nemo

ontology & conceptual modeling research group

Desenvolvimento OO com Java

9 – Exceções

Vítor E. Silva Souza

(vitorsouza@inf.ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo

Objetivos da parte 9

- Apresentar o mecanismo de tratamento de exceções de Java;
- Explicar os diferentes tipos de exceções, como lançá-las, capturá-las e tratá-las;
- Mostrar como criar suas próprias exceções.

Erros

- Quando desenvolvemos software, estamos sujeitos a muitos erros;
- Muitos erros podem ser identificados no momento da compilação:
  - Sintaxe incorreta;
  - Identificador (variável, método, etc.) desconhecido;
  - Classe não encontrada;
  - Etc.

Erros

- Porém, alguns erros ocorrem somente durante a execução;
- Podem ser bugs:
  - Cálculos incorretos, trecho de código não implementado, etc.;
  - Devem ser corrigidos na depuração.
- Podem ser condições excepcionais:
  - Falha no sistema de arquivos, entrada de dados inválida, etc.;
  - Devem ser tratados pelo próprio programa.

Exceções

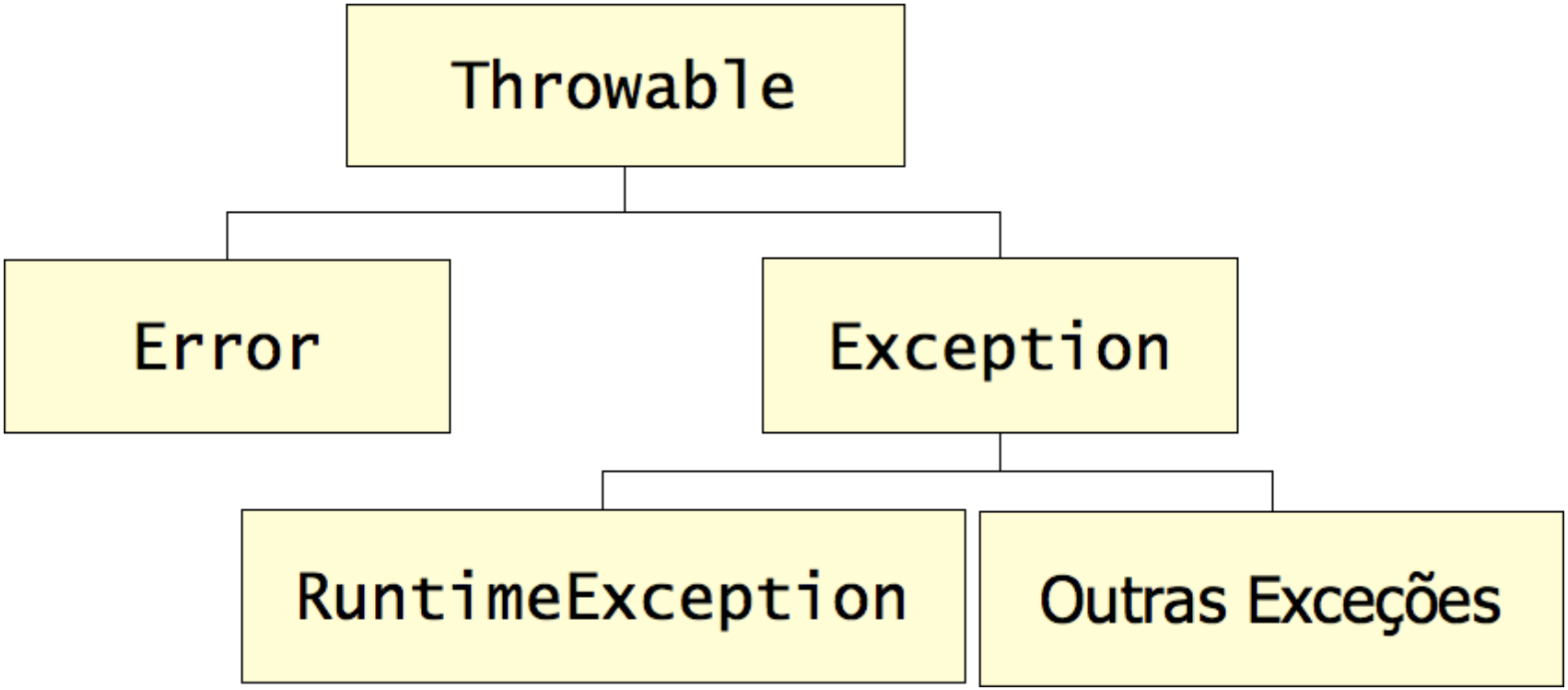
- Condição provocada por uma situação excepcional que requer uma ação específica e imediata;
- Desvia o fluxo de controle do programa para um código de tratamento;
- Podem ser causadas por diversas condições:
  - Erros sérios de hardware;
  - Erros simples de programação;
  - Condições esperadas (não são erros).

Mecanismo de exceções

- Uma LP pode ou não oferecer um mecanismo específico para tratar exceções;
- Vantagens:
  - Maior confiabilidade;
  - Maior legibilidade e redigibilidade.
- Java oferece um mecanismo de exceções.

Tipos de exceção

- Exceções, como (quase) tudo em Java, são objetos;
- Porém são objetos especiais: podem ser manipulados pelo mecanismo de exceções.



java.lang.Throwable

- Ancestral de todas as classes que recebem tratamento do mecanismo de exceções;
- Principais métodos:
  - `void printStackTrace()`: lista a sequência de métodos chamados até o ponto onde a exceção foi lançada;
  - `String getMessage()`: retorna o conteúdo de um atributo que contém uma mensagem indicadora da exceção;
  - O método `toString()` também é implementado e retorna uma descrição breve da exceção.

java.lang.Error

- Representa um problema grave, de difícil (ou impossível) recuperação;
- Exemplos:
  - `OutOfMemoryError`, `StackOverflowError`, etc.
- Geralmente causam o encerramento do programa;
- Não devem ser usadas pelos programadores.

java.lang.Exception

- Exceções que podem ser lançadas pelos métodos da API Java ou pelo seu programa;
- Devem ser tratadas;
- Em geral, representam situações inesperadas, porém contornáveis;
- O programador tem contato com esta classe e suas subclasses.

java.lang.RuntimeException

- Tipo especial de exceção;
- Não necessitam ser lançadas explicitamente pelo programa;
- Seu tratamento não é obrigatório;
- Ex.:
  - NullPointerException;
  - IndexOutOfBoundsException;
  - etc.

Exception x RuntimeException

- RuntimeException:
  - Dão menos trabalho para codificar;
  - Geralmente representam bugs;
  - O código deve ser consertado para que ela não mais ocorra.
- Exception:
  - Aumentam a confiabilidade do código;
  - Geralmente representam situações esperadas;
  - Em seu tratamento, a situação deve ser contornada e o programa continua a funcionar.

Captura e tratamento de exceções

- São obrigatórios para exceções não-*runtime*;
- Podem ser feitos para exceções *runtime*;
- Códigos que podem lançar exceções são colocados em blocos supervisionados;
- Tratadores de exceção são dispostos após o bloco, especificando quais exceções são esperadas;
- Esquema conhecido como blocos `try` – `catch`.

Blocos try – catch

```
try {
  // Código que pode lançar exceções...
}
catch (ExcecaoA e) {
  // Tratamento da exceção A,
  // ou qualquer subclasse de ExcecaoA.
  // e = instância da classe de exceção.
}
catch (ExcecaoB e) {
  // Tratamento da exceção B.
}
catch (Exception e) {
  // Tratamento de qualquer exceção.
}
```

Multi-catch (Java 7)

- Se o tratamento de duas exceções diferentes for o mesmo, há repetição de código:

```
// Imagine um método para abrir uma conexão com o BD.
Connection conn = null;
try {
  Class.forName(driver);
  conn = DriverManager.getConnection(url, usu, senha);
}
catch (ClassNotFoundException ex) {
  System.out.println("Problemas ao abrir conexão...");
}
catch (SQLException ex) {
 System.out.println("Problemas ao abrir conexão...");
}
return conn;
```

Multi-catch (Java 7)

- Podemos generalizar a exceção para a superclasse comum mais próxima:

```
// Imagine um método para abrir uma conexão com o BD.
Connection conn = null;
try {
  Class.forName(driver);
  conn = DriverManager.getConnection(url, usu, senha);
}
catch (Exception ex) {
  System.out.println("Problemas ao abrir conexão...");
}
return conn;
```

- Porém isso deixa o código genérico demais...

Multi-catch (Java 7)

- A partir do Java 7, resolve-se o problema com um `catch` múltiplo (*multi-catch*):

```
// Imagine um método para abrir uma conexão com o BD.
Connection conn = null;
try {
  Class.forName(driver);
  conn = DriverManager.getConnection(url, usu, senha);
}
catch (ClassNotFoundException | SQLException ex) {
 System.out.println("Problemas ao abrir conexão...");
}
return conn;
```

Casamento sucessivo

- Quando uma exceção ocorre, procura-se um tratador adequado;
- As cláusulas `catch` são checadas em sequência;
- A primeira que servir é executada e o programa procede após o último tratador (os demais blocos `catch` não são executados);
- Portanto, cuidado com a ordem!
  - Ex.: se a captura de `Exception` fosse a primeira, as outras nunca seriam executadas.

Lançamento de exceções

- O lançamento de uma exceção pode ser implícito ou explícito;
- Implícito:
  - Erro identificado pelo Java;
  - Ex.: ponteiro nulo, índice fora dos limites, etc.
- Explícito:
  - Lançado pela API do Java ou por seu programa quando uma situação inesperada é encontrada;
  - Ex.: entrada de dados inválida, fim de arquivo, etc.

O comando throw

- Para lançar exceções explicitamente, use a palavra-chave `throw`:

```
public class Teste {
  public static void main(String[] args) {
     try {
        throw new Exception("Exceção Teste");
     }
     catch (Exception e) {
        System.out.println("Capturada: " + e);
        e.printStackTrace();
     }
  }
}
```

Propagação de exceções

- Caso um tratador adequado não seja encontrado no bloco onde a exceção foi lançada, ela é propagada para para o nível mais externo;
- A propagação continua até que algum tratador seja encontrado ou até chegar ao nível da JVM;
- O tratamento padrão da JVM é imprimir a exceção na saída padrão.

Propagação de exceções

```
public void teste(int num) {
  try { // 1
    try { // 2
      try { // 3
        switch(num) {
        case 1: throw new NumberFormatException();
        case 2: throw new EOFException();
        case 3: throw new IOException();
        default: throw new NullPointerException();
        }
        catch (EOFException e) { }
    }
    catch (IOException e) { }
  }
  catch (NumberFormatException e) { }
}
```

Propagação de exceções

- Os blocos `try` – `catch` aninhados formam diferentes níveis;
- Se uma exceção não é tratada no bloco 3, é propagada para o 2 e assim por diante;
- Exceções não-*runtime* devem ser tratadas em algum lugar do método:
  - Do contrário gera erro de compilação;
  - Ex.: trocar `IOException` por `NullPointerException` no segundo tratador.

Propagação de exceções

```
// Este código não compila:
public void teste(int num) {
  try { // 1
     try { // 2
        try { // 3
           switch(num) {
           case 1: throw new NumberFormatException();
           case 2: throw new EOFException();
           case 3: throw new IOException();
           default: throw new NullPointerException();
        }
        catch (EOFException e) { }
     }
     catch (NullPointerException e) { }
  }
  catch (NumberFormatException e) { }
}
```

Propagação para fora do método

- É possível propagar uma exceção para fora do método onde ela ocorreu;
- Uso da palavra-chave `throws`;
- A responsabilidade de tratar fica com o código que chamou este método;
- Por sua vez, pode tratá-la ou lançá-la novamente;
- Se o método `main()` lançar uma exceção, ela é capturada pela JVM e impressa na tela.

Propagação para fora do método

```
// Este código compila!
public void teste(int num) throws IOException {
  try { // 1
     try { // 2
        try { // 3
           switch(num) {
           case 1: throw new NumberFormatException();
           case 2: throw new EOFException();
           case 3: throw new IOException();
           default: throw new NullPointerException();
        }
        catch (EOFException e) { }
     }
     catch (NullPointerException e) { }
  }
  catch (NumberFormatException e) { }
}
```

Propagação para fora do método

- Todo método é obrigado a indicar em seu cabeçalho as exceções (não-*runtime*) que propaga;
- Avisa aos usuários do método quais exceções podem ocorrer e não são tratadas;
- Um método pode propagar quantas exceções quiser:

```
public void teste(int num) throws EOFException,
IOException, NumberFormatException {
  /* ... */
}
```

Propagação para fora do método

```
public class Teste {
  public void outroTeste() {
     try {
        metodoLancador();
     }
     catch (IOException e) {
        System.out.println("Veio de outro método:");
        e.printStackTrace();
     }
  }

  public void metodoLancador() throws IOException {
     throw new IOException();
  }
}
```

Relançamento de exceções

- Exceções podem ser parcialmente tratadas em um bloco e relançadas para o bloco externo:

```
public void outroTeste() throws IOException {
  try { // 1
     try { // 2
        throw new IOException();
     }
     catch (IOException e) {
        // Tratamento parcial 1...
        throw e;
     }
  }
  catch (IOException e) {
     // Tratamento parcial 2...
     throw e;    // O tratamento termina externamente.
  }
}
```

Recapitulando...

- O que já aprendemos:
  - Erros devem ser tratados pelo mecanismo de tratamento de exceções;
  - Exceções são objetos com tratamento especial que estão na hierarquia da classe `Throwable`;
  - Usamos mais `Exception` e `RuntimeException`, sendo que esta última não precisa ser declarada;
  - Exceções ocorrem em blocos `try` – `catch` e podem ser lançadas para um nível mais externo;
- Prosseguindo...
  - Como criar minhas próprias exceções?

Criação de exceções

- Além de usar as exceções da API Java, o programador pode criar suas próprias exceções;
- Basta criar uma classe que esteja na hierarquia de `Throwable` (abaixo de `Error`, `Exception` ou `RuntimeException`);
- Coloque como propriedades da exceção informações importantes do contexto no qual ela foi lançada.

Criação de exceções

```
class ExcecaoImpar extends Exception {
  private int x;
  public ExcecaoImpar() { }
  public ExcecaoImpar(String msg) {
     super(msg);
  }
  public ExcecaoImpar(int x) {
     this.x = x;
  }
  public String toString() {
     return "O número " + x + " é ímpar!";
  }
}
```

Criação de exceções

```
public class Teste {
  public static void imprimePar(int num)
                                throws ExcecaoImpar {
     if ((num % 2) == 0)
        System.out.println(num);
     else
        throw new ExcecaoImpar(num);
  }
  public static void main(String[] args) {
     try {
        imprimePar(2);
        imprimePar(3);
     } catch (ExcecaoImpar e) {
        System.out.println(e);
     }
  }
}
```

Quando criar exceções?

- Quando em seu código puder ocorrer alguma situação inesperada e não existe uma exceção pronta que represente-a especificamente;
- Exemplos:
  - Usuário digitou informação errada;
  - Encapsular um erro interno de alguma classe que você utiliza;
  - Etc.
- Muito usadas por bibliotecas de classes e frameworks.

A cláusula `finally`

- Usada quando queremos que um trecho de código seja executado independente de haver ou não exceção;
- Colocada após o último tratador;
- O bloco `finally` é sempre executado!
- Todo bloco `try` deve ter um ou mais blocos `catch` ou um bloco `finally`;
- Pode ter ambos, formando uma estrutura conhecida como `try` – `catch` – `finally`.

A cláusula finally

```
try {
  // Código que pode lançar exceções...
}
catch (ExcecaoA e) {
  // Tratamento da exceção A,
  // ou qualquer subclasse de ExcecaoA.
  // e = instância da classe de exceção.
}
catch (ExcecaoB e) {
  // Tratamento da exceção B.
}
finally {
  // Código executado ao final.
}
```

Retomada

```
class NaoPositivoException extends Exception {}
public class Retomada {
  static Scanner in = new Scanner(System.in);
  public static void main(String[] args) {
    boolean continua = true;
    while (continua) {
      continua = false;
      try {
        System.out.print("Entre um num. positivo:");
        int i = in.nextInt();
        if (i <= 0) throw new NaoPositivoException();
      } catch(NaoPositivoException e) {
        System.out.println("Tente novamente!!!");
        continua = true;
      }
    }
  }
}
```

Exceções e herança

- As exceções adicionam certa complexidade à herança devido ao mecanismo de construção e à sobrescrita de métodos;
- Construtores e exceções:
  - Construtores são obrigados a lançar exceções declaradas no construtor da superclasse;
  - Construtores podem lançar exceções que não são declaradas no construtor da superclasse.

Por que?

```
// Este código gera erro de compilação:
// Unhandled exception type Exception

class Pai {
  Pai() throws Exception { }
}

class Filho extends Pai {
  Filho() {
    // Chamada implícita à super(),
    // super() lança Exception!
  }
}
```

Exceções e herança

- Regras para sobrescrita:
  - Não é obrigatório declarar que os métodos da subclasse lançam as exceções declaradas no método da superclasse que foi sobrescrito;
  - Métodos da subclasse não podem propagar exceções que não estão declaradas no método que foi sobrescrito;
  - A exceção: podem propagar exceções que sejam subclasses de uma das exceções declaradas no método que foi sobrescrito.

Por que?

```
// Este código gera erro de compilação:
// Exception Exception is not compatible with throws
// clause in Pai.metodo2()

class Pai {
  void metodo1() throws Exception { }
  void metodo2() throws ClassNotFoundException { }
}

class Filho extends Pai {
  @Override
  void metodo1() { } // OK!
  @Override
  void metodo2() throws Exception {
     throws new CloneNotSupportedException();
  }
}
```

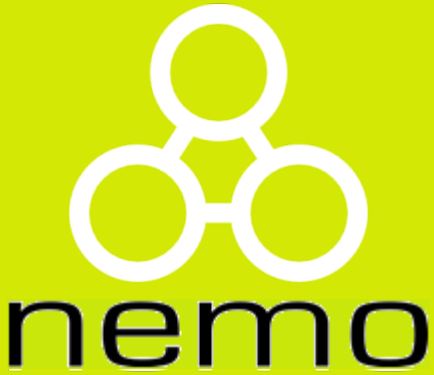
Por que?

```
public class Teste {
  public static void main(String[] args) {
    try {
      Pai p = new Filho();

      // Este método está declarado como lançando
      // ClassNotFoundException, porém a
      // implementação no filho lança outra exceção!
      p.metodo2();
    }
    catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

Exceções futuras...

- Um método pode declarar que lança uma exceção e nunca lançá-la;
- Clientes implementarão tratamento, que nunca será usado;
- Serve para deixar reservado para necessidades futuras daquele trecho de código.



http://nemo.inf.ufes.br/