Java: como programar
Programação Orientada à Objetos

- Douglas Baptista de Godoy

/in/douglasbgodoy

github.com/douglasbgodoy

Fatec
Faculdade de Tecnologia

Etec
Escola Técnica Estadual

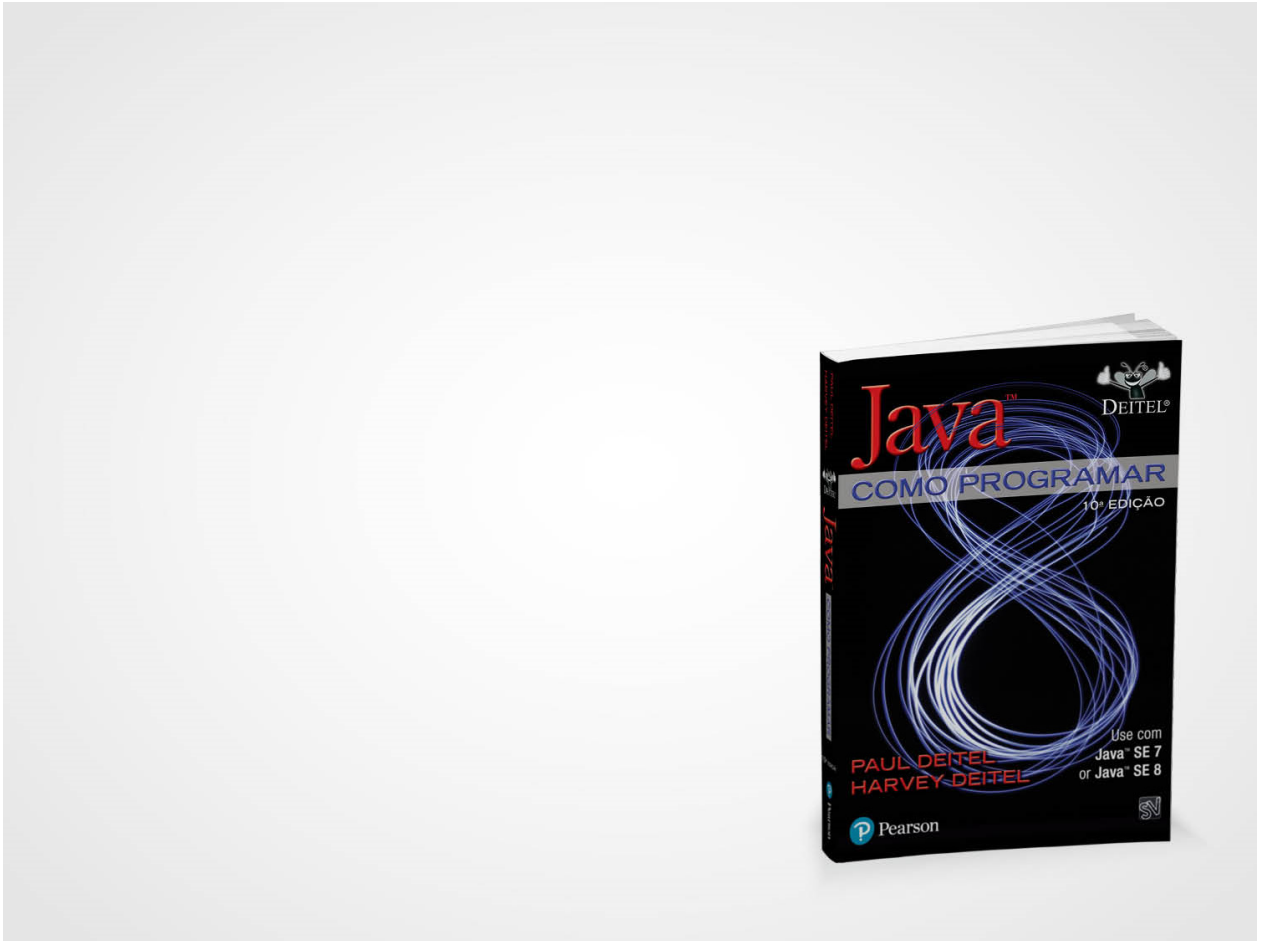
INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
São Paulo

Informação

Obs: Esta aula é baseada nos livros textos, e as transparências são baseadas nas transparências providenciadas pelos autores.

DEITEL, P. J.; DEITEL, H. M. **Java:** como programar. 10. ed. São Paulo, SP: Pearson, 2017. *E-book*. Disponível em: https://plataforma.bvirtual.com.br. Acesso em: 27 fev. 2024.

Capítulo 11:
Tratamento de exceção: um exame mais profundo



Introdução

• Uma **exceção** é uma indicação de um problema que ocorre durante a execução de um programa.

• O **tratamento de exceção** permite aos programadores criar aplicativos que podem resolver exceções.

• As exceções são **lançadas** quando um método detecta um problema e é incapaz de tratá-lo.

Exemplo: divisão por zero sem tratamento de exceção

• O **rastreamento de pilha de uma exceção** inclui o nome da exceção em uma mensagem que indica o tipo de problema que ocorreu e a pilha completa de chamadas de método no momento em que a exceção ocorreu.

• O ponto no programa em que uma exceção ocorre é chamado de **ponto de lançamento**.

Exemplo: tratando `ArithmeticExceptions` e `InputMismatchExceptions`

• Um **bloco `try`** inclui o código que talvez lance (`throw`) uma exceção e o código que não deve executar se essa exceção ocorrer.

• As exceções podem emergir por meio de código explicitamente mencionado em um bloco `try`, por chamadas para outros métodos ou até mesmo pelas chamadas de método profundamente aninhadas iniciadas pelo código no bloco `try`.

Exemplo: tratando `ArithmeticExceptions` e `InputMismatchExceptions`

- Um **bloco `catch`** inicia com a palavra-chave `catch` e um **parâmetro de exceção** seguido por um bloco de código que trata a exceção.

- Esse código executa quando o bloco `try` detecta a exceção.

Exemplo: tratando `ArithmeticExceptions` e `InputMismatchExceptions`

- Pelo menos um bloco `catch` ou `finally` deve seguir imediatamente o bloco `try`.

- Um bloco `catch` especifica entre parênteses um parâmetro de exceção identificando o tipo de exceção a tratar.

Exemplo: tratando `ArithmeticExceptions` e `InputMismatchExceptions`

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

• Uma **exceção não capturada** é uma exceção que ocorre para a qual não há nenhum bloco `catch` correspondente.

Exemplo: tratando `ArithmeticExceptions` e `InputMismatchExceptions`

• Uma exceção não capturada fará com que um programa termine antes da hora se esse programa contiver somente uma `thread`.

• Caso contrário, somente a `thread` em que a exceção ocorreu terminará. O restante do programa será executado, mas possivelmente com resultados adversos.

Exemplo: tratando `ArithmeticExceptions` e `InputMismatchExceptions`

• **Multi-`catch`** permite capturar vários tipos de exceção em uma única rotina de tratamento `catch` e realizar a mesma tarefa para cada tipo de exceção. A sintaxe para uma multi-`catch` é:

```
catch (Tipo1 | Tipo2 | Tipo3 e)
```

Exemplo: tratando `ArithmeticExceptions` e `InputMismatchExceptions`

- Cada tipo de exceção é separado do seguinte por uma barra vertical (`|`).

- Se ocorrer uma exceção em um bloco `try`, o bloco `try` termina imediatamente e o programa transfere o controle ao primeiro bloco `catch` com um tipo de parâmetro que corresponde ao tipo de exceção lançada.

Exemplo: tratando `ArithmeticExceptions` e `InputMismatchExceptions`

• Depois que uma exceção é tratada, o controle de programa não retorna ao ponto de lançamento, porque o bloco `try` expirou. Isso é conhecido como **modelo de terminação do tratamento de exceção**.

• Se houver múltiplos blocos `catch` correspondentes quando uma exceção ocorrer, somente o primeiro é executado.

Exemplo: tratando `ArithmeticExceptions` e `InputMismatchExceptions`

• A **cláusula `throws`** especifica uma lista separada por vírgula das exceções que o método pode lançar, e aparece após a lista de parâmetros do método e antes do corpo do método.

Quando utilizar o tratamento de exceção

• O tratamento de exceção processa **erros síncronos**, que ocorrem quando uma instrução é executada.

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

Hierarquia de exceção Java

• Todas as classes de exceção do Java herdam direta ou indiretamente da classe **`Exception`**.

• Programadores podem estender a hierarquia de exceções Java com suas próprias classes de exceção.

Hierarquia de exceção Java

- A classe **`Throwable`** é a superclasse de classe `Exception` e, portanto, também é a superclasse de todas as exceções. Somente objetos `Throwable` podem ser utilizados com o mecanismo de tratamento de exceção.

- A classe `Throwable` tem duas subclasses: `Exception` e `Error`.

Hierarquia de exceção Java

• A classe `Exception` e suas subclasses representam problemas que poderiam ocorrer em um programa Java e ser capturados pelo aplicativo.

• A classe `Error` e suas subclasses representam problemas que poderiam acontecer no sistema de tempo de execução do Java. `Errors` raramente acontecem e, em geral, não devem ser capturados por um aplicativo.

Hierarquia de exceção Java

• O Java distingue duas categorias de exceção: **verificadas** e **não verificadas**.

• O compilador Java não verifica se uma exceção não verificada é capturada ou declarada. Em geral, pode-se impedir a ocorrência de exceções não verificadas com codificação adequada.

Hierarquia de exceção Java

- Subclasses de **`RuntimeException`** representam exceções não verificadas.

Todos os tipos de exceção que herdam da classe `Exception`, mas não da `RuntimeException`, são exceções verificadas.

Hierarquia de exceção Java

• Se um bloco `catch` é escrito para capturar objetos de exceção de um tipo de superclasse, ele também pode capturar todos os objetos das subclasses dessa classe. Isso permite processamento polimórfico de exceções relacionadas.

Bloco `finally`

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

- O **código de liberação de recurso** é geralmente colocado em um bloco `finally`.

Bloco `finally`

• O bloco `finally` é opcional. Se estiver presente, ele é colocado depois do último bloco `catch`.

• O bloco `finally` executará se uma exceção for lançada no bloco `try` correspondente ou em qualquer um de seus blocos `catch` correspondentes.

Bloco `finally`

- Se uma exceção não puder ser capturada por uma das rotinas de tratamento `try` associadas ao bloco `catch`, o controle passa para o bloco `finally`.

- Então, a exceção é passada para o próximo bloco `try` externo.

Bloco `finally`

- Se um bloco `catch` lançar uma exceção, o bloco `finally` ainda executará.

Então, a exceção é passada para o próximo bloco `try` externo.

- A instrução `throw` pode lançar qualquer objeto `Throwable`.

Bloco `finally`

- As exceções são **relançadas** quando um bloco `catch`, ao receber uma exceção, decide que não pode processar essa exceção ou que só pode processá-la parcialmente.

Bloco `finally`

• **Relançar uma exceção** adia o tratamento de exceção (ou talvez uma parte dele) para outro bloco `catch`.

• Quando ocorre um relançamento, o próximo bloco `try` circundante detecta a exceção relançada e os blocos `catch` desse bloco `try` tentam tratá-la.

Liberando a pilha e obtendo informações de um objeto de exceção

• Quando uma exceção é lançada mas não é capturada em um escopo particular, a pilha de chamadas de método é **desempilhada** e uma tentativa de capturar (`catch`) a exceção é feita na próxima instrução `try` externa.

Liberando a pilha e obtendo informações de um objeto de exceção

- A classe **`Throwable`** oferece um método `printStackTrace` que imprime a pilha de chamadas de método. Frequentemente, isso é útil no processo de teste e depuração.

Liberando a pilha e obtendo informações de um objeto de exceção

• A classe `Throwable` também fornece um método `getStackTrace` que obtém as mesmas informações de rastreamento de pilha que são impressas por `printStackTrace`.

• O método **`getMessage`** da classe `Throwable` retorna a `string` descritiva armazenada em uma exceção.

Liberando a pilha e obtendo informações de um objeto de exceção

• O método **`getStackTrace`** obtém as informações de rastreamento de pilha como um array de objetos `StackTraceElement`. Todo `StackTraceElement` representa uma chamada de método na pilha de chamadas de método.

Liberando a pilha e obtendo informações de um objeto de exceção

- Os métodos **`StackTraceElement getClassName`**, **`getFileName`**, **`getLineNumber`** e **`getMethodName`** obtêm o nome de classe, o nome de arquivo, o número da linha e o nome do método, respectivamente.

Exceções encadeadas

• As **exceções encadeadas** permitem que um objeto de exceção mantenha as informações do rastreamento de pilha completo, incluindo as informações sobre exceções anteriores que causaram a exceção atual.

Declarando novos tipos de exceção

• 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 exceção.

Pré-condições e pós-condições

- A **pré-condição** de um método deve ser verdadeira quando o método é chamado.

- A **pós-condição** de um método é verdadeira após o método retornar com sucesso.

- Ao projetar seus próprios métodos, você deve declarar as pré-condições e pós-condições em um comentário antes da declaração de método.

Assertivas

- **Assertivas** ajudam a capturar potenciais bugs e identificar possíveis erros de lógica.

- A instrução `assert` permite validar as afirmações de forma programática.

- Para permitir assertivas em tempo de execução, utilize a switch `-ea` ao executar o comando `java`.

`try` com recursos: desalocação automática de recursos

• A instrução **`try` com recursos** simplifica escrever código em que você obtêm um recurso, usa-o em um bloco `try` e libera o recurso em um bloco `finally` correspondente.

• Em vez disso, coloque o recurso entre parênteses após a palavra-chave `try` e use o recurso no bloco `try`; então, a instrução chama implicitamente o método `close` do recurso no final do bloco `try`.

`try` com recursos: desalocação automática de recursos

• Cada recurso deve ser um objeto de uma classe que implementa a interface **`AutoCloseable`** — essa classe tem um método `close`.

• Você pode atribuir vários recursos nos parênteses depois de `try` separando-os com um ponto e vírgula (`;`).

Referências Bibliográficas

- DEITEL, P. J.; DEITEL, H. M. **Java:** como programar. 10. ed. São Paulo, SP: Pearson, 2017. .