

Computação Orientada a Objetos

Tratamento de Exceções – cont.

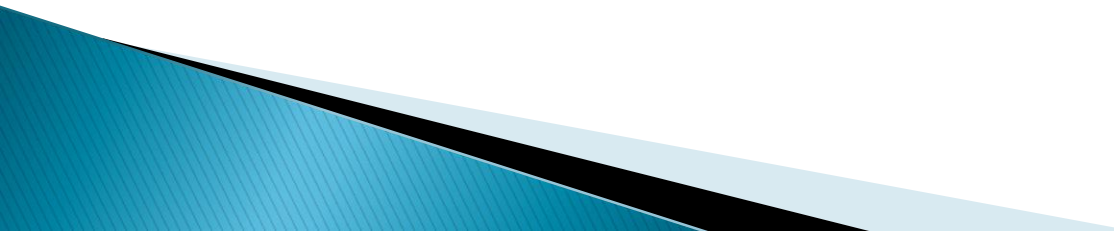
Profa. Thienne Johnson

EACH/USP



Conteúdo

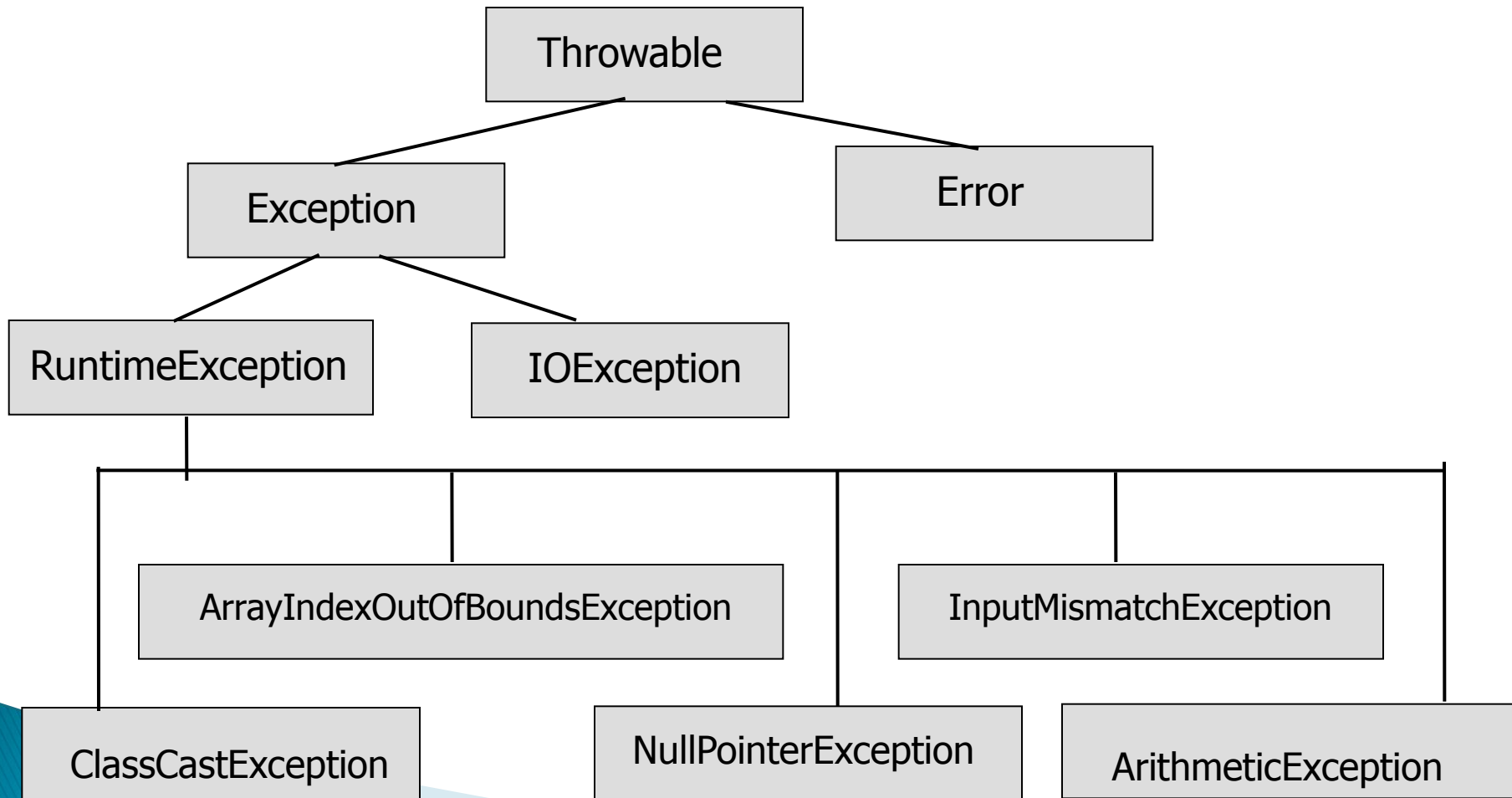
- ▶ Java, como programar
 - Deitel & Deitel

 - ▶ Capítulo 14
 - continuação
- 

Hierarquia de exceções em Java

- ▶ Todas as classes de exceção de Java herdam, direta ou indiretamente, da classe **Exception**, formando uma hierarquia de herança
- ▶ Os 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 classe **Exception** e suas subclasses representam situações excepcionais que podem ocorrer em um programa e que podem ser capturadas por um aplicativo
- ▶ 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
- ▶ Exceções **Error** acontecem raramente e não devem ser capturadas por aplicativos
 - normalmente não é possível que os aplicativos se recuperem de exceções **Error**

Hierarquia de exceções em Java

- ▶ A hierarquia de exceções Java é enorme, contendo centenas de classes
- ▶ A documentação sobre a classe **Throwable** pode ser encontrada em:

```
java.sun.com/javase/6/docs/api/java/lang/Throwable.html
```

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 verificadas

Exceções verificadas e não verificadas

- ▶ Todas as classes que herdam da classe **Exception** mas não da classe **RuntimeException** são exceções verificadas
- ▶ Ex:
 - subclasse **IOException**
- ▶ As classes que herdam da classe **Error** são consideradas não verificadas

Exceções verificadas e não verificadas

- ▶ Todos os tipos de exceção que são subclasses da classe **RuntimeException** são exceções não verificadas
- ▶ Ex:
 - subclasse **ArrayIndexOutOfBoundsException**
 - subclasse **ArithmeticException**

Exceções verificadas e 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 assegura que a exceção verificada é capturada (via blocos `try/catch`) ou declarada em uma cláusula `throws`

Exceções verificadas e não verificadas

- ▶ O compilador Java impõe um requisito *catch-or-declare* (capture ou declare) às exceções verificadas
- ▶ 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
- ▶ Isso força os programadores a pensarem nos problemas que podem ocorrer quando um método que lança exceções verificadas for chamado

Exceções verificadas e não verificadas

- ▶ Ao contrário das exceções 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

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:
 - arquivos
 - conexões com bancos de dados
 - conexões de rede

Bloco `finally`

- ▶ O bloco `finally` é opcional e consiste na palavra-chave `finally` seguida pelo código entre chaves `{ }`
- ▶ Se estiver presente, esse bloco é colocado depois do último bloco `catch`

Bloco finally

```
try
{
    statements
    resource-acquisition statements
} // end try
catch ( AKindOfException exception1 )
{
    exception-handling statements
} // end catch
...
catch ( AnotherKindOfException exception2 )
{
    exception-handling statements
} // end catch
finally
{
    statements
    resource-release statements
} // end finally
```

Fig. 11.4 | A try statement with a finally block.

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 de um bloco `try` chamando o método `System.exit`
 - Esse método fecha imediatamente um aplicativo

Lançando exceções com `throw`

- ▶ Os programadores podem lançar exceções utilizando a instrução `throw`
- ▶ A instrução `throw` é executada para sinalizar a ocorrência de uma exceção
- ▶ Assim como as exceções lançadas pelos métodos da API Java, isso indica para os aplicativos clientes que ocorreu um erro
- ▶ O operando de `throw` pode ser de qualquer classe derivada de `Throwable`

Relançando exceções

- ▶ 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
- ▶ 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: lançando exceções com *throw*

```
1 // Fig. 11.5: UsingExceptions.java
2 // try...catch...finally exception handling mechanism.
3
4 public class UsingExceptions
5 {
6     public static void main( String[] args )
7     {
8         try
9         {
10            throwException(); // call method throwException
11        } // end try
12        catch ( Exception exception ) // exception thrown by throwException
13        {
14            System.err.println( "Exception handled in main" );
15        } // end catch
16
17        doesNotThrowException();
18    } // end main
19
```

Starts a call chain in which an exception will be thrown

Starts a call chain in which no exceptions occur

Fig. 11.5 | try...catch...finally exception-handling mechanism. (Part 1 of 4.)

Exemplo: lançando exceções com *throw* (2)

```
20 // demonstrate try...catch...finally
21 public static void throwException() throws Exception
22 {
23     try // throw an exception and immediately catch it
24     {
25         System.out.println( "Method throwException" );
26         throw new Exception(); // generate exception
27     } // end try
28     catch ( Exception exception ) // catch exception thrown in try
29     {
30         System.err.println(
31             "Exception handled in method throwException" );
32         throw exception; // rethrow for further processing
33
34         // code here would not be reached; would cause compilation errors
35
36     } // end catch
37     finally // executes regardless of what occurs in try...catch
38     {
39         System.err.println( "Finally executed in throwException" );
40     } // end finally
41
42     // code here would not be reached; would cause compilation errors
43
44 }
```

This method might throw an Exception (this is a *checked* type)

Throws a new Exception that is caught at line 28 and thrown again at line 32

Rethrowing the exception means that it is not considered to have been handled

This block executes even though line 32 in the catch handler threw an exception; then the method terminates

Fig. 11.5 | try...catch...finally exception-handling mechanism. (Part 2 of 4.)

Exemplo: lançando exceções com *throw* (3)

```
45 // demonstrate finally when no exception occurs
46 public static void doesNotThrowException()
47 {
48     try // try block does not throw an exception
49     {
50         System.out.println( "Method doesNotThrowException" );
51     } // end try
52     catch ( Exception exception ) // does not execute
53     {
54         System.err.println( exception );
55     } // end catch
56     finally // executes regardless of what occurs in try...catch
57     {
58         System.err.println(
59             "Finally executed in doesNotThrowException" );
60     } // end finally
61
62     System.out.println( "End of method doesNotThrowException" );
63 } // end method doesNotThrowException
64 } // end class UsingExceptions
65
```

This method does not throw any exceptions

This try block will execute all of its statements correctly

This catch handler will be skipped; no exceptions occur

This finally block still executes

Program control continues here

Fig. 11.5 | try...catch...finally exception-handling mechanism. (Part 3 of 4.)

Exemplo: lançando exceções com *throw* (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
```

Fig. 11.5 | try...catch...finally exception-handling mechanism. (Part 4 of 4.)

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 ){...}
```


Exemplo Exceções aninhadas

```
1 // Fig. 11.6: UsingExceptions.java
2 // Stack unwinding.
3
4 public class UsingExceptions
5 {
6     public static void main( String[] args )
7     {
8         try // call throwException to demonstrate stack unwinding
9         {
10            throwException();
11        } // end try
12        catch ( Exception exception ) // exception thrown in throwException
13        {
14            System.err.println( "Exception handled in main" );
15        } // end catch
16    } // end main
17
```

Calls a method that might throw an exception

Catches the exception and displays a message

Fig. 11.6 | Stack unwinding. (Part 1 of 2.)

Exemplo Exceções aninhadas (2)

```
18 // throwException throws exception that is not caught in this method
19 public static void throwException() throws Exception
20 {
21     try // throw an exception and catch it in main
22     {
23         System.out.println( "Method throwException" );
24         throw new Exception(); // generate exception
25     } // end try
26     catch ( RuntimeException runtimeException ) // catch incorrect type
27     {
28         System.err.println(
29             "Exception handled in method throwException" );
30     } // end catch
31     finally // finally block always executes
32     {
33         System.err.println( "Finally is always executed" );
34     } // end finally
35 } // end method throwException
36 } // end class UsingExceptions
```

This method might throw an Exception (this is a *checked* type)

Throws a new Exception that is not caught by an exception handler in this method's scope

The finally block executes before the method terminates (stack unwinding) and the exception is returned to the caller

Method throwException
Finally is always executed
Exception handled in main

Fig. 11.6 | Stack unwinding. (Part 2 of 2.)

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.

Pilha de execução

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

Onde está o *catch* ?

- ▶ O ambiente Java pesquisa a pilha de execução em busca de um tratamento para a exceção que foi gerada;
- ▶ Quando um tratamento adequado (i.e., para o tipo de exceção em questão) for encontrado, este assume o controle do programa;
 - Neste caso diz-se que o tratador de exceção “captura” (*catch*) o evento;
- ▶ Caso nenhum tratador seja encontrado, o controle chega de volta até *main()* e o programa termina.

Rastreamento de pilha

- ▶ As informações de rastreamento de pilha incluem:
 - O nome da exceção (ex, `java.lang.ArithmeticException`) em uma mensagem descritiva que indica o problema que ocorreu
 - O caminho de execução (pilha de chamadas de métodos) que resultou na exceção, método por método

`printStackTrace`, `getStackTrace` e `getMessage`

- ▶ A classe `Throwable` oferece um método chamado `printStackTrace` que envia para o fluxo de erro padrão o rastreamento da pilha
 - Útil para o processo de teste e depuração

`printStackTrace`, `getStackTrace` e
`getMessage`

- ▶ A classe **Throwable** também fornece o método `getStackTrace` que recupera informações sobre o rastreamento da pilha que podem ser impressas por `printStackTrace`
- ▶ O método `getMessage` da classe **Throwable** retorna a string descritiva armazenada em uma exceção

Exemplo printStackTrace

```
1 // Fig. 11.7: UsingExceptions.java
2 // Throwable methods getMessage, getStackTrace and printStackTrace.
3
4 public class UsingExceptions
5 {
6     public static void main( String[] args )
7     {
8         try
9         {
10            method1(); // call method1
11        } // end try
12    catch ( Exception exception ) // catch exception thrown in method1
13    {
14        System.err.printf( "%s\n\n", exception.getMessage() );
15        exception.printStackTrace(); // print exception stack trace
16
17        // obtain the stack-trace information
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    }
```

Starts the call chain that will lead to an exception in this program

None of the other methods catch the exception; so the stack is unwound and the exception is caught here

Gets an array of StackTraceElements

Fig. 11.7 | Throwable methods getMessage, getStackTrace and printStackTrace. (Part I of 3.)

Exemplo printStackTrace (2)

```
23     // loop through traceElements to get exception description
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     } // end for
31 } // end catch
32 } // end main
33
34 // call method2; throw exceptions back to main
35 public static void method1() throws Exception
36 {
37     method2();
38 } // end method method1
39
40 // call method3; throw exceptions back to method1
41 public static void method2() throws Exception
42 {
43     method3();
44 } // end method method2
```

StackTraceElement methods returns the class name, file name, line number and method name for a particular stack frame

This method might throw an Exception (this is a *checked* type)

Continues the call chain to method2

This method might throw an Exception (this is a *checked* type)

Continues the call chain to method3

Fig. 11.7 | Throwable methods getMessage, getStackTrace and printStackTrace. (Part 2 of 3.)

Exemplo printStackTrace (3)

```
45
46 // throw Exception back to method2
47 public static void method3() throws Exception
48 {
49     throw new Exception( "Exception thrown in method3" );
50 } // end method method3
51 } // end class UsingExceptions
```

This method might throw an Exception (this is a *checked* type)

Throws a new Exception and begins stack unwinding

Exception thrown in method3

Shows just the error message that was stored in the Exception object

```
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)
```

Shows the complete error message and stack trace

Stack trace from getStackTrace:

Shows the stack trace information obtained from StackTraceElements

| Class | File | Line | Method |
|-----------------|----------------------|------|---------|
| UsingExceptions | UsingExceptions.java | 49 | method3 |
| UsingExceptions | UsingExceptions.java | 43 | method2 |
| UsingExceptions | UsingExceptions.java | 37 | method1 |
| UsingExceptions | UsingExceptions.java | 10 | main |

Fig. 11.7 | Throwable methods getMessage, getStackTrace and printStackTrace. (Part 3 of 3.)

Declarando novos tipos de exceção

- ▶ Programadores podem achar útil declarar suas próprias classes de exceção
 - específicas aos problemas que podem ocorrer quando outro programador empregar suas classes reutilizáveis
- ▶ Uma nova classe de exceção deve estender uma classe de exceção existente
 - assegura que a classe pode ser utilizada com o mecanismo de tratamento de exceções

Declarando novos tipos de exceção

- ▶ Exceções são derivadas da classe *Exception*
- ▶ O construtor da exceção armazena no objeto criado informações sobre o evento (e.g., a mensagem de erro a ser exibida etc)
- ▶ Em geral uma classe de exceção possuirá dois construtores:
 - Um construtor default (i.e., sem argumentos) criando uma mensagem de erro padrão
 - Um construtor que recebe uma mensagem de exceção personalizada

Declarando novos tipos de exceção

- ▶ A string da mensagem é armazenada em uma variável do objeto **exceção** criado;
- ▶ Essa string pode ser recuperada pelo método **getMessage** da classe **Exception**;
- ▶ O próprio nome da exceção pode ser obtido com **toString(exceção)**

Gerando e tratando exceções

- ▶ O código em `try` (ou nos métodos por ele invocados) pode conter comandos `throw` para lançar uma nova exceção

```
try{
    if (condição) throw new MinhaExceção();
}
catch (MinhaExceção x) {
    System.out.println(x.getMessage());
}
```

Gerando e tratando exceções

- ▶ No tratamento da exceção (bloco `catch`) a mensagem criada pelo construtor da exceção pode ser obtida pelo método `getMessage` da classe **Exception**

```
try{
    if (condição) throw new MinhaExceção();
}
catch (MinhaExceção x) {
    System.out.println(x.getMessage());
}
```


Exemplo: divisão por zero

- ▶ Problema: o usuário entra com dois inteiros para divisão, e desejamos capturar erros de divisão por zero;
- ▶ Em `java.lang` não há uma exceção específica para divisão por zero
 - o mais próxima é a **`ArithmeticException`**
- ▶ Então estendemos e criamos nossa própria subclasse de exceção, que será chamada **`ExceçãoDivisãoPorZero`**

Exemplo: divisão por zero

```
// ExcecaoDivisaoPorZero.java
public class ExceçãoDivisãoPorZero extends Exception {

    public ExceçãoDivisãoPorZero() {
        super("Tentativa de divisão por zero");
    }

    public ExceçãoDivisãoPorZero(String msg) {
        super(msg);
    }
}
```

Exemplo: divisão por zero

```
69     try {
73         result = divisão( n1, n2 );
74         System.out.println(result);
75     }
82     catch (ExceçãoDivisãoPorZero e ) {
83         System.out.println(e.toString(), e.getMessage());
86     }
87 }
```

ExceçãoDivisãoPorZero:
Tentativa de divisão
por zero

- Se for gerada alguma exceção dentro do bloco *try*, o bloco inteiro é encerrado, e a execução é desviada para a cláusula *catch* correspondente;
- Não ocorrendo uma exceção, o código em *catch* é ignorado.

Exemplo: divisão por zero

- Uma exceção é lançada pelo comando **throw**

```
// em algum lugar dentro do método divisão...
94     {
95         if ( denominador == 0 )
96             throw new ExceçãoDivisãoPorZero();
97
98         return numerador / denominador;
99     }
100
```

Quando criar uma classe do tipo exceção ?

- ▶ Quando for preciso usar um tipo de exceção não definido na plataforma Java
- ▶ Quando for útil a distinção entre suas exceções e as geradas por outros programadores
- ▶ Quando o código gera várias exceções relacionadas

Lembre-se

- ▶ O tratamento de exceções pode fazer muito mais do que simplesmente exibir uma mensagem de erro:
 - Recuperação de erros;
 - Solicitar ao usuário orientação sobre como proceder;
 - Propagar o erro até um gerenciador de exceções de alto nível.

Vantagens em tratar exceções

1. Separação entre o código principal (e.g., da seqüência típica de eventos) do código de tratamento de erros;
2. Propagação de erros ao topo da pilha de execução;
 - Erros só precisam ser tratados por métodos que estão interessados neles

Vantagens em tratar exceções

3. Agrupamento e diferenciação de tipos de exceções em uma hierarquia:

- O tratamento de exceções pode ser tão genérico ou tão específico quanto desejado;
- Em geral procura-se definir exceções tão específicas quanto possível.

Erro comum de programação

- ▶ 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
- ▶ Então, ocorre um erro de compilação

Exercício 1

- ▶ Este código está correto ?

```
try    {  
}  
finally {  
}
```

Exercício 2- Qual a saída deste programa?

```
package yourownexception;
class MyException extends Exception {
    public MyException() {
    }
    public MyException(String msg) {
        super(msg);
    }
}
public class Main {
    public static void f() throws MyException {
        System.out.println("Throwing MyException from f()");
        throw new MyException();
    }
    public static void g() throws MyException {
        System.out.println("Throwing MyException from g()");
        throw new MyException("Originated in g()");
    }
    public static void main(String[] args) {
        try {
            f();
        } catch (MyException e) {
            System.out.println("Exception 1");
        }
        try {
            g();
        } catch (MyException e) {
            System.out.println("Exception 2");
        }
    }
}
```

Exercício 3 – Qual a saída deste programa?

```
package rethrowdifferentexception;

class OneException extends Exception {
    public OneException(String s) {
        super(s);
    }
}

class TwoException extends Exception {
    public TwoException(String s) {
        super(s);
    }
}

public class Main {

    public static void someMethod() throws OneException {
        System.out.println("originating the exception in someMethod()");
        throw new OneException("thrown from f()");
    }

    public static void main(String[] args) throws TwoException {
        try {
            someMethod();
        } catch (OneException e) {
            System.err.println("Caught in main, e.printStackTrace()");
            e.printStackTrace();

            throw new TwoException("from main()");
        }
    }
}
```

Exercício 4– Qual a saída deste programa?

```
class MyParentException extends Exception {  
    }  
class MyChildException extends MyParentException {  
    }  
  
public class Main {  
    public static void main(String[] args) {  
        try {  
            throw new MyChildException();  
        } catch (MyChildException s) {  
            System.err.println("Caught MyChildException");  
        } catch (MyParentException a) {  
            System.err.println("Caught MyParentException");  
        }  
    }  
}
```

Exercício 5 – Qual o erro deste código?

```
package catchingexceptionhierarchycompileerror;

class MyParentException extends Exception {
}

class MyChildException extends MyParentException {
}

public class Main {

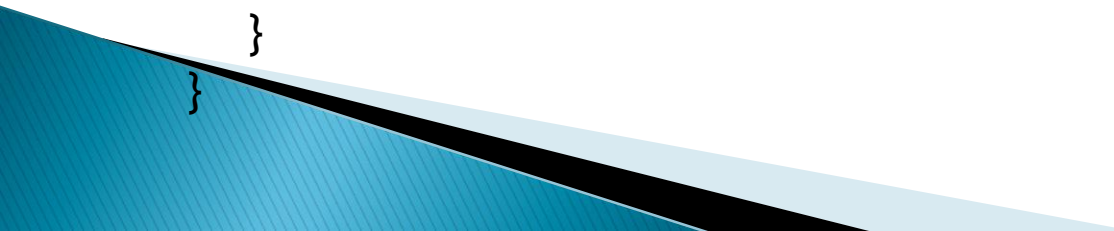
    public static void main(String[] args) {
        try {
            throw new MyChildException();
        } catch (MyParentException s) {
            System.err.println("Caught MyParentException");
        } catch (MyChildException a) {
            System.err.println("Caught MyChildtException");
        }
    }
}
```

Exercício 6 – Complete/corrija este código

```
package finallyworks;
class MyException extends Exception {
}

public class Main {
    static int count = 0;

    public static void main(String[] args) {
        while (true) {
            try {
                // Post-increment is zero first time:
                if (count++ == 0)
                    throw new MyException();
                if (count == 2)
                    break; // out of "while"
            }
        }
    }
}
```



Tutorial Eclipse

Introdução ao Eclipse

