Demonstrates the assert statement
3 import java.util.Scanner;
4
5 public class AssertTest
6 {
7     public static void main( String args[] )
8     {
9         Scanner input = new Scanner( System.in );
10
11         System.out.print( "Enter a number between 0 and 10: " );
12         int number = input.nextInt();
13
14         // assert that the absolute value is >= 0
15         assert ( number >= 0 && number <= 10 ) : "bad number: " + number;
16
17         System.out.printf( "You entered %d\n", number );
18     } // end main
19 } // end class AssertTest
```

```
Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
    at AssertTest.main(AssertTest.java:15)
```



try com recursos: desalocação automática de recursos

- A forma geral de uma instrução try com recursos é:

```
try (NomeDaClasse theObject = new NomeDaClasse())  
{  
    // usa theObject aqui  
}  
catch (Exception e)  
{  
    // captura exceções que ocorrem durante o uso do recurso  
}
```

onde *NomeDaClasse* é uma classe que implementa a interface **AutoCloseable**.



try com recursos: desalocação automática de recursos

- Cada recurso deve ser um objeto de uma classe que implementa a interface ***AutoCloseable*** e, portanto, fornece um método *close*.
- É possível atribuir vários recursos nos parênteses depois de **try** separando-os com um ponto e vírgula (;)



Referencias

- Java How to program 3, a 10 ed.
Deitel e Deitel