13

Tratamento de exceção

OBJETIVOS

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

Sumário

13.1 Introdução

- **Exceção — uma indicação de um problema que ocorre durante a execução de um programa.**
- **Tratamento de exceções — resolver exceções que poderiam ocorrer para que o programa continue ou termine elegantemente.**
- **O tratamento de exceções permite que os programadores criem programas mais robustos e tolerantes a falhas.**

Dica de prevenção de erro 13.1

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

13.1 Introdução

- **Exemplos:**
  - `ArrayIndexOutOfBoundsException` — **é feita uma tentativa de acessar um elemento depois do final de um array.**
  - `ClassCastException` — **ocorre uma tentativa de fazer uma coerção em um objeto que não tem um relacionamento *é um* com o tipo especificado no operador de coerção.**
  - `NullPointerException` — **quando uma referência `null` é utilizada onde um objeto é esperado.**

13.2 Visão geral do tratamento de exceções

- **Misturar a lógica do programa com a lógica do tratamento de erros pode tornar os programas difíceis de ler, modificar, manter e depurar.**
- **O tratamento de exceções permite aos programadores remover código de tratamento de erro da 'linha principal' de execução do programa.**
- **Aprimora a clareza.**
- **Aprimora a modificabilidade.**

Dica de desempenho 13.1

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

13.3 Exemplo: Divisão por zero sem tratamento de exceções

- **Exceção lançada — uma exceção que ocorreu.**
- **Rastreamento de pilha:**
  - **Nome da exceção em uma mensagem descritiva que indica o problema.**
  - **Pilha de chamadas de método.**
- **`ArithmeticException` — pode surgir a partir de diferentes problemas na aritmética.**
- **Ponto de lançamento — ponto inicial em que a exceção ocorre, linha superior da cadeia de chamadas.**
- **Uma `InputMismatchException` ocorre quando o método `Scanner nextInt` recebe uma string que não representa um inteiro válido.**

Resumo

DivideByZeroNoExceptionHandling.java

(1 de 2)

```
1  // Fig. 13.1: DivideByZeroNoExceptionHandling.java
2  // Um aplicativo que tenta dividir por zero.
3  import java.util.Scanner;
4
5  public class DivideByZeroNoExceptionHandling
6  {
7     // demonstra o lançamento de uma exce
8     public static int quotient( int numer
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
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    } // fim de main
26 } // fim da classe DivideByZeroNoExceptionHandling
```

Tentativa de divisão; `denominator` poderia ser zero

Lê a entrada; a exceção ocorre se a entrada não for um inteiro válido

```
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: 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)
```

Resumo

DivideByZeroNoExceptionHandling.java

(2 de 2)

13.4 Exemplo: Tratando `ArithmeticExceptions` e `InputMismatchExceptions`

- **Com o tratamento de exceções, o programa captura e trata (isto é, lida com) a exceção.**
- **O exemplo a seguir permite que o usuário tente novamente se uma entrada inválida for inserida (zero para denominador ou entrada de não-inteiro).**

Incluindo código em um bloco `try`

- **Bloco `catch` — inclui código que poderia lançar uma exceção e o código que não deve ser executado se uma exceção ocorrer.**
- **Consiste na palavra-chave `try` seguida por um bloco de código entre chaves.**

Observação de engenharia de software 13.1

Exceções podem emergir por meio de código explicitamente mencionado em um bloco `try`, por chamadas para outros métodos, por chamadas de método profundamente aninhadas iniciado pelo código em um bloco `try` ou a partir da Java Virtual Machine à medida que ela executa os bytecodes do Java.

Capturando exceções

- **Um bloco `catch`:**
  - **Captura, isto é, recebe e trata uma exceção.**
  - **Começa com a palavra-chave `catch`.**
  - **Parâmetro de exceção entre parênteses – o parâmetro de exceção identifica o tipo de exceção e permite que o bloco `catch` interaja com o objeto da exceção capturada.**
  - **Bloco do código entre chaves que executa quando uma exceção do tipo adequado ocorre.**
- **Bloco `catch` correspondente – o tipo do parâmetro de exceção corresponde exatamente ao tipo de exceção lançado ou é uma superclasse dele.**
- **Exceção não-capturada – uma exceção que ocorre para a qual não há nenhum bloco `catch` correspondente.**
  - **Faz com que o programa termine se o programa tiver somente um thread; do contrário apenas o thread atual é terminado e pode haver efeitos adversos no restante do programa.**

Erro comum de programação 13.1

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

Erro comum de programação 13.2

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

Erro comum de programação 13.3

É um erro de compilação capturar o mesmo tipo em dois blocos `catch` diferentes em uma única instrução `try`.

Modelo de terminação de tratamento de exceções

- **Quando uma exceção ocorre:**
  - **O bloco `catch` termina imediatamente.**
  - **O programa transfere o controle para o primeiro bloco `catch` correspondente.**
- **Depois de a exceção ser tratada:**
  - **Modelo de terminação do tratamento de exceções – o controle do programa não retorna ao ponto de lançamento porque o bloco `catch` expirou; o fluxo de controle prossegue para a primeira instrução depois do último bloco `catch`.**
  - **Modelo de retomada do tratamento de exceções – o controle do programa é retomado logo depois do ponto de lançamento.**
- **Instrução `catch` – consiste em um bloco `try` e correspondentes blocos `catch` e/ou `finally`.**

Erro comum de programação 13.4

Erros de lógica podem ocorrer se você assumir que, depois de uma exceção ser tratada, o controle retornará à primeira instrução depois do ponto de lançamento.

Dica de prevenção de erro 13.2

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

Boa prática de programação 13.1

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

Utilizando a cláusula `throws`

- **Cláusula `throws` — especifica as exceções que um método pode lançar.**
  - **Aparece depois da lista de parâmetros do método e antes do corpo do método.**
  - **Contém uma lista separada por vírgulas das exceções.**
  - **As exceções podem ser lançadas pelas instruções no corpo do método pelos métodos chamados no corpo do método.**
  - **As exceções podem ser dos tipos listados na cláusula `throws` ou subclasses.**

Dica de prevenção de erro 13.3

Se souber que um método pode lançar uma exceção, inclua o código de tratamento de exceções apropriado no programa para torná-lo mais robusto.

Dica de prevenção de erro 13.4

Leia a documentação on-line 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. Então forneça o tratamento para essas exceções em seu programa.

Dica de prevenção de erro 13.5

Leia a documentação on-line da API de uma classe de exceção antes de escrever o código de tratamento de exceções para esse tipo de exceção. Em geral, a documentação de uma classe de exceção contém as razões potenciais de sua ocorrência durante a execução de programa.

Resumo

DivideByZeroWithExceptionHandling.java

(1 de 3)

```
1  // Fig. 13.2: DivideByZeroWithExceptionHandling.java
2  // Um exemplo de tratamento de exceções que verifica a divisão por zero.
3  import java.util.InputMismatchException;
4  import java.util.Scanner;
5
6  public class DivideByZeroWithExceptionHandling
7  {
8     // demonstra lançamento de uma execeção quando ocorre uma divisão por zero
9     public static int quotient( int numer
10       throws ArithmeticException
11    {
12       return numerator / denominator; //
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 c                                   radas 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
```

A cláusula `throws` especifica que o quociente do método talvez lance uma `ArithmeticException`

A instrução de repetição faz um loop até o bloco `catch` completar com sucesso

O bloco `catch` tenta ler a entrada e realizar a divisão

Recupera entrada; `InputMismatchException` lançada se a entrada não tiver inteiros válidos

```
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 de loop
33         } // fim de try
34         catch ( InputMismatchException inputMismatchException )
35         {
36            System.err.printf( "\nException: %s\n",
37               inputMismatchException );
38            scanner.nextLine(); // descarta entrada para o usuário
39            System.out.println(
40               "You must enter integers. Please try again.\n" );
41         } // fim de catch
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 de catch
48      } while ( continueLoop ); // fim de do...while
49   } // fim de main
50 } // fim da classe DivideByZeroWithExceptionHandling
```

Chama o método `quotient`, que pode lançar uma `ArithmeticException`

Se alcançarmos esse ponto, a entrada era válida e o denominador era não zero, portanto o loop pode parar

Capturando uma `InputMismatchException` (usuário inseriu uma entrada de não-inteiro)

Parâmetros de exceção

Lê a entrada inválida, porém não faz nada com ela

(2 de 3)

Notifica o usuário de que ocorreu um erro

Capturando uma `ArithmeticException` (usuário inseriu zero para o denominador)

Se a linha 32 nunca for alcançada com sucesso, o loop continua e o usuário pode tentar novamente

```
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
```

Resumo

`DivideByZeroWithExceptionHandling.java`

(3 de 3)

13.5 Quando utilizar o tratamento de exceções

- **O tratamento de exceções foi concebido para processar erros síncronos.**
- **Erros síncronos – ocorrem quando uma instrução executa.**
- **Erros assíncronos – ocorrem em paralelo e independente do fluxo de controle do programa.**

Observação de engenharia de software 13.2

Incorpore sua estratégia de tratamento de exceções no sistema desde o princípio do processo de projeto. Pode ser difícil incluir um tratamento de exceções eficiente depois que um sistema foi implementado.

Observação de engenharia de software 13.3

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

Observação de engenharia de software 13.4

Evite utilizar o tratamento de exceções como uma forma alternativa de fluxo de controle. Essas 'exceções adicionais' podem interferir nas verdadeiras exceções do tipo erro.

Observação de engenharia de software 13.5

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 do aplicativo, que, então, podem processar os problemas de maneira específica ao aplicativo.

13.6 Hierarquia de exceção em Java

- Todas as exceções são herdadas direta ou indiretamente da classe `Exception`.
- As classes `Exception` formam uma hierarquia de herança que pode ser estendida.
- Classe `Throwable`, superclasse da `Exception`:
  - Somente objetos `Throwable` podem ser utilizados com o mecanismo de tratamento de exceções.
  - Tem duas subclasses: `Exception` e `Error`.
    - A classe `Exception` e suas subclasses representam situações excepcionais que podem ocorrer em um programa Java e que podem ser capturadas pelo aplicativo.
    - A classe `Error` e suas subclasses representam situações anormais que poderiam acontecer na JVM – normalmente não é possível que um programa se recupere de `Error`s.

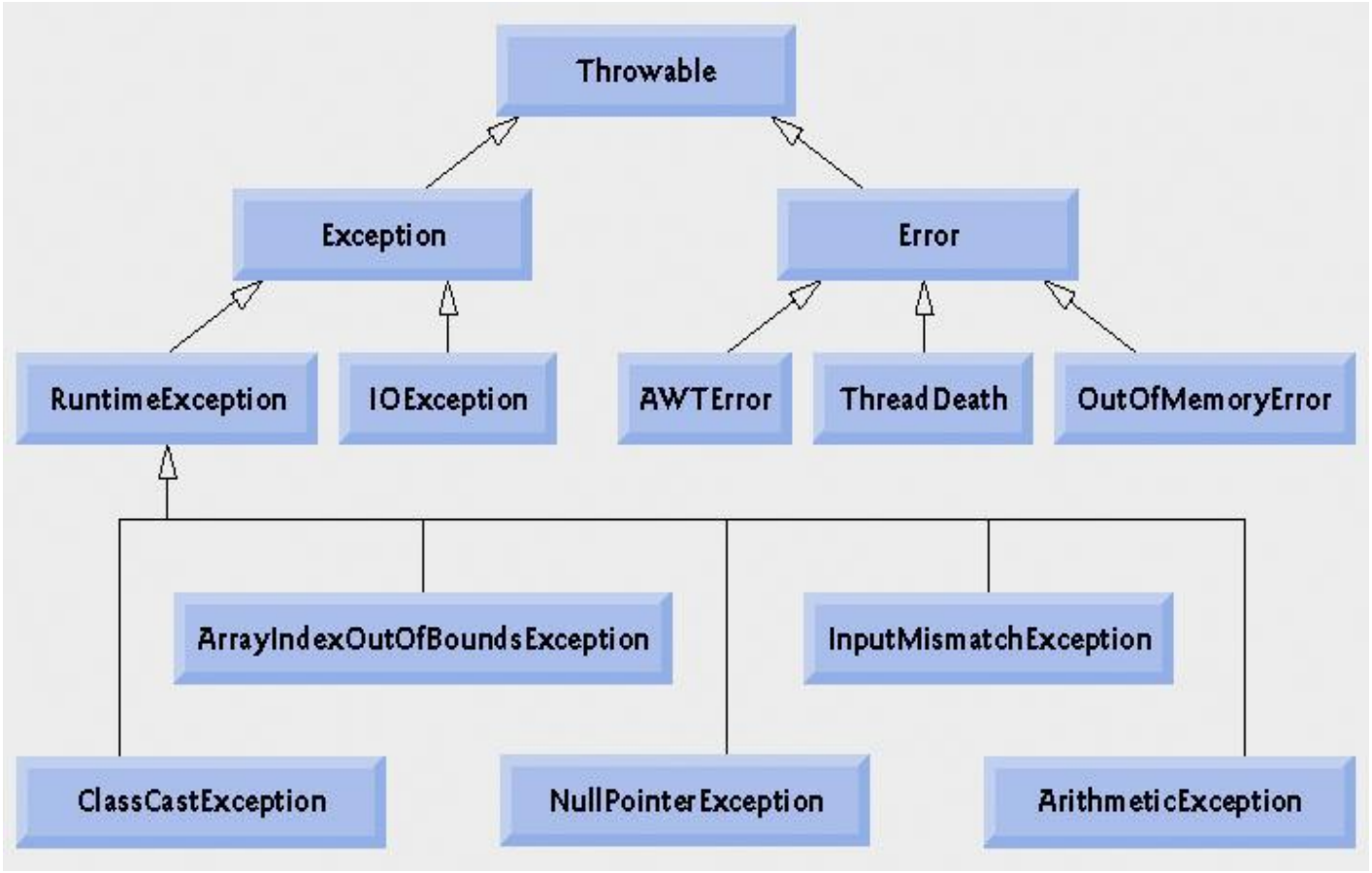

Figura 13.3 | Parte da hierarquia de herança da classe `Throwable`.

13.6 Hierarquia de exceções do Java

- **Duas categorias de exceções: verificadas e não-verificadas.**
- **Exceções verificadas:**
  - **As exceções que são herdadas da classe `Exception`, mas não de `RuntimeException`.**
  - **O compilador impõe um requisito do tipo 'capturar ou declarar'.**
  - **O compilador verifica cada chamada de método e declaração de método para determinar se o método lança (`throws`) exceções verificadas. Se lançar, o compilador assegura que a exceção verificada é capturada ou declarada em uma cláusula `throws`. Se não capturada nem declarada, ocorre um erro de compilador.**
- **Exceções não-verificadas:**
  - **Herdam da classe `RuntimeException` ou da classe `Error`.**
  - **O compilador não verifica o código para ver se a exceção foi capturada ou declarada.**
  - **Se uma exceção não-verificada ocorrer e não tiver sido capturada, o programa terminará ou executará com resultados inesperados.**
  - **Em geral, podem ser evitadas com uma codificação adequada.**

Observação de engenharia de software 13.6

Os programadores são forçados a lidar com as exceções verificadas. Isso resulta em código mais robusto do que aquele que seria criado se os programadores fossem capazes de simplesmente ignorar as exceções.

Erro comum de programação 13.5

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.

Erro comum de programação 13.6

Se um método de subclasse anula um método de superclasse, é um erro o método de subclasse listar mais exceções em sua lista `throws` do que o método anulado da superclasse anulada. Mas a cláusula `throws` de uma subclasse pode conter um subconjunto da lista `throws` de uma superclasse.

Observação de engenharia de software 13.7

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.

Observação de engenharia de software 13.8

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`) for um tipo de exceção não-verificada. Isso torna os programas mais robustos.

13.6 Hierarquia de exceções do Java (*Continuação*)

- **O bloco `catch` captura todas as exceções do seu tipo e das subclasses do seu tipo.**
- **Se houver múltiplos blocos `catch` que correspondam a um tipo particular de exceção, somente a primeiro bloco `catch` correspondente executa.**
- **Faz sentido utilizar um bloco `catch` de uma superclasse quando todos os blocos `catch` para as subclasses dessa classe realizarem a mesma funcionalidade.**

Dica de prevenção de erro 13.6

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

Erro comum de programação 13.7

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, ocorrendo, então, um erro de compilação.

13.7 Bloco `finally`

- **Programas que obtêm certos recursos devem retorná-los ao sistema explicitamente para evitar *vazamentos de recursos*.**
- **Bloco `finally`:**
  - **Consiste na palavra-chave `finally` seguida por um bloco do código entre chaves.**
  - **Opcional em uma instrução `try`.**
  - **Se presente, é colocado depois do último bloco `catch`.**
  - **Executa se uma exceção for lançada no bloco `try` correspondente ou qualquer um dos seus blocos `catch` correspondentes.**
  - **Não executará se a aplicação encerrar prematuramente em um bloco `try` via o método `System.exit`.**
  - **Em geral, contém código de liberação de recursos.**

Dica de prevenção de erro 13.7

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 não haver mais nenhuma referência a ele. Portanto, vazamentos de memória podem ocorrer, se os programadores mantiverem erroneamente referências a objetos indesejáveis.

Resumo

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

Figura 13.4 | A posição do bloco `finally` depois do último bloco `catch` em uma instrução `try`.

13.7 Bloco `finally` (*Continuação*)

- **Se nenhuma exceção ocorrer, os blocos `catch` são pulados e o controle prossegue para o bloco `finally`.**
- **Depois de o bloco `finally` executar, o controle prossegue para a primeira instrução depois do bloco `finally`.**
- **Se ocorrer uma exceção no bloco `try`, o programa pula o restante do bloco `try`. A primeira correspondência no bloco `catch` é executada e o controle prossegue para o bloco `finally`. Se ocorrer uma exceção e não houver nenhum bloco `catch` correspondente, o controle prossegue para o bloco `finally`. Depois de o bloco `finally` executar, o programa passa a exceção para o próximo bloco `try` externo.**
- **Se um bloco `catch` lançar uma exceção, o bloco `finally` ainda executará.**

Dica de desempenho 13.2

Sempre libere todos os recursos explicitamente e logo que o recurso não for mais necessário. Isso torna os recursos imediatamente disponíveis para serem reutilizados pelo seu programa ou outros programas, aprimorando assim o uso de recursos. Como é garantido que o bloco `finally` executará se ocorrer uma exceção no bloco `try` correspondente, esse bloco é um lugar ideal para liberar recursos adquiridos em um bloco `try`.

Dica de prevenção de erro 13.8

Um bloco `finally` geralmente contém código para liberar recursos adquiridos em seu bloco `try` correspondente; essa é uma maneira eficiente de eliminar vazamento de recursos. Por exemplo, o bloco `finally` deve fechar quaisquer arquivos abertos no bloco `try`.

13.7 Bloco `finally` (*Continuação*)

- **Fluxos-padrão:**
  - `System.out` – **o fluxo de saída padrão.**
  - `System.err` – **o fluxo de erros padrão.**
- `System.err` **pode ser utilizado para separar saída com erro de uma saída normal.**
- `System.err.println` **e** `System.out.println` **exibem dados para o prompt de comando por padrão.**

Lançando exceções com a instrução `throw`

- **Instrução `throw` — utilizada para lançar exceções.**
- **Os próprios programadores podem lançar exceções a partir de um método se algo der errado.**
- **A instrução `throw` consiste na palavra-chave `throw` seguida pelo objeto de exceção.**

Observação de engenharia de software 13.9

Quando `toString` for 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.

Observação de engenharia de software 13.10

Um objeto pode ser lançado sem conter informação sobre o problema que ocorreu. Nesse caso, o simples conhecimento de que uma exceção de um tipo particular ocorreu pode fornecer informações suficientes para que a pessoa responsável processe o problema corretamente.

Observação de engenharia de software 13.11

Exceções podem ser lançadas a partir de construtores. Quando um erro é detectado em um construtor, deve-se lançar uma exceção em vez de se permitir a criação de um objeto formado inadequadamente.

Relançando exceções

- As exceções são relançadas quando um bloco `catch` decide que ele não pode processar a exceção ou apenas processá-la parcialmente.
- A exceção é adiada para a instrução `try` externa.
- A exceção é relançada utilizando a palavra-chave `throw` seguida por uma referência ao objeto de exceção.

Erro comum de programação 13.8

Se uma exceção não tiver sido capturada quando o controle entrar em um bloco `finally` e o bloco `finally` lançar uma exceção que não é capturada no bloco `finally`, a primeira exceção será perdida e a exceção do bloco `finally` será retornada ao método chamador.

Dica de prevenção de erro 13.9

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`.

Erro comum de programação 13.9

Assumir que uma exceção lançada de um bloco `catch` será processada por esse bloco `catch` ou por qualquer outro bloco `catch` associado com a mesma instrução `try` pode resultar em erros de lógica.

Boa prática de programação 13.2

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 cada instrução que pode lançar uma exceção. Isso dificulta a leitura dos programas. Em vez disso, coloque um bloco `try` em torno de uma parte significativa do código, de modo que esse bloco `try` seja seguido por blocos `catch` que tratem cada possível exceção e os blocos `catch` sejam seguidos por um único bloco `finally` (se algum for necessário).

Resumo

UsingExceptions.java

(1 de 3)

```
1  // Fig. 13.5: UsingExceptions.java
2  // Demonstração do tratamento de exceções try...catch...finally
3  // mechanism.
4
5  public class UsingExceptions
6  {
7     public static void main( String args[] )
8     {
9        try
10       {
11          throwException(); // chama método throwException
12       } // fim de try
13       catch ( Exception exception )
14       {
15          System.err.println( "Exception handled in main" );
16       } // fim de catch
17
18       doesNotThrowException();
19    } // fim de main
20
```

Chama o método que lança uma exceção

Resumo

UsingExceptions.java

(2 de 3)

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

Cria uma nova `Exception` e a lança

Lança uma `Exception` previamente criada

O bloco `finally` executa mesmo que uma exceção seja relançada no bloco `catch`

Resumo

UsingExceptions.java

(3 de 3)

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

O bloco `finally` executa mesmo que nenhuma exceção seja lançada

```
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
```

13.8 Desempilhamento de pilha

- **Desempilhamento de pilha — Quando uma exceção é lançada, mas não capturada em um escopo em particular, a pilha de chamadas de método é 'desempilhada', e é feita uma tentativa de capturar (`catch`) a exceção no próximo bloco `try` externo.**
- **Quando o desempilhamento ocorre:**
  - **O método em que a exceção não foi capturada termina.**
  - **Todas as variáveis locais nesse método saem do escopo.**
  - **O controle retorna à instrução que originalmente invocou o método – se um bloco `try` incluir a chamada de método, é feita uma tentativa de capturar a exceção.**

Resumo

UsingExceptions.java

(1 de 2)

```
1  // Fig. 13.6: UsingExceptions.java
2  // Demonstração do desempilhamento.
3
4  public class UsingExceptions
5  {
6     public static void main( String args[] )
7     {
8        try // chama throwExce
9        {
10          throwException();
11       } // fim de try
12       catch ( Exception exception ) // exceção lançada em throwException
13       {
14          System.err.println( "Excepti
15       } // fim de catch
16    } // fim de main
17
```

Chama o método que lança uma exceção

Captura a exceção que poderia ocorrer no bloco `try` anterior, incluindo a chamada ao método `throwException`

Resumo

```
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 de try
26       catch ( RuntimeException runtimeEx
27       {
28          System.err.println(
29             "Exception handled in method throwException" );
30       } // fim de ca
31       finally // o b
32       {
33          System.err.
34       } // fim de finally
35    } // fim do método throwException
36 } // fim da classe UsingExceptions
```

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

O método lança uma exceção

Lança nova exceção; exceção não capturada no bloco `try` atual, portanto é tratada no bloco `try` externo

O bloco `finally` executa antes de retornar o controle ao bloco `try` externo

`UsingExceptions.java`

(2 de 2)

13.9 printStackTrace, getStackTrace e getMessage

- **Os métodos na classe `Throwable` recuperam informações adicionais sobre uma exceção.**
  - **`printStackTrace` – envia a saída do rastreamento de pilha para o fluxo de erros padrão.**
  - **`getStackTrace` – recupera informações do rastreamento de pilha como um array de objetos. `StackTraceElement`; permite processamento personalizado das informações sobre a exceção.**
  - **`getMessage` – retorna a string descritiva armazenada em uma exceção.**

Dica de prevenção de erro 13.10

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.

Dica de prevenção de erro 13.11

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

13.9 printStackTrace, getStackTrace e getMessage (*Continuação*)

- **Métodos `StackTraceElement`:**
  - `getClassName`
  - `getFileName`
  - `getLineNumber`
  - `getMethodName`
- **As informações sobre o rastreamento de pilha seguem o padrão *nomeDaClasse.nomeDoMétodo(nomeDoArquivo: númeroDaLinha).***

Resumo

UsingExceptions.java

```
1  // Fig. 13.7: UsingExceptions.java
2  // Demonstrando GetMessage e printStackTrace a partir da classe Exception.
3
4  public class UsingExceptions
5  {
6     public static void main(
7     {
8        try
9        {
10          method1(); // cham
11       } // fim de try
12       catch ( Exception exception ) // captura exceção lanç
13       {
14          System.err.printf( "%s\n\n", exception.getMessage() );
15          exception.printStackTrace(); // imprime rastreamento de pi
16
17          // obtém informações de rastreamento de pilha
18          StackTraceElement[] traceElements = exception.getStackTrace();
19
```

Chamada ao método1, método1 chama o método2, o método2 chama o método3 e o método3 lança uma nova Exception

Exibe a string descritiva da exceção lançada no método3

Recupera as informações na pilha como um array dos objetos StackTraceElement

Exibe o rastreamento de pilha da exceção lançada no método3

```
20          System.out.println( "\nStack trace from getStackTrace:" );
21          System.out.println( "Class\t\tFile\t\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() );
28             System.out.printf( "%s\t", element.getLineNumber() );
29             System.out.printf( "%s\n", element.getMethodName() );
30          } // fim de for
31       } // fim de catch
32    } // fim de main
33
34    // chama method2; lança exceções de volta para ma
35    public static void method1() throws Exception
36    {
37       method2();
38    } // fim do método method1
39
```

Recupera o nome de classe do `StackTraceElement` atual

Recupera o nome de arquivo do `StackTraceElement` atual

Recupera o número da linha do `StackTraceElement` atual

Recupera o nome de método do `StackTraceElement` atual

`método1` chama `método2`, `método2` chama `método3` e este lança uma `Exception`

```
40    // chama method3; lança exceções de volta para method1
41    public static void method2() throws Exception
42    {
43       method3();
44    } // fim do método method2
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
```

método2 chama método3, que lança uma Exception

UsingExceptions

Exceção criada e lançada

(3 de 3)

```
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
```

Observação de engenharia de software 13.12

Nunca ignore uma exceção que você captura. Pelo menos utilize `printStackTrace` para gerar a saída de uma mensagem de erro. Isso informará os usuários de que existe um problema; assim eles poderão adotar as ações adequadas.

13.10 Exceções encadeadas

- **Exceções encadeadas permitem a um objeto de exceção manter informações completas sobre o rastreamento de pilha quando uma exceção é lançada a partir de um bloco `catch`.**
- **Os usuários podem recuperar as informações sobre uma exceção original.**
- **O rastreamento de pilha proveniente de uma exceção encadeada exibe quantas exceções encadeadas restam.**

Resumo

UsingChainedExceptions.java

```
1  // Fig. 13.8: UsingChainedExceptions.java
2  // Demonstrando 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 de try
12       catch ( Exception exception ) // exceções lançadas de method1
13       {
14          exception.printStackTrace();
15       } // fim de catch
16    } // fim de main
17
```

Captura a exceção no method1 bem como quaisquer exceções encadeadas associadas

Resumo

```
18    // chama method2; lança exceções de volta para main
19    public static void method1() throws ExceptionW
20    {
21       try
22       {
23          method2(); // chama method2
24       } // fim de try
25       catch ( Exception exception ) // exceção lançada de method2
26       {
27          throw new Exception( "Exception thrown in method1", exception );
28       } // fim de try
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 de try
38       catch ( Exception exception ) // exceção lançada de method3
39       {
40          throw new Exception( "Exception thrown in method2", exception );
41       } // fim de catch
42    } // fim do método method2
43
```

Captura a exceção no `method2`, lança uma nova exceção a ser encadeada com exceções anteriores

`ions.java`

(2 de 3)

Captura a exceção no `method3`, lança uma nova exceção a ser encadeada com exceções anteriores

Resumo

```
44    // lança Exception novamente 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 lançada

```
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
```

(3 de 3)

13.11 Declarando novos tipos de exceção

- **Você pode declarar suas próprias classes de exceção específicas dos problemas que podem ocorrer quando um outro programa utiliza suas classes reutilizáveis.**
- **A nova classe de exceção deve estender uma classe de exceção existente.**
- **Em geral, ela contém somente dois construtores:**
  - **Um não recebe nenhum argumento, passa mensagens de exceção padrão para o construtor da superclasse.**
  - **O outro recebe uma mensagem personalizada de exceção como uma string e a passa para o construtor da superclasse.**

Observação de engenharia de software 13.13

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

Boa prática de programação 13.3

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

Observação de engenharia de software 13.14

Ao definir seu próprio tipo de exceção, estude as classes de exceção existentes na API do Java 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. (C*ontinua*…)

Observação de engenharia de software 13.14 (*Continuação*)

A nova classe de exceção deve ser uma exceção verificada (isto é, estender `Exception`, mas não `RuntimeException`) se possíveis 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 de cliente for capaz de ignorar a exceção (isto é, se a exceção for uma exceção não-verificada).

Boa prática de programação 13.4

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

13.12 Precondições e pós-condições

- **As precondições e pós-condições são os estados antes e depois da execução de um método.**
- **São utilizadas para facilitar a depuração e melhorar o projeto.**
- **Você deve declarar as precondições e as pós-condições em um comentário antes da declaração de método.**

13.12 Precondições e pós-condições (*Continuação*)

- **Precondições**
  - **A condição que deve ser verdadeira quando o método é invocado.**
  - **Descrevem parâmetros de método e quaisquer outras expectativas que o método tenha sobre o estado atual de um programa.**
  - **Se as precondições não forem satisfeitas, o comportamento do método será indefinido.**
- **Pós-condições**
  - **A condição que é verdadeira depois de o método retornar com sucesso.**
  - **Descrevem 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.**

13.13 Assertivas

- **Assertivas são condições que devem ser verdadeiras em um ponto particular em um método.**
- **Ajudam a assegurar a validade de um programa capturando potenciais bugs.**
- **As precondições e as pós-condições são dois tipos de assertivas.**
- **As assertivas podem ser declaradas como comentários ou podem ser validadas programaticamente utilizando a instrução `assert`.**

13.13 Assertivas (*Continuação*)

- **Instrução `assert`:**
  - **Avalia uma expressão `boolean` e determina se ela é `true` ou `false`**
  - **Duas formas:**
    - **Expressão `assert`; — `AssertionError` é lançada se *expressão* for `false`.**
    - **`Assert` *expressão1*: *expressão2*; — `AssertionError` é lançada se *expressão1* for `false`, *expressão2* é a mensagem de erro.**
  - **Utilizada para verificar estados intermediários a fim de assegurar que o código funciona corretamente.**
  - **Utilizado para implementar as precondições e pós-condições programaticamente.**
- **Por padrão, assertivas estão desativadas.**
- **As assertivas podem ser ativadas com a opção de linha de comando `-ea`.**

Resumo

AssertTest.java

```
1  // Fig. 13.9: AssertTest.java
2  // Demonstra a instrução assert
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 =
13
14       // afirma que o valor absoluto é > = 0
15       assert ( number >= 0 && number <= 10 ) : "bad number: " + number;
16
17       System.out.printf( "You e
18    } // fim de main
19 } // fim da classe AssertTest
```

Instrução `assert`

A mensagem a ser exibida com o `AssertionError`

Se o número for menor que 0 ou maior que 10, um `AssertionError` ocorrerá

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

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