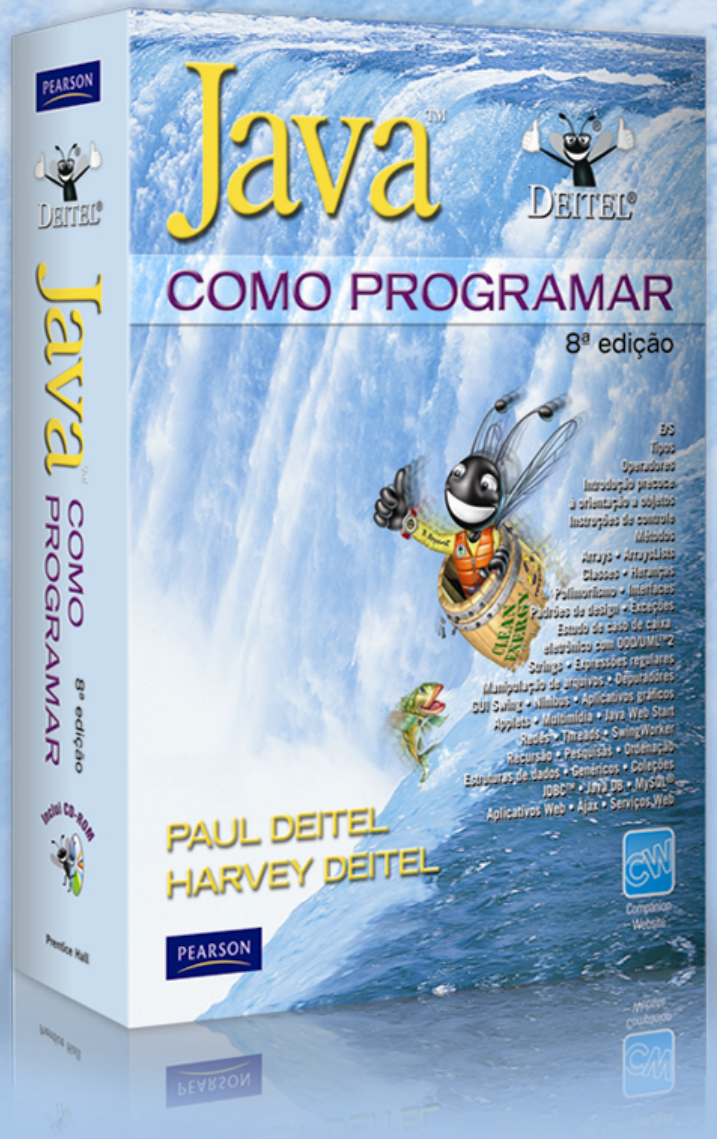
Capítulo 11
Tratamento de exceções

Java™ Como Programar, 8/E



COMO PROGRAMAR

8ª edição

OBJETIVOS

Neste capítulo, você aprenderá:

- O que são exceções.
- Como o tratamento de exceções e de erros funciona.
- A utilizar `try`, `throw` e `catch` para detectar, indicar e tratar exceções, respectivamente.
- A utilizar o bloco `finally` para liberar recursos.
- Como o desempilhamento permite que exceções não capturadas em um escopo sejam capturadas em outro.
- Como os rastreamentos de pilha ajudam na depuração.
- Como as exceções são organizadas em uma hierarquia de classes de exceção.
- A declarar novas classes de exceção.
- A criar exceções encadeadas que mantêm informações do rastreamento de pilha completo.

COMO PROGRAMAR

8ª edição

11.1 Introdução

- **Tratamento de exceções**
- **Exceção** — uma indicação de um problema que ocorre durante a execução de um programa.

  O nome "exceção" significa que o problema não ocorre frequentemente.
- Com o tratamento de exceções, um programa pode continuar executando (em vez de encerrar) depois de lidar com um problema.

  Computação de missão crítica ou de negócios críticos.

  Programas robustos e **tolerantes a falhas** (isto é, programas que podem lidar com problemas à medida que eles surgem e continuar a executar).

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 11.1

O tratamento de exceções ajuda a aprimorar a tolerância a falhas de um programa.

- `ArrayIndexOutOfBoundsException` ocorre quando é feita uma tentativa de acessar um elemento depois do final de um array.
- `ClassCastException` ocorre quando é feita uma tentativa de fazer uma coerção em um objeto que não tem um relacionamento *é um* com o tipo especificado no operador de coerção.
- Uma `NullPointerException` ocorre quando uma referência `null` é utilizada onde um objeto é esperado.
- Somente classes que estendem `Throwable` (pacote `java.lang`) direta ou indiretamente podem ser utilizadas com o tratamento de exceções.

11.2 Visão geral do tratamento de erros

- Os programas costumam testar condições para determinar como a execução do programa deve prosseguir.
- Considere o seguinte pseudocódigo:

  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

  …

  Inicie executando uma tarefa; então teste se ele executou corretamente.

  Em caso negativo, realize processamento de erro.

  Caso contrário, continue com a próxima tarefa.
- Misturar programa e lógica de tratamento de erro desta maneira pode tornar os programas difíceis de ler, modificar, manter e depurar especialmente em grandes aplicativos.

Java™ COMO PROGRAMAR 8ª edição

Dica de desempenho 11.1

Se os problemas potenciais ocorrem raramente, mesclar o programa e a lógica do tratamento de erro pode degradar o desempenho do programa, porque o programa deve potencialmente realizar testes frequentes para determinar se a tarefa foi executada corretamente e se a próxima tarefa pode ser realizada.

- O tratamento de exceções permite aos programadores remover código de tratamento de erro da "linha principal" de execução do programa.
  - Aprimora a clareza do programa.
  - Aprimora a modificabilidade.
- Trate qualquer exceção que você escolha.
  - Todas as exceções.
  - Todas as exceções de certo tipo.
  - Todas as exceções de um grupo de tipos relacionados (isto é, relacionados por meio de uma superclasse).
- Essa flexibilidade reduz a probabilidade de que erros serão negligenciados, tornando assim os programas mais robustos.

11.3 Exemplo: Divisão por zero sem tratamento de exceções

- As exceções são **lançadas** (isto é, a exceção ocorre) quando um método detecta um problema e é incapaz de tratá-lo.
- **Rastreamento de pilha** — informações exibidas quando uma exceção ocorre e não é tratada.
- As informações incluem:

  O nome da exceção em uma mensagem descritiva que indica o problema que ocorreu.

  A pilha de chamada do método (isto é, a cadeia de chamadas) no momento em que a exceção ocorreu. Representa o caminho de execução que levou à exceção método por método.
- Essas informações o ajudam a depurar o programa.

- O Java não permite divisão por zero na aritmética de números inteiros.

  Lança uma **ArithmeticException.**

  Pode surgir de vários problemas, portanto, uma mensagem de erro (por exemplo, "`/ by zero`") fornece informações mais específicas.

- O Java *realmente* permite a divisão por zero com valores de ponto flutuante.

  Um cálculo como esse resulta em um valor infinito positivo ou negativo.

  Valor de ponto flutuante que exibe como `Infinity` ou `-Infinity`.

  Se 0.0 for dividido por 0.0, o resultado será NaN (não um número), que é representado como um valor de ponto flutuante que exibe como `NaN`.


```
1  // Figura 11.1: DivideByZeroNoExceptionHandling.java
2  // Divisão de inteiro sem tratamento de exceções.
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    } // fim do método quotient
12
13    public static void main( String[] args )
14    {
15       Scanner scanner = new Scanner( System.in ); // scanner para entrada
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 );
```

A JVM lança uma exceção se `denominator` for 0

O usuário poderia digitar uma entrada inválida

O usuário poderia digitar uma entrada inválida (incluindo 0)

Figura 11.1 | Divisão de inteiro sem tratamento de exceções. (Parte 1 de 3.)


```
23          System.out.printf(
24             "\nResult: %d / %d = %d\n", numerator, denominator, result );
25       } // fim de main
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)
```

Causa divisão por 0; o rastreamento da pilha mostra o que levou à exceção

Figura 11.1 | Divisão de inteiro sem tratamento de exceções. (Parte 2 de 3.)


```
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)
```

O usuário digitou valor de números não inteiros; o rastreamento da pilha mostra o que levou à exceção

Figura 11.1 | Divisão de inteiro sem tratamento de exceções. (Parte 3 de 3.)

- A última linha "at" do rastreamento de pilha iniciou a cadeia de chamadas.
- Cada linha contém o nome da classe e o método seguidos pelo nome do arquivo e o número da linha.
- A linha "at" superior da cadeia de chamadas indica o **ponto de lançamento** — o ponto inicial em que a exceção ocorre.
- À medida que você lê um rastreamento de pilha de cima para baixo, a primeira linha "at" que contém o nome da sua classe e o nome do seu método é, em geral, o ponto no programa que levou à exceção.

- Os exemplos anteriores que leem valores numéricos a partir do usuário assumiram que o usuário iria inserir um valor de número inteiro adequado.
- Às vezes, os usuários cometem erros e inserem valores não inteiros.
- Uma **InputMismatchException** ocorre quando o método `Scanner` `nextInt` recebe uma `String` que não representa um inteiro válido.
- Se um rastreamento de pilha contiver "`Unknown Source`" para um método em particular, os símbolos de depuração para a classe desse método não estavam disponíveis para a JVM — esse é tipicamente o caso das classes da Java API.

11.4 Exemplo: Tratando `ArithmeticException`s e `InputMismatchException`s

- O aplicativo na Fig. 11.2 utiliza o tratamento de exceções para processar quaisquer `ArithmeticException`s e `InputMistmatchException`s que surgirem.
- Se o usuário cometer um erro, o programa captura e trata (isto é, lida com) a exceção — nesse caso, permitindo ao usuário tentar inserir a entrada novamente.


```
1  // Figura 11.2: 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    } // fim do método quotient
14
15    public static void main( String[] args )
16    {
17       Scanner scanner = new Scanner( System.in ); // scanner para entrada
18       boolean continueLoop = true; // determina se mais entradas são necessárias
19
```

Tipo de exceção lançada por vários métodos da classe `Scanner`

Indica que esse método talvez lance uma `ArithmeticException`

Figura 11.2 | Tratando `ArithmeticExceptions` e `InputMismatchExceptions`. (Parte 1 de 4.)


```
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             } // fim do try
34             catch ( InputMismatchException inputMismatchException )
35             {
36                System.err.printf( "\nException: %s\n",
37                   inputMismatchException );
38                scanner.nextLine(); // descarta entrada para o usuário poder tentar de novo
39                System.out.println(
40                   "You must enter integers. Please try again.\n" );
41             } // fim do catch
```

Inicia um bloco de código no qual uma exceção talvez ocorra; o bloco também contém código que não deveria executar se uma exceção ocorrer

Captura e processa InputMismatchExceptions

Figura 11.2 | Tratando `ArithmeticExceptions` e `InputMismatchExceptions`. (Parte 2 de 4.)


```
42            catch ( ArithmeticException arithmeticException )
43            {
44               System.err.printf( "\nException: %s\n", arithmeticException );
45               System.out.println(
46                  "Zero is an invalid denominator. Please try again.\n" );
47            } // fim do catch
48         } while ( continueLoop ); // fim da instrução do...while
49      } // fim de main
50   } // fim da classe DivideByZeroWithExceptionHandling
```

Captura e processa ArithmeticExceptions

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7
Result: 100 / 7 = 14
```

Figura 11.2 | Tratando `ArithmeticExceptions` e `InputMismatchExceptions`. (Parte 3 de 4.)

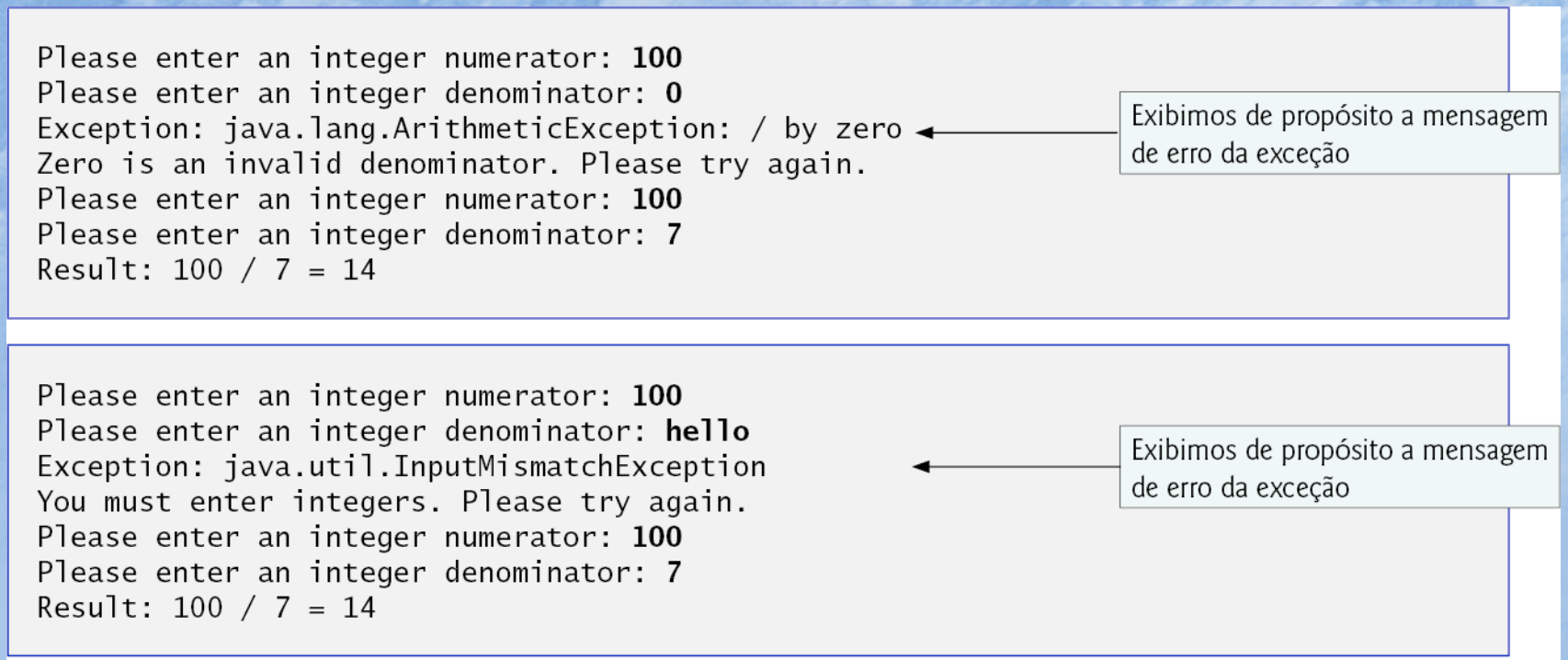


Figura 11.2 | Tratando `ArithmeticExceptions` e `InputMismatchExceptions`. (Parte 4 de 4.)

- O **bloco try** envolve o código que poderia lançar (`throw`) uma exceção e o código não deve executar se uma exceção ocorrer.
- Consiste na palavra-chave `try` seguida por um bloco de código entre chaves.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 11.1

As exceções emergem pelo código explicitamente mencionado em um bloco `try`*, por chamadas para outros métodos, por chamadas de método profundamente aninhadas iniciadas pelo código em um bloco* `try` *ou a partir da Java Virtual Machine à medida que ela executa os bytecodes do Java.*

- O **bloco catch** (também chamado **cláusula catch** ou **handler de exceção**) captura e trata uma exceção.

  Inicia com a palavra-chave `catch` e é seguido por um parâmetro de exceção entre parênteses e um bloco de código entre chaves.
- Pelo menos um bloco `catch` ou um **bloco finally** (Seção 11.7) deve se seguir imediatamente ao bloco `try`.
- O **parâmetro de exceção** identifica o tipo de exceção que o handler pode processar.

  O nome do parâmetro de exceção permite ao bloco `catch` interagir com um objeto de exceção capturado.

- Quando ocorre uma exceção em um bloco `try`, o bloco `catch` que executa é o primeiro cujo tipo corresponde ao tipo de exceção que ocorreu.
- Utilize o **objeto System.err (fluxo de erro padrão)** para imprimir mensagens de erro.

  Por padrão, exibe dados para o prompt de comando.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 11.1

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

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 11.2

Cada bloco `catch` *pode ter apenas um único parâmetro — especificar uma lista de parâmetros de exceção separados por vírgulas é um erro de sintaxe.*

- **Exceção não capturada** — aquela para a qual não há blocos `catch`.
- Lembre-se de que as exceções não capturadas anteriores fizeram com que o aplicativo terminasse prematuramente.

  Isso nem sempre ocorre como um resultado de exceções não capturadas.
- O Java utiliza um modelo de múltiplas threads para a execução de programas.

  Cada **thread** é uma atividade paralela.

  Um programa pode ter muitas threads.

  Se um programa tiver apenas uma thread, uma exceção não capturada fará com que o programa seja encerrado.

  Se um programa tiver múltiplas threads, uma exceção não capturada encerrará apenas a thread em que ocorreu a exceção.

- Se ocorrer uma exceção em um bloco `try`, esse bloco terminará imediatamente e o controle do programa será transferido para o primeiro bloco `catch` correspondente.
- Após uma exceção ser tratada, o controle é retomado logo depois do último bloco `catch`.
- Isso é conhecido como o **modelo de terminação de tratamento de exceções.**

  Algumas linguagens utilizam o **modelo de retomada de tratamento de exceções**, no qual, após uma exceção ser tratada, o controle é retomado logo depois do ponto de lançamento.

Java™ COMO PROGRAMAR 8ª edição

Boa prática de programação 11.1

Utilizar um nome de parâmetro de exceção que reflita o tipo do parâmetro promove a clareza lembrando-lhe do tipo de exceção em tratamento.

- Se nenhuma exceção for lançada em um bloco `try`, os blocos `catch` são pulados e o controle continua com a primeira instrução depois dos blocos `catch`. Aprenderemos outra possibilidade ao discutirmos o bloco `finally` na Seção 11.7.
- O bloco `try` e seus blocos `catch` e/ou `finally` correspondentes formam uma **instrução try.**

- Quando um bloco `try` termina, as variáveis locais declaradas no bloco saem de escopo.

  As variáveis locais de um bloco `try` não são acessíveis em blocos `catch` correspondentes.
- Quando um bloco `catch` termina, as variáveis locais declaradas dentro do bloco `catch` (incluindo o parâmetro de exceção) também saem de escopo.
- Quaisquer blocos `catch` restantes na instrução `try` são ignorados, e a execução é retomada na primeira linha de código depois da sequência `try`...`catch`.

  Um bloco `finally`, se algum estiver presente.

- **Cláusula throws** — especifica as exceções que um método lança.
  Aparece depois da lista de parâmetros do método e antes do corpo do método.
  Contém uma lista das exceções separadas por vírgulas que o método lançará se vários problemas ocorrerem.
  Podem ser lançadas por instruções no corpo do método ou por métodos chamados a partir do corpo.
  Um método pode lançar exceções das classes listadas em sua cláusula `throws` ou de suas subclasses.
  Os clientes de um método com uma cláusula `throws` são assim informados de que o método pode lançar exceções.

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 11.2

Leia a documentação online da API para obter informações sobre um método antes de utilizar esse método em um programa. A documentação especifica as exceções lançadas pelo método (se houver alguma) e indica as razões pelas quais tais exceções podem ocorrer. Em seguida, leia a documentação online da API para as classes de exceção especificadas. A documentação para uma classe de exceção costuma conter possíveis razões por que essas exceções ocorrem. Por fim, forneça o tratamento para essas exceções em seu programa.

- Quando um método lança uma exceção, o método termina e não retorna um valor, e suas variáveis locais saem de escopo.

  Se as variáveis locais fossem referências a objetos e não houvesse nenhuma outra referência a esses objetos, os objetos estariam disponíveis para a coleta de lixo.

11.5 Quando utilizar o tratamento de exceções

- O tratamento de exceções é projetado para processar **erros síncronos**, que ocorrem quando uma instrução executa.
- Exemplos comuns nesse livro:
  índices de array fora do intervalo
  estouro aritmético
  divisão por zero
  parâmetros de método inválidos
  interrupção de thread
  alocação de memória malsucedida

- O tratamento de exceções não é projetado para processar problemas associados com **eventos assíncronos**, a exemplo de:
  - conclusões de E/S de disco,
  - chegadas de mensagem de rede,
  - cliques de mouse e pressionamentos de tecla.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 11.2

Incorpore sua estratégia de tratamento de exceções ao sistema desde o princípio do processo de design. Pode ser difícil incluir um tratamento de exceções depois que um sistema foi implementado.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 11.3

O tratamento de exceções fornece uma técnica única e uniforme para processamento de problemas. Isso ajuda os programadores que trabalham em grandes projetos a entender o código de processamento de erro uns dos outros.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 11.4

O tratamento de exceções simplifica a combinação de componentes de software e permite trabalhar em conjunto eficientemente possibilitando que os componentes predefinidos comuniquem problemas para componentes específicos ao aplicativo, que então podem processar os problemas de maneira específica ao aplicativo.

11.6 Hierarquia de exceções Java

- As classes de exceção herdam direta ou indiretamente da classe **Exception**, formando uma hierarquia de herança.
  Pode estender essa hierarquia com suas próprias classes de exceção.
- A Figura 11.3 mostra uma pequena parte da hierarquia de herança da classe **Throwable** (uma subclasse de `Object`), que é a superclasse da classe Exception.
  Somente objetos `Throwable` podem ser utilizados com o mecanismo de tratamento de exceções.
- A classe `Throwable` tem duas subclasses: `Exception` e `Error`.

- A classe `Exception` e suas subclasses representam situações excepcionais que podem ocorrer em um programa Java

  Essas subclasses podem ser capturadas e tratadas pelo aplicativo.
- A classe **Error** e suas subclasses representam situações anormais que podem acontecer na JVM.

  `Erro`s não acontecem frequentemente.

  Eles não devem ser capturados pelos aplicativos.

  Os aplicativos normalmente não se recuperam de `Error`s.

Java™ COMO PROGRAMAR 8ª edição

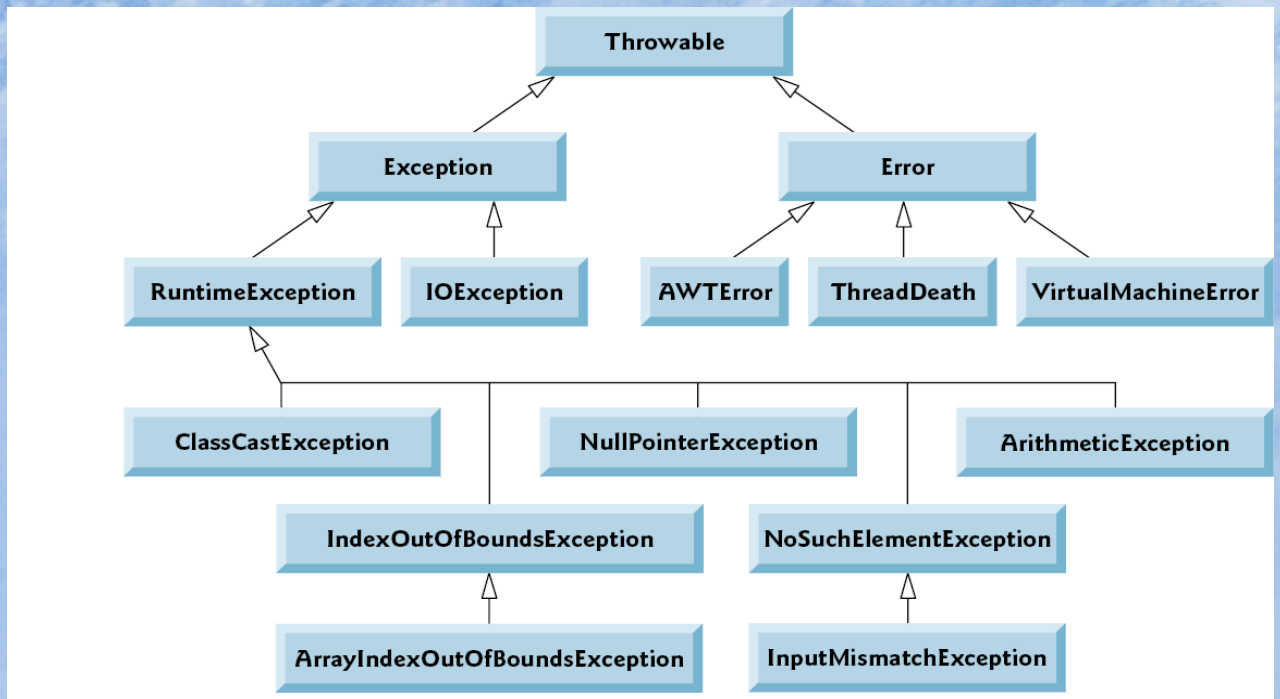


Figura 11.3 | Parte da hierarquia de herança da classe `Throwable`.

- **Exceções verificadas** versus **exceções não verificadas.**

  O compilador impõe um **requisito catch-or-declare** para exceções verificadas.
- O tipo de uma exceção determina se a exceção é verificada ou não verificada.
- As subclasses diretas ou indiretas da classe **RuntimeException** (pacote `java.lang`) são exceções *não verificadas*.

  Costumam ser causadas por deficiências no código do seu programa (por exemplo, `ArrayIndexOutOfBoundsException`s).
- As subclasses de `Exception`, mas não `RuntimeException`, são exceções *verificadas*.

  São causadas por condições que não estão no controle do programa — por exemplo, no processamento de arquivos, o programa não pode abrir um arquivo porque o arquivo não existe.

- As classes que herdam da classe `Error` são consideradas *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 verifica se a exceção verificada é capturada ou declarada em uma cláusula `throws`.
- A cláusula **throws** especifica as exceções que o método lança.

  Tais exceções normalmente não são capturadas no corpo do método.

- Para satisfazer a parte *catch* do requisito *catch-or-declare*, o código que gera a exceção deve ser empacotado em um bloco `try` e deve fornecer um handler `catch` para o tipo de exceção verificada (ou uma de suas superclasses).
- Para satisfazer a parte *declare* do requisito *catch-or-declare*, o método deve fornecer uma cláusula `throws` contendo o tipo de exceção verificada depois de sua lista de parâmetros e antes do corpo do método.
- 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.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 11.5

Você precisa 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.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 11.3

Um erro de compilação ocorre se um método tentar explicitamente lançar uma exceção verificada (ou chamar outro método que lança uma exceção verificada) e essa exceção não estiver listada na cláusula `throws` *do método.*

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 11.4

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 sobrescrito da superclasse. Mas a cláusula* `throws` *de uma subclasse pode conter um subconjunto da lista de* `throws` *de uma superclasse.*

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 11.6

Se o método chamar outros métodos que lançam explicitamente exceções verificadas, essas exceções devem ser capturadas ou declaradas no método. Se uma exceção pode ser significativamente tratada em um método, o método deve capturar a exceção em vez de declará-la.

- O compilador do Java não verifica o código para determinar se uma exceção não verificada é capturada ou declarada.

  Em geral, pode-se impedir essas exceções com a codificação adequada.

  Por exemplo, uma `ArithmeticException` pode ser evitada se o método assegurar que o denominador não é zero antes de tentar realizar a divisão.
- Não é necessário que as exceções não verificadas sejam listadas em uma cláusula `throws` do método.

  Ainda que sejam, não se exige que essas exceções sejam capturadas por um aplicativo.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 11.7

Embora o compilador não imponha o requisito capture ou declare para as exceções não verificadas, ele fornece o código de tratamento de exceções adequado quando se sabe que tais exceções são possíveis. Por exemplo, um programa deve processar a `NumberFormatException` *do método* `Integer parseInt`*, mesmo que* `NumberFormatException` *(uma subclasse de* `RuntimeException`*) seja um tipo de exceção não verificada. Isso torna os programas mais robustos.*

- Um parâmetro `catch` de um tipo de superclasse também pode capturar todos os tipos de subclasse desse tipo de exceção.
  Permite a `catch` tratar erros relacionados com uma notação concisa.
  Permite o processamento polimórfico de exceções relacionadas.
  Capturar exceções relacionadas em um bloco `catch` só faz sentido se o comportamento do tratamento for o mesmo para todas as subclasses.
- Você também pode capturar cada tipo de subclasse individualmente se essas exceções exigirem processamento diferente.

- Se houver múltiplos blocos `catch` que correspondem a um tipo particular de exceção, somente o primeiro bloco `catch` correspondente executará.
- É um erro de compilação capturar o mesmo tipo exato em dois blocos `catch` diferentes associados com um bloco `try` específico.

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 11.3

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 para subclasses dessa superclasse assegura que todas as exceções de subclasse sejam por fim capturadas.*

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 11.5

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.*

11.7 Bloco finally

- Os programas que obtêm certos tipos de recursos devem retorná-los ao sistema explicitamente para evitar os supostos **vazamentos de recursos.**

  Em linguagens de programação como C e C++, o tipo mais comum de vazamento de recurso é um vazamento de memória.

  O Java realiza coleta automática de lixo da memória não mais utilizada por programas, evitando assim a maioria dos vazamentos de memória.

  Outros tipos de vazamentos de recursos podem ocorrer.

  Arquivos, conexões de banco de dados e conexões de rede que não são fechadas adequadamente talvez não estejam disponíveis para uso em outros programas.
- O bloco `finally` é utilizado para desalocação de recurso.
  É posicionado depois do último bloco `catch`.

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 11.4

Uma questão sutil é que o Java não elimina inteiramente os vazamentos de memória. O Java não efetuará coleta de lixo de um objeto até que não haja nenhuma referência a ele. Portanto, se os programadores mantiverem erroneamente referências a objetos indesejáveis, vazamentos de memória podem ocorrer. Para ajudar a evitar esse problema, configure variáveis de tipo por referência como `null`*, quando elas não são mais necessárias.*

```
try
{
    instruções
    instruções de aquisição de recursos
} // fim do try
catch ( UmTipoDeExceção exceção1 )
{
    instruções de tratamento de exceções
} // fim do catch
...
catch ( OutroTipoDeExceção exceção2 )
{
    instruções de tratamento de exceções
} // fim do catch
finally
{
    instruções
    instruções de liberação de recursos
} // fim de finally
```

Figura 11.4 | Uma instrução `try` com um bloco `finally`.

- O bloco `finally` executará se uma exceção for lançada no bloco `try` correspondente.
- O bloco `finally` executará se um bloco `try` fechar utilizando uma instrução `return`, `break` ou `continue` ou simplesmente alcançando sua chave direita de fechamento.
- O bloco finally *não* executará se o aplicativo terminar imediatamente chamando o método System.exit.

- Como um bloco `finally` quase sempre executa, em geral, ele contém código de liberação de recursos.
- Suponha que um recurso esteja alocado em um bloco `try`.

  Se nenhuma exceção ocorrer, o controle passa para o bloco `finally`, que libera o recurso. O controle então prossegue à primeira instrução depois do bloco `finally`.

  Se uma exceção ocorrer, o bloco `try` termina. O programa captura e processa a exceção em um dos blocos `catch` correspondentes, então o bloco `finally` libera o recurso e o controle prossegue na primeira instrução depois do bloco `finally`.

  Se o programa não capturar a exceção, o bloco `finally` ainda libera o recurso e é feita uma tentativa de capturar a exceção em um método de chamada.

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 11.5

O bloco `finally` *é um lugar ideal para liberar recursos adquiridos em um bloco* `try` *(como arquivos abertos), o que ajuda a eliminar vazamentos de recurso.*

Java™ COMO PROGRAMAR 8ª edição

Dica de desempenho 11.2

Sempre libere um recurso explicitamente e logo que ele não for mais necessário. Isso torna os recursos disponíveis para reutilização, aprimorando assim a utilização deles.

- Se uma exceção que ocorre em um bloco `try` não puder ser capturada por um dos handlers `catch` desses blocos `try`, o controle passa para o bloco `finally`.
- Então o programa passa a exceção para o próximo bloco `try` externo — normalmente no método de chamada —, onde um bloco `catch` associado talvez possa capturá-lo.

  Esse processo pode ocorrer pelos muitos níveis dos blocos `try`.

  A exceção talvez não seja capturada.
- Se um bloco `catch` lançar uma exceção, o bloco `finally` ainda executará.

  Então a exceção é passada para o próximo bloco `try` externo — novamente, normalmente no método de chamada.


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

Inicia uma cadeia de chamada na qual a exceção será lançada

Inicia uma cadeia de chamada na qual nenhuma exceção ocorre

Figura 11.5 | Mecanismo de tratamento de exceções. (Parte 1 de 4.)


```
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         } // fim do try
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; causaria erros de compilação
35
36         } // fim do catch
37         finally // executa independentemente do que ocorre em try...catch
38         {
39            System.err.println( "Finally executed in throwException" );
40         } // fim de finally
41
42         // o código aqui não seria alcançado; causaria erros de compilação
43
44      } // fim do método throwException
```

Esse método talvez lance uma `Exception` (esse é um tipo *verificado*)

Lança uma nova `Exception` que é capturada na linha 28 e lançada novamente na linha 32

Relançar uma exceção significa que ela não é considerada como tendo sido tratada

Esse bloco executa mesmo que a linha 32 no handler `catch` lançasse uma exceção; então o método termina

Figura 11.5 | Mecanismo de tratamento de exceções. (Parte 2 de 4.)

```
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          } // fim do try
53          catch ( Exception exception ) // não executa
54          {
55             System.err.println( exception );
56          } // fim do catch
57          finally // executa independentemente do que ocorre em try...catch
58          {
59             System.err.println(
60                "Finally executed in doesNotThrowException" );
61          } // fim de finally
62
63          System.out.println( "End of method doesNotThrowException" );
64       } // fim do método doesNotThrowException
65    } // fim da classe UsingExceptions
```

Esse método não lança nenhuma exceção

Esse bloco `try` executará todas as suas instruções corretamente

O handler `catch` será pulado; nenhuma exceção ocorre

Esse bloco `finally` ainda executa

O controle do programa continua aqui

Figura 11.5 | Mecanismo de tratamento de exceções. (Parte 3 de 4.)


```
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
```

Figura 11.5 | Mecanismo de tratamento de exceções `try...catch...finally`. (Parte 4 de 4.)

- `System.out` e `System.err` são **fluxos** — uma sequência de bytes.
  `System.out` (o **fluxo de saída padrão**) exibe a saída.
  `System.err` (o **fluxo de erro padrão**) exibe os erros.
- A saída desses fluxos pode ser redirecionada (por exemplo, para um arquivo).
- Utilizar dois fluxos diferentes permite-lhe separar facilmente as mensagens de erro de outra saída.
  A saída de dados de `System.err` poderia ser enviada para um arquivo de log.
  A saída de dados de `System.out` pode ser exibida na tela.

- **Instrução throw** — indica que uma exceção ocorreu.

  Utilizada para lançar exceções.

  Indica para o cliente que ocorreu um erro.

  Especifica o objeto a ser lançado.

  O operando de um `throw` pode ser de qualquer classe derivada da classe `Throwable`.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 11.8

Quando toString é invocada em qualquer objeto `Throwable`*, sua* `string` *resultante inclui a* `string` *descritiva que foi fornecida para o construtor, ou simplesmente o nome de classe se nenhuma* `string` *foi fornecida.*

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 11.9

Um objeto pode ser lançado sem conter informações sobre o problema que ocorreu. Nesse caso, simplesmente saber que uma exceção de um tipo particular ocorreu pode fornecer informações suficientes para o handler processar o problema corretamente.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 11.10

As exceções podem ser lançadas a partir de construtores. Quando um erro é detectado em um construtor, uma exceção deve ser lançada para evitar-se a criação de um objeto formado inadequadamente.

- **Relançamento de uma exceção**

  É feito quando um bloco `catch` não pode processar essa exceção ou pode somente processá-la parcialmente.

  Adia o tratamento de exceções (ou talvez uma parte dele) para outro bloco `catch` associado com uma instrução `try` externa.
- Relança uma exceção utilizando a **palavra-chave throw**, seguida por uma referência ao objeto de exceção que acabou de ser capturado.
- Quando ocorre um relançamento, o próximo bloco `try` circundante detecta a exceção e os blocos `catch` desses blocos `try` tentam tratá-la.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 11.6

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.*

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 11.6

Evite colocar código que possa lançar (`throw`) *uma exceção em um bloco* `finally`. *Se esse código for necessário, inclua o código em um* `try...catch` *dentro do bloco* `finally`.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 11.7

Assumir 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.*

Java™ COMO PROGRAMAR 8ª edição

Boa prática de programação 11.2

O mecanismo de tratamento de exceções do Java é projetado para remover código de processamento de erro da linha principal do código de um programa para aprimorar a clareza de programa. Não coloque `try...catch...finally` *em torno de toda instrução que possa lançar uma exceção. Isso torna os programas difíceis de ler. Em vez disso, coloque um bloco* `try` *em torno de uma parte significativa do código, esse bloco* `try` *deve ser seguido por blocos* `catch` *que tratam cada possível exceção e os blocos* `catch` *devem ser seguidos por um único bloco* `finally` *(se algum for necessário).*

COMO PROGRAMAR

8ª edição

11.8 Desempilhamento

- **Desempilhamento** — Quando uma exceção é lançada mas não é capturada em um escopo particular, a pilha do método de chamada é "desempilhada"
- É feita uma tentativa para `capturar` a exceção no próximo bloco `try` externo.
- Todas as variáveis locais do método de desempilhamento saem de escopo e o controle retorna à instrução que originalmente invocou esse método.
- Se um bloco `try` incluir essa instrução, é feita uma tentativa para `capturar` a exceção.
- Se um bloco `try` não incluir essa instrução ou se a exceção não for capturada, ocorre novamente o desempilhamento.


```
1  // Figura 11.6: UsingExceptions.java
2  //Desempilhamento.
3
4  public class UsingExceptions
5  {
6     public static void main( String[] args )
7     {
8        try // chama throwException para demonstrar o desempilhamento
9        {
10          throwException();
11       } // fim do try
12       catch ( Exception exception ) // exceção lançada em throwException
13       {
14          System.err.println( "Exception handled in main" );
15       } // fim do catch
16    } // fim de main
17
```

Chama um método que talvez lance uma exceção

Captura a exceção e exibe uma mensagem

Figura 11.6 | Desempilhamento. (Parte 1 de 2.)


```
18       // throwException lança exceção que não é capturada nesse método
19       public static void throwException()throws Exception
20       {
21          try // lança uma exceção e a captura em main
22          {
23             System.out.println( "Method throwException" );
24             throw new Exception(); // gera a exceção
25          } // fim do try
26          catch ( RuntimeException runtimeException ) // captura o tipo correto
27          {
28             System.err.println(
29                "Exception handled in method throwException" );
30          } // fim do catch
31          finally // o bloco finally sempre executa
32          {
33             System.err.println( "Finally is always executed" );
34          } // fim de finally
35       } // fim do método throwException
36    } // fim da classe UsingExceptions
```

Esse método talvez lance uma `Exception` (esse é um tipo *verificado*)

Lança uma nova `Exception` que não é capturada por um handler de exceção nesse escopo do método

O bloco `finally` executa antes do método terminar (desempilhamento) e a exceção é retornada ao chamador

```
Method throwException
Finally is always executed
Exception handled in main
```

Figura 11.6 | Desempilhamento. (Parte 2 de 2.)

11.9 printStackTrace, getStackTrace e getMessage

- O método `Throwable` **printStackTrace** gera a saída do rastreamento de pilha para o fluxo de erro padrão.
  É útil no processo de teste e depuração.
- O método `Throwable` **getStackTrace** recupera as informações sobre o rastreamento de pilha.
- O método `Throwable` **getMessage** retorna a string descritiva armazenada em uma exceção.
- Para enviar as informações de rastreamento de pilha para outros fluxos que não o fluxo de erro padrão:
  Utilize as informações retornadas de `getStackTrace` e as envie para outro fluxo.
  Utilize uma das versões sobrecarregadas do método `printStackTrace`.

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 11.7

Uma exceção que não é capturada em um aplicativo faz com que o handler de exceção padrão do Java execute. Isso exibe o nome da exceção, uma mensagem descritiva que indica o problema que ocorreu e um completo rastreamento da pilha de execução. Em um aplicativo com uma única thread de execução, o aplicativo termina. Em um aplicativo com múltiplas threads, a thread que causou a exceção termina.

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 11.8

O método `Throwable toString` *(herdado por todas as subclasses* `Throwable`*) retorna uma string contendo o nome da classe da exceção e uma mensagem descritiva.*

```
1  // Figura 11.7: UsingExceptions.java
2  // Métodos Throwable getMessage, getStackTrace e printStackTrace.
3
4  public class UsingExceptions
5  {
6     public static void main( String[] args )
7     {
8        try
9        {
10          method1(); // chama method1
11       } // fim do try
12       catch ( Exception exception ) // captura a exceção lançada em method1
13       {
14          System.err.printf( "%s\n\n", exception.getMessage() );
15          exception.printStackTrace(); // imprime rastreamento da pilha de exceções
16
17          // obtém informações de rastreamento da pilha
18          StackTraceElement[] traceElements = exception.getStackTrace();
19
20          System.out.println( "\nStack trace from getStackTrace:" );
21          System.out.println( "Class\t\tFile\t\t\tLine\tMethod" );
22
```

Inicia a cadeia de chamada que levará a uma exceção no programa

Nenhum dos outros métodos captura a exceção; assim a pilha é desempilhada e a exceção é capturada aqui

Obtém um array de StackTraceElements

Figura 11.7 | Métodos Throwable getMessage, getStackTrace e printStackTrace. (Parte 1 de 3.)


```
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() );
28               System.out.printf( "%s\t", element.getLineNumber() );
29               System.out.printf( "%s\n", element.getMethodName() );
30            } // for final
31         } // fim do catch
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      } // fim do método method1
39
40      // chama method3; lança exceções de volta para method1
41      public static void method2()throws Exception
42      {
43         method3();
44      } // fim do método method2
```

Os métodos `StackTraceElement` retorna o nome da classe, o nome do arquivo, o número da linha e o nome do método do frame de uma pilha em particular

Esse método talvez lance uma `Exception` (esse é um tipo *verificado*)

Continua a cadeia de chamada a `method2`

Esse método talvez lance uma `Exception` (esse é um tipo *verificado*)

Continua a cadeia de chamada a `method3`

Figura 11.7 | Métodos `Throwable` `getMessage`, `getStackTrace` e `printStackTrace`. (Parte 2 de 3.)


```
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       } // fim do método method3
51    } // fim da classe UsingExceptions
```

Esse método talvez lance uma `Exception` (esse é um tipo *verificado*)

Lança uma nova `Exception` e inicia o desempilhamento

```
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
```

Mostra apenas a mensagem de erro que estava armazenada em um objeto `Exception`

Mostra a mensagem de erro completa e o rastreamento de pilha

Mostra as informações do rastreamento de pilha obtidas de `StackTraceElements`

Figura 11.7 | Métodos `Throwable` `getMessage`, `getStackTrace` e `printStackTrace`. (Parte 3 de 3.)

- O método `getStackTrace` de uma exceção obtém informações sobre o rastreamento de pilha como um array de objetos **StackTraceElement.**

  Os métodos **getClassName, getFileName, getLineNumber** e **getMethodName** de `StackTraceElement` obtêm o nome da classe, nome do arquivo, número da linha e nome do método, respectivamente, desse `StackTraceElement`.
- Todo `StackTraceElement` representa uma chamada de método na pilha de chamadas de método.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 11.11

Nunca ignore uma exceção que você captura. Pelo menos utilize `printStackTrace` *para gerar saída de uma mensagem de erro. Isso informará os usuários de que existe um problema, assim eles podem adotar as ações apropriadas.*

11.10 Exceções encadeadas

- Às vezes, um método responde a uma exceção lançando um tipo diferente de exceção específico do aplicativo em uso.
- Se um bloco `catch` lançar uma nova exceção, as informações da exceção original e do rastreamento de pilha serão perdidas.
- As primeiras versões do Java não forneciam nenhum mecanismo para empacotar as informações da exceção original com as informações da nova exceção.

  Isso torna a depuração desses problemas particularmente difícil.
- **Exceções encadeadas** permitem a um objeto de exceção manter informações completas sobre o rastreamento de pilha a partir da exceção original.


```
1  // Figura 11.8: UsingChainedExceptions.java
2  // Exceções encadeadas.
3
4  public class UsingChainedExceptions
5  {
6     public static void main( String[] args )
7     {
8        try
9        {
10          method1(); // chama method1
11       } // fim do try
12       catch ( Exception exception ) // exceções lançadas de method1
13       {
14          exception.printStackTrace();
15       } // fim do catch
16    } // fim de main
17
```

Captura a exceção encadeada e exibe o rastreamento de pilha

Figura 11.8 | Exceções encadeadas. (Parte 1 de 3.)


```
18      // chama method2 ; lança exceções para main
19      public static void method1()throws Exception
20      {
21         try
22         {
23            method2(); // chama method2
24         } // fim do try
25         catch ( Exception exception ) // exceção lançada de method2
26         {
27          throw new Exception( "Exception thrown in method1", exception );
28         } // fim do catch
29      } // fim do método method1
30
31      // chama method3; lança exceções de volta para method1
32      public static void method2()throws Exception
33      {
34         try
35         {
36            method3(); // chama method3
37         } // fim do try
38         catch ( Exception exception ) // exceção lançada de method3
39         {
40          throw new Exception( "Exception thrown in method2", exception );
41         } // fim do catch
42      } // fim do método method2
```

Cria uma nova exceção com uma mensagem personalizada: encadeia a exceção lançada por `method2`

Cria uma nova exceção com uma mensagem personalizada: encadeia a exceção lançada por `method3`

Figura 11.8 | Exceções encadeadas. (Parte 2 de 3.)


```
43
44       // lança Exception de volta para method2
45       public static void method3() throws Exception
46       {
47          throw new Exception( "Exception thrown in method3" );
48       } // fim do método method3
49    } // fim da classe UsingChainedExceptions
```

Exceção original

```
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
```

Observe que as exceções encadeadas aparecem nas informações do rastreamento de pilha

Figura 11.8 | Exceções encadeadas. (Parte 3 de 3.)

11.11 Declarando novos tipos de exceção

- Às vezes, é útil declarar suas próprias classes de exceção específicas dos problemas que podem ocorrer quando outro programa utiliza suas classes reutilizáveis.
- Uma nova classe de exceção deve estender uma classe de exceção existente para assegurar que a classe pode ser utilizada com o mecanismo de tratamento de exceções.

- Uma nova classe de exceção típica contém somente quatro construtores:

  um que não recebe nenhum argumento e passa uma mensagem de erro padrão `String` para o construtor de superclasse;

  um que recebe uma mensagem de erro personalizada como uma `String` e a passa para o construtor da superclasse;

  um que recebe uma mensagem de erro personalizada como uma `String` e uma `Throwable` (para encadear exceções) e a passa para o construtor da superclasse;

  e um que recebe uma `Throwable` (para encadear exceções) e a passa para o construtor da superclasse.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 11.12

Se possível, indique as exceções provenientes de seus métodos utilizando classes de exceção existentes, em vez de criar novas. A Java API contém muitas classes de exceção que podem ser adequadas ao tipo de problema que seus métodos precisam indicar.

Java™ COMO PROGRAMAR 8ª edição

Boa prática de programação 11.3

Associar cada tipo de malfuncionamento sério em tempo de execução com uma classe `Exception` *apropriadamente identificada aprimora a clareza do programa.*

Observação de engenharia de software 11.13

Ao definir seu próprio tipo de exceção, estude as classes de exceção existentes na Java API e tente estender uma classe de exceção relacionada. Por exemplo, se estiver criando uma nova classe para representar quando um método tenta uma divisão por zero, você poderia estender a classe `ArithmeticException`*, porque a divisão por zero ocorre durante a aritmética. Se as classes existentes não forem superclasses apropriadas para sua nova classe de exceção, decida se a nova classe deve ser uma classe de exceção verificada ou não verificada. A nova classe de exceção deve ser uma exceção verificada (isto é, estende* `Exception` *mas não* `RuntimeException`*) se clientes precisarem tratar a exceção. A aplicação cliente deve ser razoavelmente capaz de se recuperar de tal exceção. A nova classe de exceção deve estender* `RuntimeException` *se o código do cliente deve ser capaz de ignorar a exceção (isto é, a exceção é uma exceção não verificada).*

Java™ COMO PROGRAMAR 8ª edição

Boa prática de programação 11.4

Por convenção, todos os nomes de classe de exceções devem terminar com a palavra `Exception`.

11.12 Pré-condições e pós-condições

- Os programadores passam longos períodos de tempo mantendo e depurando código.
- Para facilitar essas tarefas e aprimorar o projeto como um todo, eles podem especificar os estados esperados antes e depois da execução de um método.
- Esses estados são chamados pré-condições e pós-condições, respectivamente.

- Uma **pré-condição** deve ser verdadeira quando um método é invocado.

  Descreve restrições nos parâmetros de método e quaisquer outras expectativas que o método tem sobre o estado atual de um programa um pouco antes de ele começar a executar.

  Se as pré-condições não forem satisfeitas, o comportamento do método será indefinido.

  Você nunca deve esperar um comportamento consistente se as pré-condições não forem satisfeitas.

- Uma **pós-condição** é verdadeira depois que o método retorna com sucesso.
  Descreve as restrições sobre o valor de retorno e quaisquer outros efeitos colaterais que o método possa apresentar.
  Ao chamar um método, você pode assumir que um método satisfaz todas as suas pós-condições.
  Se estiver escrevendo seu próprio método, documente todas as pós-condições para que outros saibam o que esperar quando chamam seu método, e você deve certificar-se de que seu método cumpre todas as pós-condições se as pré-condições forem satisfeitas.
- Quando as pré-condições ou pós-condições não são satisfeitas, os métodos costumam lançar exceções.

- Como exemplo, examine o método `String charAt`, que tem um parâmetro `int` — um índice da `String`.

  Para uma pré-condição, o método `charAt` assume que `index` é maior que ou igual a zero e menor que o comprimento da `String`.

  Se a pré-condição é atendida, a pós-condição declara que o método retornará o caractere à posição da `String` especificada pelo parâmetro `index`.

  Caso contrário, o método lança uma `IndexOutOfBoundsException`.

  Confiamos em que o método `charAt` satisfaz sua pós-condição, desde que atendamos à pré-condição.

  Não precisamos nos preocupar com os detalhes de como o método realmente recupera o caractere no índice.

- Alguns programadores declaram as pré-condições e pós-condições informalmente como parte da especificação do método geral, enquanto outros preferem uma abordagem mais formal, definindo-as explicitamente.
- Declare as pré-condições e pós-condições em um comentário antes da declaração de método.
- Declarar as pré-condições e pós-condições antes de escrever um método também ajudará a orientá-lo durante a implementação do método.

11.13 Assertivas

- Ao implementar e depurar uma classe, às vezes é útil declarar as condições que devem ser verdadeiras em um ponto específico de um método.
- **Assertivas** ajudam a assegurar a validade de um programa capturando potenciais bugs e identificando possíveis erros de lógica durante o desenvolvimento.
- As pré-condições e as pós-condições são dois tipos de assertivas.

- O Java inclui duas versões da instrução **assert** para validar assertivas programaticamente.
  - `assert` avalia uma expressão `booleana` e, se for `falsa`, lança um **AssertionError** (uma subclasse de `Error`).

    assert *expression*;

    lança um `AssertionError` se a *expressão for* `falsa`.

    assert *expression1 : expression2;*

    avalia *expressão1* e lança um `AssertionError` com *expressão2* como a mensagem de erro se a *expressão1* for `falsa`.
- Pode ser utilizada para implementar programaticamente as pré-condições e pós-condições ou verificar qualquer outro estado intermediário que ajude a assegurar que o código está funcionando corretamente.


```
1  // Figura 11.9: AssertTest.java
2  // Verificando com assert se um valor está dentro do intervalo
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       // faz a assertiva de que o valor é >= 0 e <= 10
15       assert ( number >= 0 && number <= 10 ) : "bad number: " + number;
16
17       System.out.printf( "You entered %d\n", number );
18    } // fim de main
19 } // fim da classe AssertTest
```

Exemplo mecânico que testa se a entrada de um usuário está no intervalo especificado

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

Figura 11.9 | Verificando com `assert` se um valor está dentro do intervalo. (Parte 1 de 2.)


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

Figura 11.9 | Verificando com `assert` se um valor está dentro do intervalo. (Parte 2 de 2.)

- Você utiliza assertivas principalmente para depurar e identificar erros de lógica em um aplicativo.
- Você deve ativar explicitamente as assertivas ao executar um programa.

  Reduzem o desempenho.

  São desnecessárias para o usuário do programa.
- Para permitir assertivas, utilize a opção de linha de comando `-ea` do comando `java`, como em `java -ea AssertTest`.

- Os usuários não devem encontrar quaisquer `AssertionError`s por meio da execução normal de um programa adequadamente escrito.

  Tais erros devem apenas indicar bugs na implementação.

  Por isso, você nunca deve capturar um `AssertionError`.

  Permita que o programa termine quando ocorrer erros, para que você possa ver a mensagem de erro e, então, localizar e corrigir a fonte do problema.
- Uma vez que os usuários do aplicativo podem escolher não ativar assertivas em tempo de execução

  Você *não deve* utilizar `assert` para indicar problemas de tempo de execução no código de produção.

  Você *deve* utilizar o mecanismo de exceção para esse propósito.