Computação Orientada a Objetos

Tratamento de Exceções

Parte II

Slides baseados em:

Deitel, H.M.; Deitel P.J. Java: Como Programar, Pearson Prentice Hall, 9a Edição, 2010. Capítulo 11.

Horstmann, C. Big Java, Porto Alegre: Bookman, 2004. Capítulo 14.

Profa. Karina Valdivia Delgado
EACH-USP

Objetivo Capítulo I:

- Discutir em detalhes o mecanismo de tratamento de exceções com o intuito de que o aluno possa utilizá-lo em seus programas.
- Aprenderemos:
  - a capturar exceçoes
  - quando e onde capturar uma exceção
  - a lançar exceções
  - a diferença entre exceções verificadas e não verificadas

Revisando...

- Problemas podem ocorrer quando os programas são executados. Ex:
  - erros cometidos pelo programador (escrever numa posição de memória do vetor que não existe)
  - problemas nos ambientes externos ao da execução do programa (tentar ler um arquivo que não existe)
- Os programas devem ser capazes de lidar com possíveis problemas
- Existem duas ações importantes:
  - ponto de detecção
  - tratamento da exceção (recuperação)

Revisando:

- Tipos de Exceção
  - Erros aritméticos
  - Entrada de dados inválidos
  - Estouro de limite de array
  - Erros na manipulação de arquivos;
  - Erros na comunicação com bancos de dados;
  - ...

Revisando:

- O tratamento de exceções
  - é projetado para processar erros síncronos (que ocorrem quando uma instrução é executada).
  - não é projetado para processar problemas relacionados com eventos assíncronos (que ocorrem paralelamente com o fluxo de controle do programa). Ex: clique do mouse.

Revisando ...

```
try{
   instrução
   instrução
   instrução
   ...
}
catch (classeExceção objetoExceção) {
   código de tratamento da exceção
}
catch (classeExceção objetoExceção) {
   código de tratamento da exceção
}
catch (classeExceção objetoExceção) {
   código de tratamento da exceção
}
finally{
   código a ser executado sempre
}
```

Revisando

Sintaxe: Especificação de exceção

```
especificadorAcesso tipoRetorno nomeMetodo(tipoParametro
nomeParametro, ...) throws classeExceção, classeExceçao, ...
```

Exemplo:

```
public void read(BufferedReader in) throws IOException
```

Propósito: Declara as exceções que esse método pode lançar. Os clientes do método são informados assim de que o método pode lançar essas exceções e de que elas deverão ser tratadas.

Exemplo: incluir tratamento de exceções

```
import java.util.Scanner;

public class DivideByZeroVector
{
public static void main(String[] args) {
 Rectangle rectangle = new Rectangle();
 int a = Integer.parseInt(args[0]);
 int b = Integer.parseInt(args[1]);
 int area = rectangle.calculate(a, b);
 System.out.println("Area of a rectangle is  : " + area);
 }
} // fim da classe
```

ArrayIndexOutOfBoundsException

InputMismatchException

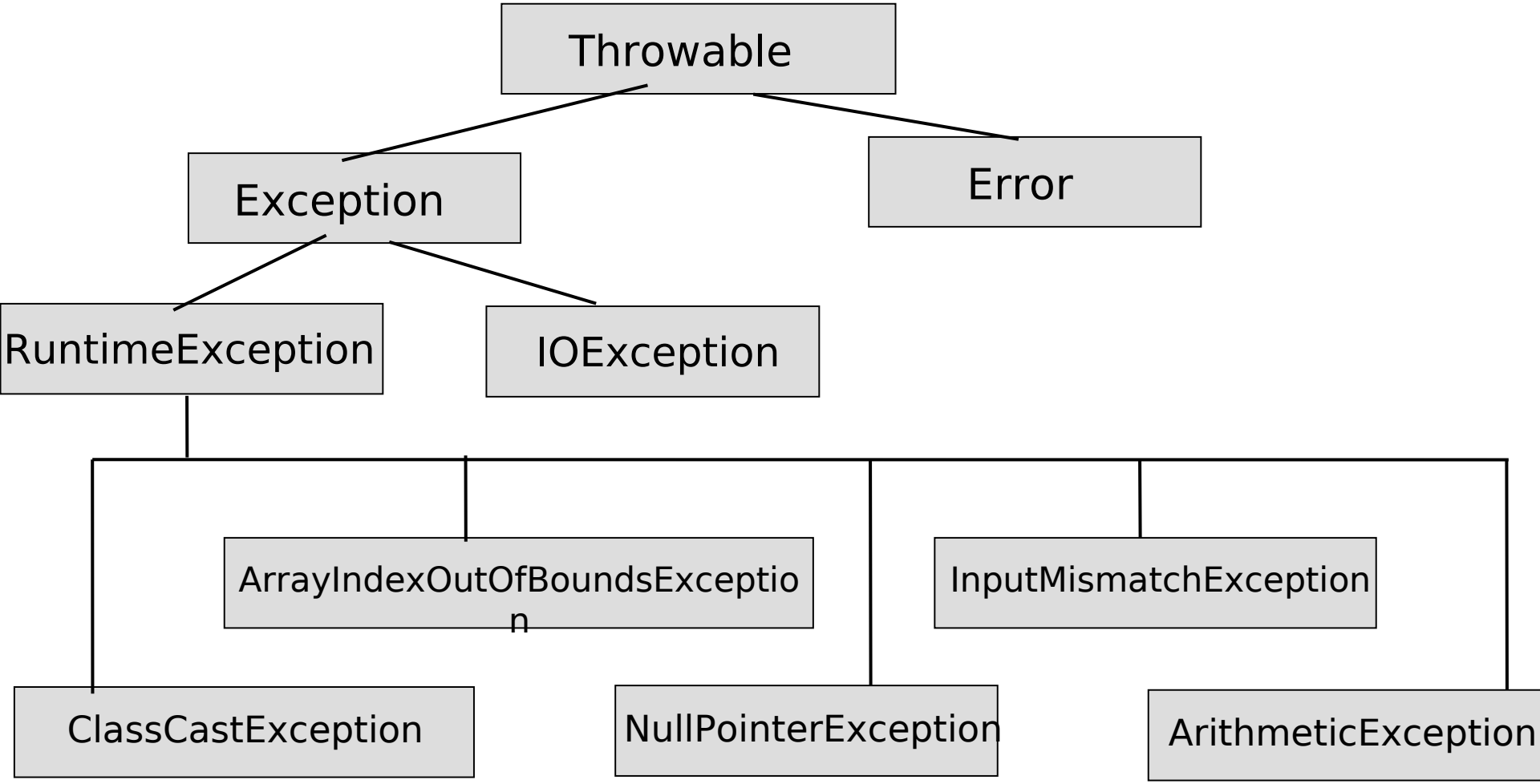
Novidade da aula de hoje:

- Hierarquia de exceções em Java
- Exceções verificadas e não verificadas
- Bloco Finally
- Lançar exceções
- Desempilhamento de pilha

Hierarquia de exceções em Java

- Todas as classes de exceção herdam, direta ou indiretamente, da classe **`Exception`**
- `Programadores podem estender essa hierarquia para criar suas próprias classes de exceção.`

Parte da hierarquia de herança da classe Throwable



Hierarquia de exceções em Java

- A hierarquia de exceções Java contem centenas de classes.
- A documentação sobre a classe **`Throwable`** pode ser encontrada em:

```
java.sun.com/j2se/1.5.0/docs/api/java/lang
  /Throwable.html
```

Hierarquia de exceções em Java

- A classe **`Exception`** e suas subclasses representam situações excepcionais que:
  - podem ocorrer e
  - que podem ser capturadas por aplicativos
- Ex:
  - subclasse **`RuntimeException`** (pacote **`java.lang`**)
  - subclasse **`IOException`** (pacote **`java.io`**)

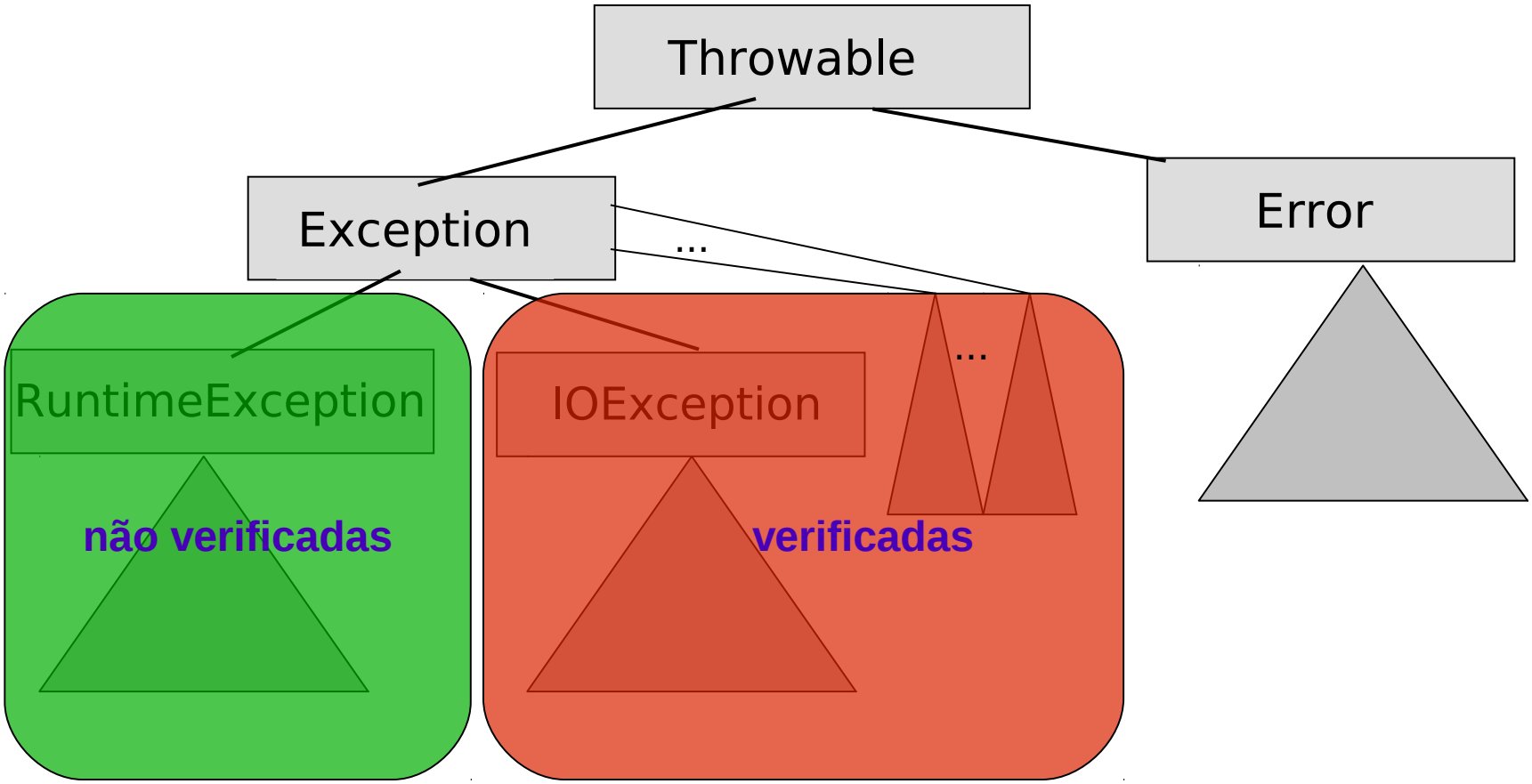
Hierarquia de exceções em Java

- A classe **`Error`** e suas subclasses (ex, **`OutOfMemoryError`**) representam situações anormais que:
  - podem acontecer na JVM,
  - acontecem raramente e
  - não são capturadas por aplicativos (não é possível que aplicativos se recuperem)

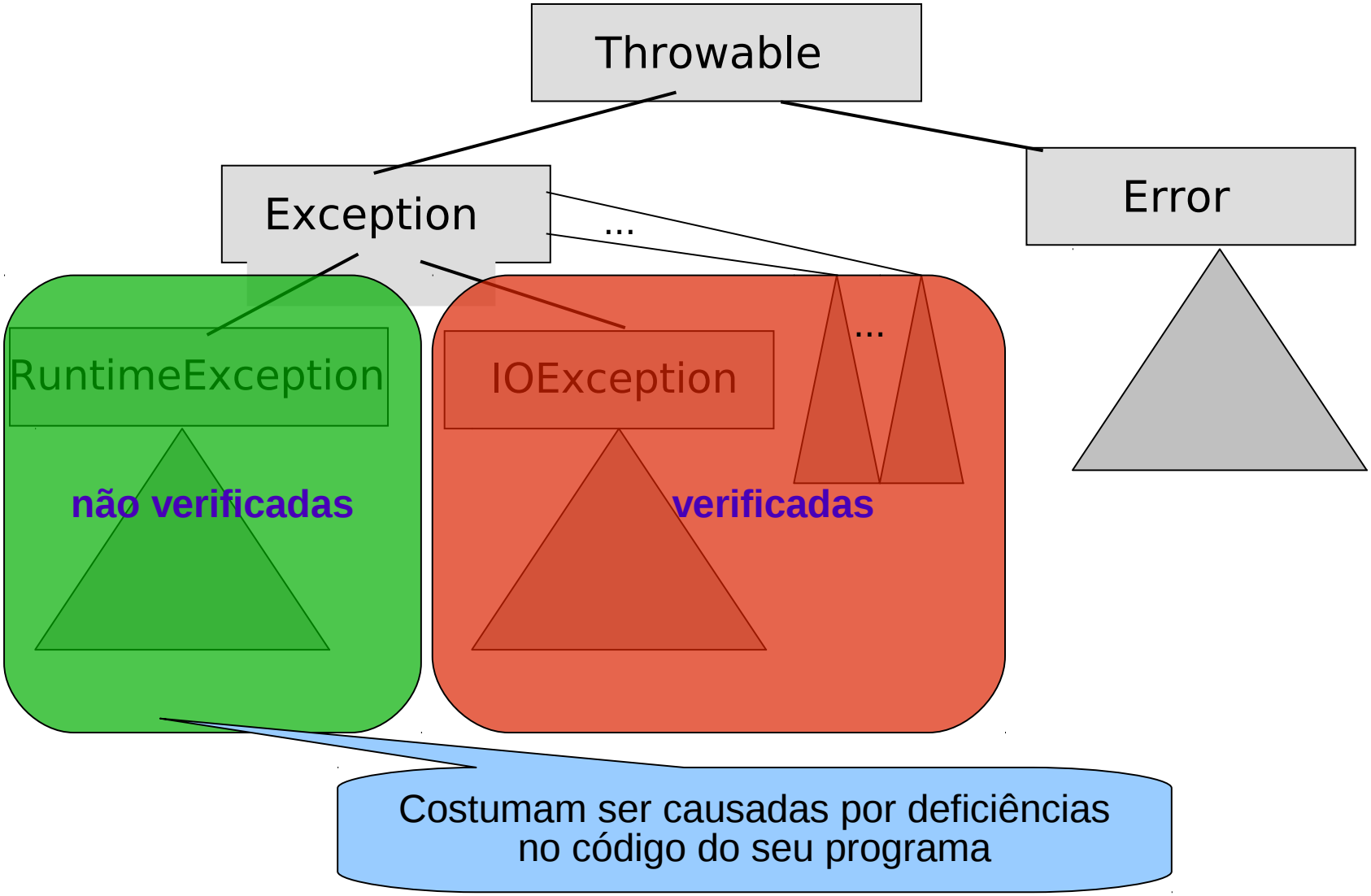
Exceções verificadas e não verificadas

- Java faz distinção entre duas categorias de exceção:
  - verificadas
  - não verificadas
- O tipo de uma exceção determina se ela é verificada ou não verificada.

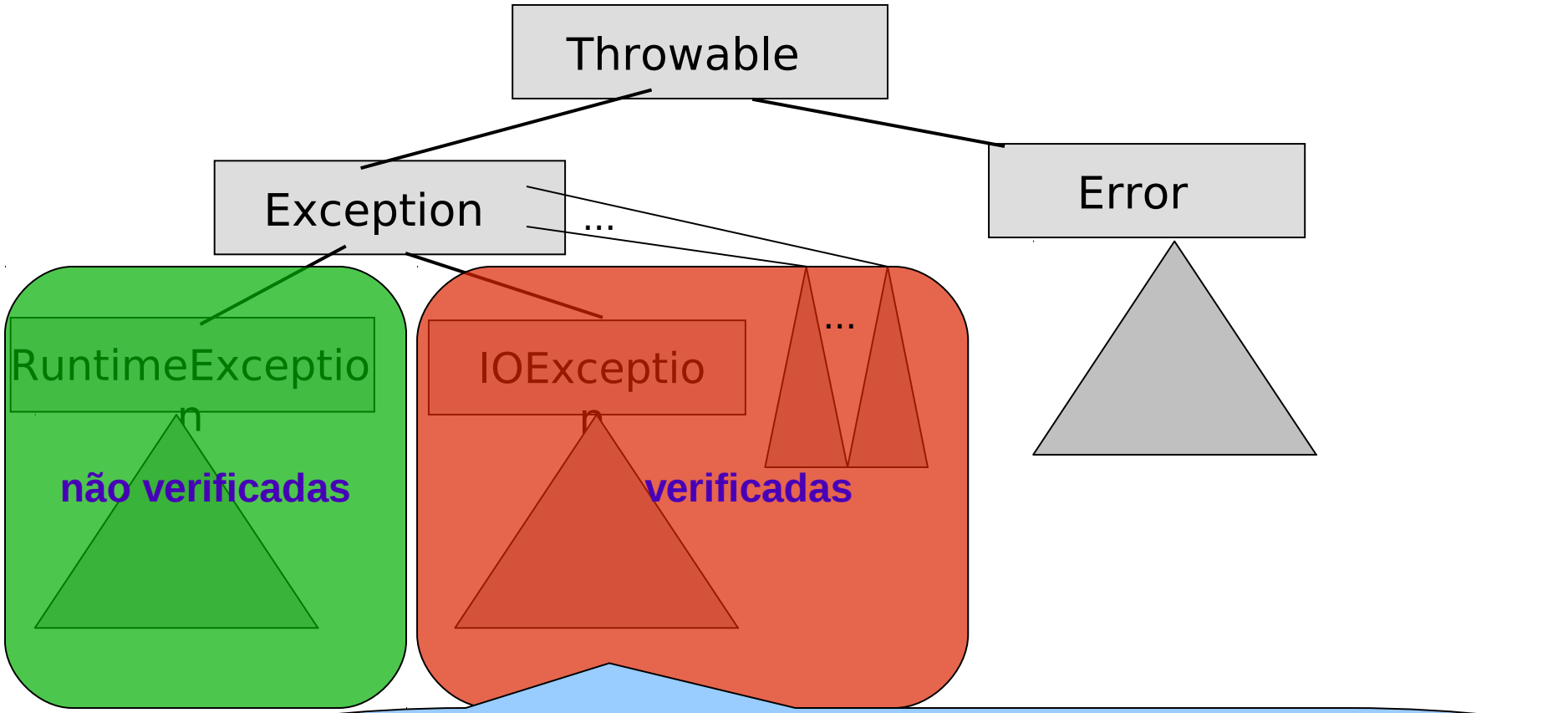
Exceções verificadas e não verificadas



Exceções verificadas e não verificadas



Exceções verificadas e não verificadas



Exceções verificadas

- Exceções são definidas como verificadas se a exceção é suficientemente importante para ser tratata
- O compilador Java impõe um requisito *catch-or-declare* (capture ou declare) às exceções verificadas.

Exceções verificadas

- O compilador verifica cada chamada de método e declaração de método para determinar se o método lança exceções verificadas.
- Se lançar, o compilador assegura que a exceção verificada seja capturada (via blocos **`try/catch`**) ou declarada em uma cláusula **`throws`**.

Exceções verificadas

- Se o requisito *catch-or-declare* não for satisfeito, o compilador emitirá uma mensagem de erro.
- Isso força os programadores a pensarem nos problemas que podem ocorrer quando um método que lança exceções verificadas for chamado.

Exceções não verificadas

- O compilador Java não verifica o código para determinar se uma exceção não verificada é capturada ou declarada.
- Não é necessário que as exceções não verificadas sejam listadas na cláusula **`throws`** de um método
  - mesmo se forem, essas exceções não precisam ser capturadas por um aplicativo.

Exemplo com tratamento de exceções

```
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling
{
   public static int quotient( int numerator, int denominator ) throws
   ArithmeticException
   {
      return numerator / denominator;
   } // fim de método quotient

continua...
```

Exemplo com tratamento de exceções

```
public static void main( String args[] )
{
    Scanner scanner = new Scanner( System.in ); // scanner para entrada
    boolean continueLoop=true;
    do{
        try{
            System.out.print( “Enter an integer numerator: " );
            int numerator = scanner.nextInt();
            System.out.print( "Enter an integer denominator: " );
            int denominator = scanner.nextInt();
            int result = quotient( numerator, denominator );
            System.out.printf("\nResult: %d / %d = %d\n", numerator,denominator,result);
            continueLoop=false;
        }
        catch ( InputMismatchException inputMismatchException )
        {
            System.err.printf( "\nException: %s\n", inputMismatchException );
            scanner.nextLine(); // descarta entrada
            System.out.println(“You must enter integers. Try again \n" );
        } // fim de catch
        catch ( ArithmeticException arithmeticException )
        {
            System.err.printf( "\nException: %s\n", arithmeticException );
            System.out.println("Zero is an invalid denominator.Try again.\n");
        } // fim de catch
    }while (continueLoop);
} // fim de main
} // fim da classe DivideByZeroNoExceptionHandling
```

Bloco `finally`

- Os programas que obtém certos tipos de recurso devem retorná-los ao sistema explicitamente para evitar os supostos vazamentos de recurso
- Exemplos de recursos que deveriam ser fechados:
  - arquivos
  - conexões com bancos de dados
  - conexões de rede

Instrução try:

```
try{
  instruções

  instruções de adquisição de recursos

  instruções

  ...

}

catch (classeExceção1 objetoExceção1) {
   código de tratamento da exceção
}

catch (classeExceção2 objetoExceção2) {
   código de tratamento da exceção
}

finally{

  instruções

  instruções de liberação de recursos

}
```

Bloco `finally`

- O bloco **`finally`** quase sempre será executado, independentemente de ter ocorrido uma exceção ou de esta ter sido tratada ou não.
- O bloco **`finally`** não será executado somente se o aplicativo fechar antes chamando o método **`System.exit()`**
  - Esse método fecha imediatamente um aplicativo.

Lançando exceções

- Os programadores podem lançar exceções utilizando a instrução **`throw`**.
- Assim como as exceções lançadas pelos métodos da API Java, isso indica para os aplicativos clientes que ocorreu um problema.
- O operando de **`throw`** pode ser de qualquer classe derivada de **`Throwable`**.

Sintaxe: Lançando uma exceção

```
throw objetoExceção;
```

Exemplos:

```
throw new IllegalArgumentException();
throw new IllegalArgumentException("Argumento errado");
```

Propósito: Lançar uma exceção e transferir o controle para o tratador de esse tipo de exceção.

Exemplo 1 lançamento exceção

```
public class LancaExcecaoPropria
{
  public static void main (String [] args)
  {
    try {
      throw new ArithmeticException("Erro de cálculo");
    }
    catch(ArithmeticException arithmeticException) {
      System.out.println("Ocorreu a excecao: " + arithmeticException);
    }
  }
}
```

Exemplo 1 lançamento exceção

```
public class LancaExcecaoPropria
{
  public static void main (String [] args)
  {
    try {
      throw new ArithmeticException("Erro de cálculo");
    }
    catch(ArithmeticException arithmeticException) {
      System.out.println("Ocorreu a excecao: " + arithmeticException);
    }
  }
}
```

Exemplo 2 lançamento exceção

```
public static void throwException()
   {
      try // lança uma exceção e imediatamente a captura
      {
         System.out.println( "Metodo throwException" );
         throw new Exception(); // gera a exceção
      } // fim de try
      catch ( Exception exception ) // captura exceção lançada em try
      {
         System.err.println("Excecao tratada no metodo throwException");
      } // fim de catch
      finally // executa independentemente do que ocorre em try...catch
      {
         System.err.println( "Finally executado em throwException" );
      } // fim de finally

   } // fim de método throwException
```

Exemplo 3 retirar dinheiro de uma conta

```
public class BankAccount
   {
      ....
      public void withdraw (double amount)
      {
         if (amount > balance) {
           IllegalArgumentException illegalArgumentException
           =new IllegalArgumentException (´´A quantia ultrapassa o saldo´´);
           throw illegalArgumentException;
         }
         balance=balance-amount;
      }
      ....
   } // fim de método throwException
```

Ao lançar uma exceção, o método atual termina imediatamente

Responsabilidade de tratamento de exceções

- Quando um método lança uma exceção, o ambiente Java tenta encontrar algum código capaz de tratá-la.
- Em alguns casos é conveniente que o próprio método que gerou a exceção faça seu tratamento.
- Em outros, é mais adequado propagá-la ao método que o chamou.

Relançando exceções

- Exceções são relançadas quando um bloco **`catch`**, ao receber uma exceção, decide que:
  - não pode processá-la ou
  - que só pode processá-la parcialmente.
- Relançar uma exceção adia o tratamento de exceções (ou parte dele) para um outro bloco **`catch`** associado com uma instrução **`try`** externa.
- Uma exceção é relançada utilizando a palavra-chave **`throw`** seguida por uma referência ao objeto que acabou de ser capturado.

Exemplo relançamento exceção

```
public static void main( String args[] )
   {
      try
      {
         throwException(); // chama método throwException
      } // fim de try
      catch ( Exception exception ) //exceção lançada por throwException
      {
         System.err.println( "Excecao tratada em main" );
      } // fim de catch

} // fim de main
```

Exemplo relançamento exceção

```
public static void throwException() throws Exception
   {
      try // lança uma exceção e imediatamente a captura
      {
         System.out.println( "Metodo throwException" );
         throw new Exception(); // lança a exceção
      } // fim de try
      catch ( Exception exception ) // captura exceção lançada em try
      {
         System.err.println("Excecao parcialmente tratada em throwException" );
         throw exception; // relança para processamento adicional

      } // fim de catch
      finally // executa independentemente do que ocorre em try...catch
      {
         System.err.println( "Finally executado em throwException" );
      } // fim de finally
   } // fim de método throwException
```

Exemplo relançamento exceção

```
public static void main( String args[] )
   {
      try
      {
         throwException(); // chama método throwException
      } // fim de try
      catch ( Exception exception ) //exceção lançada por throwException
      {
         System.err.println( "Excecao tratada em main" );
      } // fim de catch

} // fim de main
```

Exemplo relançamento exceção

```
Metodo throwException
Excecao parcialmente tratada em throwException
Finally executado em throwException
Excecao tratada em main
```

Exceções aninhadas

- A captura e tratamento de exceções pode ser aninhada em vários níveis de *try/catch*:

```
try{
   try{
      throw Exceção2
   }
   catch ( Exceção1 ){
      ...
   }
}
catch( Exceção2 ){
      ...
}
```

Desempilhamento da pilha de execução

- A pilha de execução é a lista ordenada de métodos que foram chamados até chegar ao método que gerou a exceção.
- O código para tratamento da exceção pode estar:
  - no próprio método que a provocou, ou
  - em algum método anterior na *pilha de execução*.

Desempilhamento da pilha de execução: Onde está o catch?

- O ambiente Java pesquisa a pilha de execução em busca de um tratamento adequado para a exceção que foi gerada.
  - Se encontrado, esse bloco de tratamento assume o controle do programa.
  - Senão existe nenhum tratador, o controle chega de volta até *main*() e o programa termina.

Exemplo desempilhamento

```
public static void main( String args[] )
   {
      try // chama throwException para demonstrar o desempilhamento
      {
         throwException();
      } // fim de try
      catch ( Exception exception ) // exceção lançada em throwException
      {
        System.err.println( "Exceção tratada em main" );
      } // fim de catch
   } // fim de main
```

Exemplo desempilhamento

```
// throwException lança exceção que não é capturada nesse método
   public static void throwException() throws Exception
   {
      try // lança uma exceção e a captura em main
      {
         System.out.println( "Metodo throwException" );
         throw new Exception(); // lança a exceção
      } // fim de try
      catch ( RuntimeException runtimeException ) // captura tipo incorreto
      {
         System.err.println("Excecao tratada no meto    throwException" );
      } // fim de catch
      finally // o bloco finally sempre executa
      {
         System.err.println( "Finally sempre executado"
      } // fim de finally
   } // fim de método throwException
```

Captura o tipo incorreto!!!

Exemplo desempilhamento

```
public static void main( String args[] )
   {
      try // chama throwException para demonstrar o desempilhamento
      {
         throwException();
      } // fim de try
      catch ( Exception exception ) // exceção lançada em throwException
      {
        System.err.println( "Exceção tratada em main" );
      } // fim de catch
   } // fim de main
```

Resumo:

- Requisito *catch-or-declare* (capture ou declare) para as exceções verificadas.
- Todos as subclasses de **`RuntimeException`** são exceções não verificadas.
- Instruções de liberação de recursos podem ser colocadas no bloco *finally*
- Os programadores podem lançar exceções utilizando a instrução **`throw`**.
- O código para tratamento da exceção pode estar:
  - no próprio método que a provocou, ou
  - em algum método anterior na *pilha de execução*.

Resumo:

```
try{
     código que pode gerar exceções
}
catch (classeExceção objetoExceção) {
      código de tratamento da exceção
}
finally {
  instruções
  instruções de liberação de recursos
}
```

```
  throw objetoExceção
```

Exercício 1: incluir o tratamento de exceções

```
public class BankAccount
   {
      public static void main( String args[] )
      {
          ...
          … withdraw(amount);
          ...
      }
      public void withdraw (double amount)
      {
         if (amount > balance)
         { IllegalArgumentException illegalArgumentException
           =new IllegalArgumentException (´´A quantia ultrapassa o saldo´´);
           throw illegalArgumentException;
         }
         balance=balance-amount;
      }


   } // fim de método throwException
```

Exercício 2:

Um programa lança uma exceção e o tratador de exceção apropriado começa a ser executado. O que acontece se o próprio tratador de exceção lança a mesma exceção? Pode acontecer uma recursão infinita?

Exercício 3:

Se um programa tenta capturar um tipo de exceção de superclasse antes de um tipo de exceção de subclasse, o compilador irá apontar erros? a ordem dos tratadores de exceção é importante?