

Programação Orientada a Objetos

Professor Filipe Dwan Pereira

Aula 09 – Controle de Erros e Exceções

“Confia no Senhor de todo o seu coração e não de apoia no teu próprio entendimento”.

Provérbios 3:5

Objetivo desta aula

- Ao término desta aula você será capaz de:
 - controlar erros e tomar decisões baseadas nos mesmos;
 - criar novos tipos de erros para melhorar o tratamento deles em sua aplicação ou biblioteca;
 - assegurar que um método funcionou como diz em seu “contrato”.

Motivação

Voltando ao exemplo da Conta:

```
Conta minhaConta = new Conta();  
minhaConta.deposita(100);  
minhaConta.setLimite(100);  
minhaConta.saca(1000);  
//      o saldo é -900? É 100? É 0? A chamada ao método saca funcionou?
```

Em sistemas de verdade, é muito comum que quem saiba tratar o erro é aquele que chamou o método e não a própria classe! Portanto, nada mais natural do que a classe sinalizar que um erro ocorreu.

Avisando Sobre Falhas em Métodos

- Você precisa avisar quem chamou o método e informar que o método não executou como deveria. Como fazer?
- As abordagens mais comuns são
 - Usar booleanos
 - Usar magic numbers

Problema de usar booleanos

E se o retorno
não for tratado?

```
boolean sucesso = o.processar();  
  
if(sucesso) {  
    //código em caso de sucesso  
} else {  
    //código em caso de falha  
}
```

O que
falhou?

Problema ao usar Magic Numbers

E se o retorno
não for tratado?

```
int resultado = o.processar();

if(resultado == 100) {
    //sucesso
} else if (resultado == 110) {
    //falha na validação
} else if (resultado == 120) {
    //falha de gravação no arquivo
}
```

Como entender este código sem
uma tabela de códigos de erro?

Colocar um retorno booleano para o método `saca()`.

```
boolean saca(double quantidade) {  
    // posso sacar até saldo+limite  
    if (quantidade > this.saldo + this.limite) {  
        System.out.println("Não posso sacar fora do limite!");  
        return false;  
    } else {  
        this.saldo = this.saldo - quantidade;  
        return true;  
    }  
}
```

Um novo exemplo de chamada do método:

```
Conta minhaConta = new Conta();  
minhaConta.deposita(100);  
minhaConta.setLimite(100);  
if (!minhaConta.saca(1000)) {  
    System.out.println("Não saquei");  
}
```

Repare que tivemos de lembrar de testar o retorno do método:

```
Conta minhaConta = new Conta();  
minhaConta.deposita(100);  
  
// ...  
double valor = 5000;  
minhaConta.saca(valor); // vai retornar false, mas ninguém verifica!  
caixaEletronico.emite(valor);
```


Mesmo se nós tratássemos o retorno. O que faríamos se fosse necessário sinalizar que um usuário passou um valor negativo no método `deposita()`?

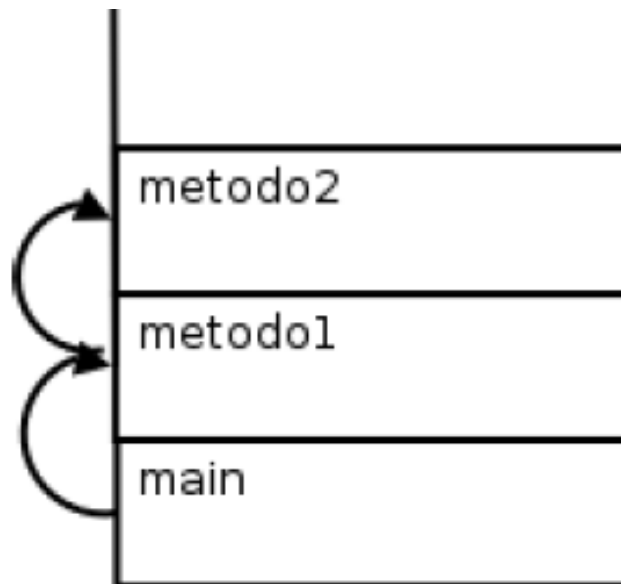
- Outra solução seria trocar o tipo de retorno para `int` e com o código de erro que ocorreu. O que também é um má prática (magic numbers).
- Perderíamos o retorno do método e o código poderia ainda continuar sem ser tratado. Além disso, geraria um manual informando o que cada número significa.

Vamos ver como a JVM age quando se depara com divisões por zero ou acesso a um índice de array que não existe:

```
class TesteErro {  
    public static void main(String[] args) {  
        System.out.println("inicio do main");  
        metodo1();  
        System.out.println("fim do main");  
    }  
}
```

```
static void metodo1() {  
    System.out.println("inicio do metodo1");  
    metodo2();  
    System.out.println("fim do metodo1");  
}  
  
static void metodo2() {  
    System.out.println("inicio do metodo2");  
    int[] array = new int[10];  
    for (int i = 0; i <= 15; i++) {  
        array[i] = i;  
        System.out.println(i);  
    }  
    System.out.println("fim do metodo2");  
}
```

Em Java toda invocação de método é empilhada em uma estrutura de dados que isola a área de memória de cada um. Quando um método termina(retorna), ele volta para o método que o invocou. Isso é a pilha de execução (stack), basta jogar fora um gomo da pilha (**stackframe**):



O nosso método2 propositalmente possui um grande problema:

```
início do main
início do metodo1
início do metodo2
0
1
2
3
4
5
6
7
8
9
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
    at Teste.metodo2(Teste.java:18)
    at Teste.metodo1(Teste.java:10)
    at Teste.main(Teste.java:4)
```

Esse é um rastro da pilha stacktrace

- Quando uma exceção é **lançada (throws)** a JVM entra em estado de alerta e vai ver se o método atual toma alguma precaução ao **tentar (try)** executar esse trecho de código.
- O método2 não está tratando este problema, então a JVM pára a execução dele anormalmente e volta um stackframe para baixo, onde será feita nova verificação.
- O método1 está tratando de um algum problema chamado de `ArrayOutOfBoundsException`? Não!
- Logo, volta para o main, onde também não há proteção, então a JVM morre (a thread corrente morre).

```
1 // Figura 11.1: DivideByZeroNoExceptionHandling.java
2 // Divisão de inteiro sem tratamento de exceções.
3 import java.util.Scanner;
4
5 public class DivideByZeroNoExceptionHandling
6 {
7     // demonstra o lançamento de uma exceção quando ocorre uma divisão por zero
8     public static int quotient( int numerator, int denominator )
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 denominator: " );
20         int denominator = scanner.nextInt();
21
22         int result = quotient( numerator, denominator );
```

A JVM lança uma exceção se denominator for 0

O usuário poderia digitar uma entrada inválida

O usuário poderia digitar uma entrada inválida (incluindo 0)

Figura 11.1 | Divisão de inteiro sem tratamento de exceções. (Parte 1 de 3.)

```
23         System.out.printf(  
24             "\nResult: %d / %d = %d\n", numerator, denominator, result );  
25     } // fim de main  
26 } // fim da classe DivideByZeroNoExceptionHandling
```

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)

Causa divisão por 0; o rastreamento da pilha mostra o que levou à exceção

Figura 11.1 | Divisão de inteiro sem tratamento de exceções. (Parte 2 de 3.)

Exceções

- Exceções representam algo estranho ao sistema que normalmente não ocorre
- Em Java, o tratamento de exceções é feito por um código diferente do código executado quando não ocorre a exceção

Classes que representam Exceções

- Exceções são representadas por classes
- As classes devem herdar direta ou indiretamente de Exception
- O Java tem classes que representam diversos tipos de exceção, mas o programador pode criar exceções específicas de acordo com a necessidade

Voltando ao caso do `ArrayOutOfBoundsException`. Podemos colocar um try/cath em volta do for:

```
try {  
    for (int i = 0; i <= 15; i++) {  
        array[i] = i;  
        System.out.println(i);  
    }  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("erro: " + e);  
}
```

O código vai imprimir:

```
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
erro: java.lang.ArrayIndexOutOfBoundsException: 10
fim do metodo2
fim do metodo1
fim do main
```

Se em vez de fazer o try no for inteiro eu fizesse dentro dele:

```

início do main
início do metodo1
início do metodo2
0
1
2
3
4
5
6
7
8
9
erro: java.lang.ArrayIndexOutOfBoundsException: 10
erro: java.lang.ArrayIndexOutOfBoundsException: 11
erro: java.lang.ArrayIndexOutOfBoundsException: 12
erro: java.lang.ArrayIndexOutOfBoundsException: 13
erro: java.lang.ArrayIndexOutOfBoundsException: 14
erro: java.lang.ArrayIndexOutOfBoundsException: 15
fim do metodo2
fim do metodo1
fim do main

for (int i = 0; i <= 15; i++) {
    try {
        array[i] = i;
        System.out.println(i);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("erro: " + e);
    }
}
```

Colocando o try/catch na chamada do método 2:

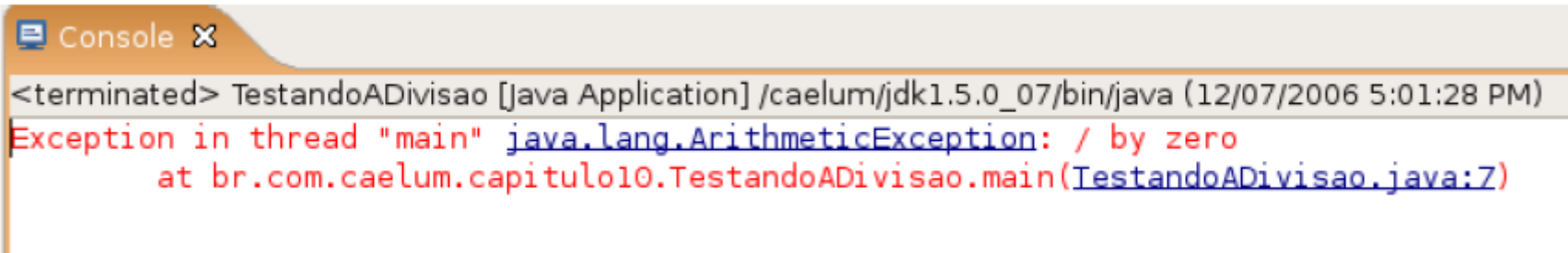
```
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
erro: java.lang.ArrayIndexOutOfBoundsException: 10
fim do metodo1
fim do main
```

Colocando o try/catch na chamada do método 1 dentro do main:

```
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
Erro : java.lang.ArrayIndexOutOfBoundsException: 10
fim do main
```

Exceções de Runtime mais comuns

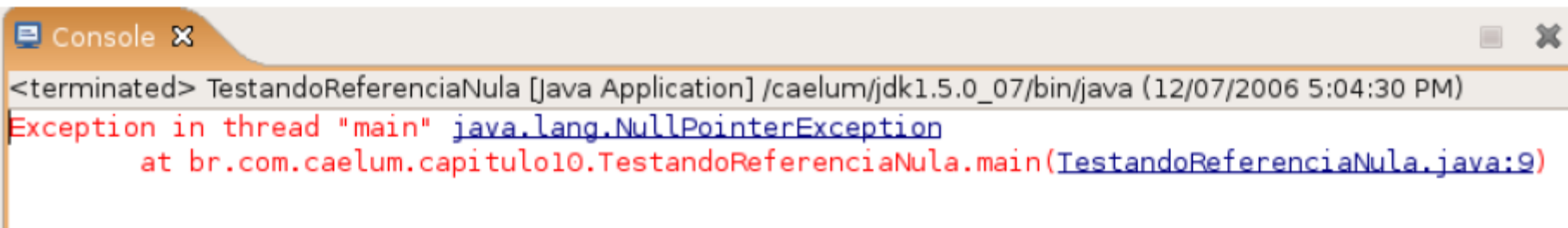
```
public class TestandoADivisao {  
  
    public static void main(String args[]) {  
        int i = 5571;  
        i = i / 0;  
        System.out.println("O resultado " + i);  
    }  
}
```

A screenshot of a Java IDE's console window. The window has a title bar with a document icon, the word 'Console', and a close button. The text inside the console shows the execution of a Java application. It starts with '<terminated> TestandoADivisao [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 5:01:28 PM)'. Below this, an exception is displayed in red text: 'Exception in thread "main" java.lang.ArithmeticException: / by zero'. The stack trace continues in black text: 'at br.com.caelum.capitulo10.TestandoADivisao.main(TestandoADivisao.java:7)'.

```
Console X  
<terminated> TestandoADivisao [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 5:01:28 PM)  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at br.com.caelum.capitulo10.TestandoADivisao.main(TestandoADivisao.java:7)
```

Exceções de Runtime mais comuns

```
public class TestandoReferenciaNula {  
    public static void main(String args[]) {  
        Conta c = null;  
        System.out.println("Saldo atual " + c.getSaldo());  
    }  
}
```

A screenshot of a Java IDE's console window. The window has a title bar with a 'Console' tab and a close button. The text inside shows the execution of a Java application. It starts with a prompt '<terminated>' followed by the application name 'TestandoReferenciaNula [Java Application]' and the command used to run it: '/caelum/jdk1.5.0_07/bin/java (12/07/2006 5:04:30 PM)'. Below this, an exception is displayed in red text: 'Exception in thread "main" java.lang.NullPointerException'. The exception details include the class and method where it occurred: 'at br.com.caelum.capitulo10.TestandoReferenciaNula.main(TestandoReferenciaNula.java:9)'.

```
<terminated> TestandoReferenciaNula [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 5:04:30 PM)  
Exception in thread "main" java.lang.NullPointerException  
    at br.com.caelum.capitulo10.TestandoReferenciaNula.main(TestandoReferenciaNula.java:9)
```

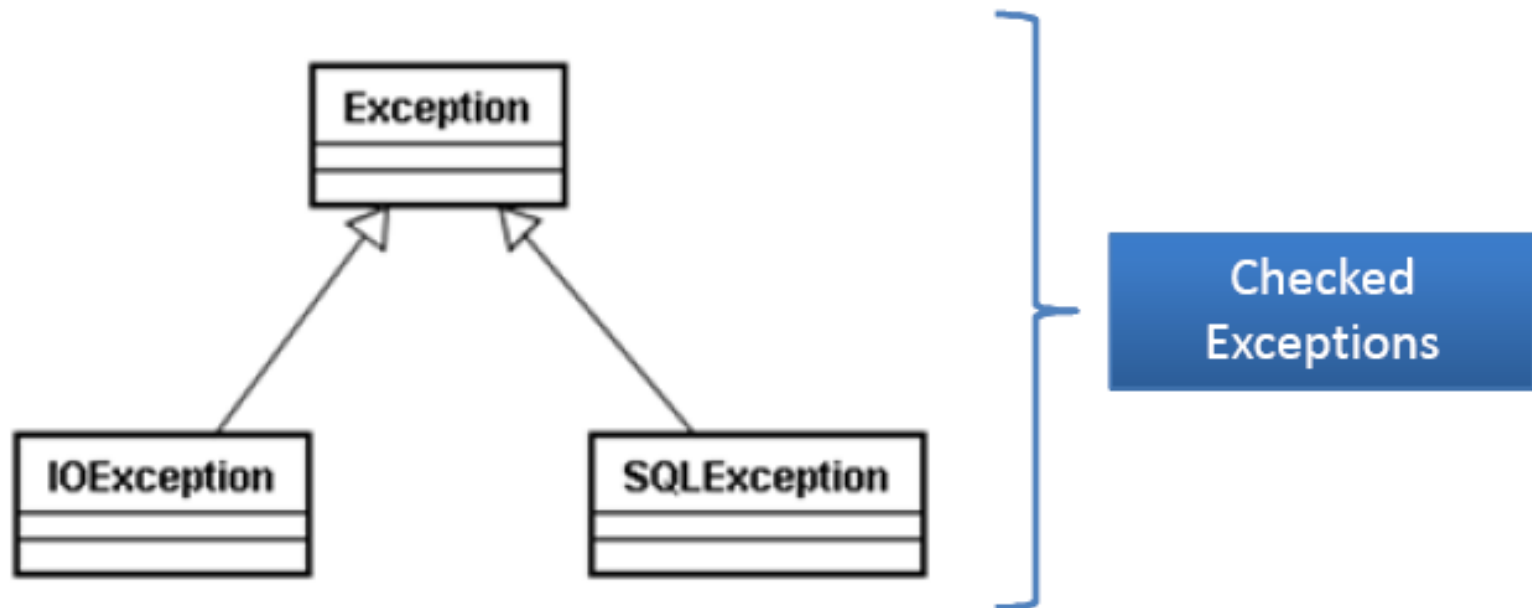

- Perceba que o `ArrayIndexOutOfBoundsException` ou um `NullPointerException` poderia ser facilmente evitado com o `for` corretamente escrito ou com `ifs` que checariam o limite do `Array`.
- Quando ocorre um `cast` errado também...
- Todos os casos poderiam ser evitados pelo programador.
- É por esse motivo que o `java` não te obriga a dar o `try/catch` nessas exceptions e chamamos essas exceções de **unchecked**. Ou seja, o compilador não checa se você está tratando as exceções.

Outro tipo de exceção: Checked Exceptions

- Nos exemplos que usamos com ou sem o try/catch, compilaram e rodaram. Em um, o erro terminou o programa e, no outro, foi possível tratá-lo.
- Existe outro tipo de exceção que obriga a quem chama o método ou construtor a tratá-la (checked).
- O compiladorá checará se ela está sendo devidamente tratada.

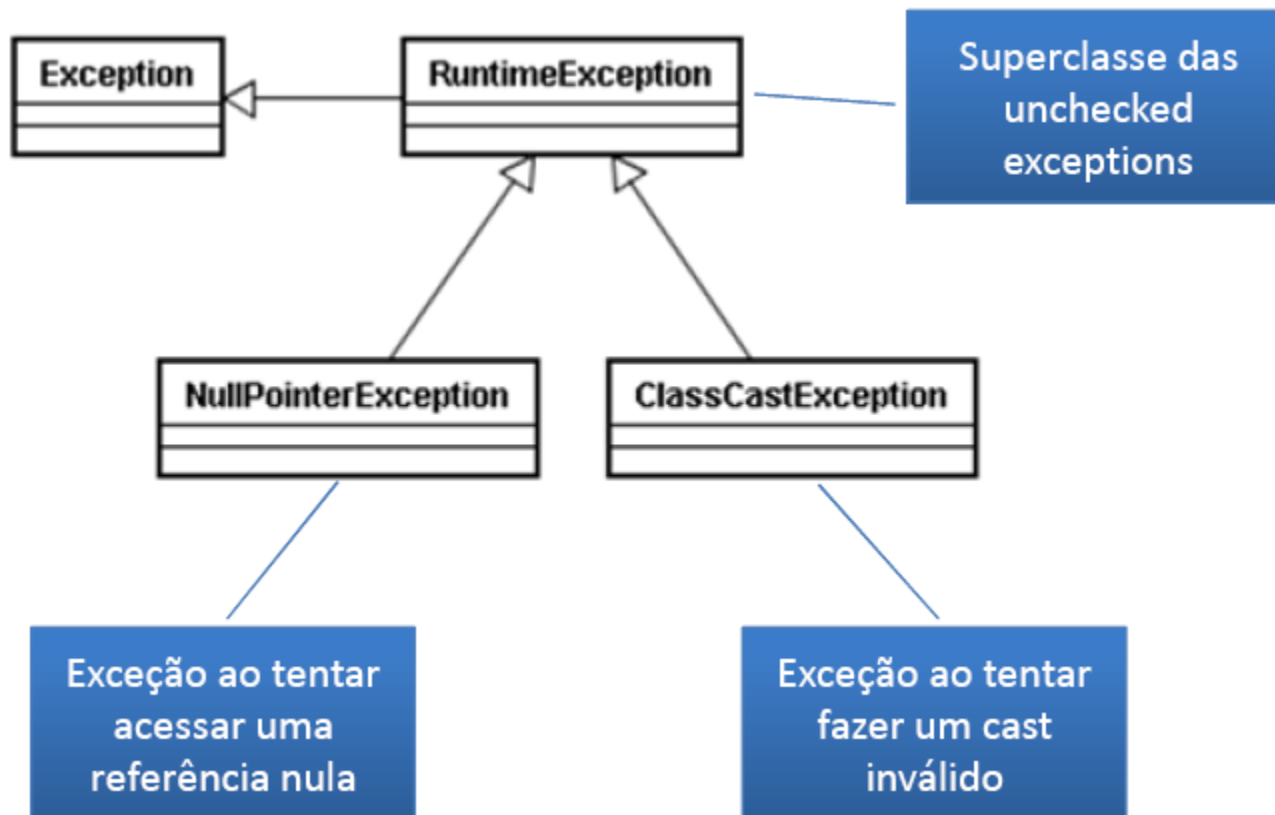
Checked Exceptions

- Herdam direta ou indiretamente de **Exception**
- Só não podem herdar de RuntimeException



Unchecked Exceptions

- Também chamadas de runtime exceptions
- Herdam direta ou indiretamente de **RuntimeException**



Exemplo de abrir um arquivo para leitura, onde pode ocorrer o erro de o arquivo não existir:

```
class Teste {  
    public static void metodo() {  
        new java.io.FileInputStream("arquivo.txt");  
    }  
}
```

O código acima não compila e o compilador avisará que é necessário tratar o `FileNotFoundException` que pode ocorrer:

```
Teste.java:3: unreported exception java.io.FileNotFoundException; must be caught  
or declared to be thrown  
    new java.io.FileReader("arquivo.txt");  
    ^  
1 error
```

- Para fazer o programa funcionar temos duas formas: utilizando o **try/catch** e o **throws**

```
public static void metodo() {  
  
    try {  
        new java.io.FileInputStream("arquivo.txt");  
    } catch (java.io.FileNotFoundException e) {  
        System.out.println("Nao foi possível abrir o arquivo para leitura");  
    }  
}
```

- Podemos delegar ele para quem chamou o nosso método, ou seja, passar para a frente:

```
public static void metodo() throws java.io.FileNotFoundException {  
  
    new java.io.FileInputStream("arquivo.txt");  
  
}
```

Lançando Exceções

- O lançamento de exceções é feito através do **throw**

```
public void fazerAlgo() throws Exception {  
    throw new Exception();  
}
```

O *throw* é usado para lançar a exceção

O *throws* indica que o método pode lançar a exceção

Lançando Exceções

- É possível também lançar subclasses da exceção declarada pelo **throws**

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

*IOException é uma
subclasse de Exception*

*A declaração `throws Exception`
está de acordo com a exceção
lançada pelo método*

Tratando exceções

- Exceções podem ser tratadas através do uso do bloco try/catch
 - Determinado código tenta (try) executar um método e, caso alguma exceção aconteça, ele pega (catch) a exceção ocorrida e faz o que deseja
- Após uma exceção ter alcançado o bloco catch, o código volta o seu fluxo normal de execução

Tratando Exceções

```
public void m1() throws Exception {  
    throw new Exception();  
}
```

```
public void m2() {  
    try {  
        m1();  
    } catch (Exception e) {  
        ...  
    }  
    ...  
}
```

Se uma *Exception* acontecer,
o fluxo é desviado para o
bloco *catch*

Ao fim do bloco *catch*, a
execução continua após o bloco

Tratando Múltiplas Exceções

```
public void m1() throws IOException, SQLException {  
    ...  
}
```

```
public void m2() {  
    try {  
        m1();  
    } catch (IOException e) {  
        ...  
    } catch (SQLException e) {  
        ...  
    }  
    ...  
}
```

Dependendo da exceção, o bloco *catch* correspondente é executado

No máximo um bloco *catch* é executado

Multi-Catch

- É possível fazer o catch de mais de uma exceção ao mesmo tempo



```
try {  
    m();  
} catch (MyException1 e) {  
    ...  
} catch (MyException2 e) {  
    ...  
} catch (MyException3 e) {  
    ...  
}
```

```
try {  
    m();  
} catch (MyException1 | MyException2 | MyException3 e) {  
    ...  
}
```

Ordem das Exceções do catch

```
public void m1() throws IOException {  
    throw new IOException();  
}
```

```
public void m2() {  
    try {  
        m1();  
    } catch (Exception e) {  
        ...  
    } catch (IOException e) {  
        ...  
    }  
    ...  
}
```

Toda exceção será tratada
por este bloco *catch*

Este bloco *catch* nunca será
executado

Tratando e lançando exceções

```
public void m1() throws IOException, SQLException {  
    ...  
}
```

```
public void m2() throws IOException {  
    try {  
        m1();  
    } catch (SQLException e) {  
        ...  
    }  
    ...  
}
```

Apenas a *SQLException* é tratada.
A *IOException* é lançada para
quem chamou *m2()*

Transformando Exceções

```
public void m1() throws IOException {  
    ...  
}
```

```
public void m2() throws AppException {  
    try {  
        m1();  
    } catch (IOException e) {  
        throw new AppException();  
    }  
    ...  
}
```



A *IOException* é relançada
como uma *AppException*

Exemplificando...

- A palavra chave **throw** lança uma exceção. Diferente de **throws** que avisa da possibilidade de um método lançar exceção, obrigando o método que chama esse a se preocupar com a exceção.

```
void saca(double valor) {  
    if (this.saldo < valor) {  
        throw new RuntimeException();  
    } else {  
        this.saldo-=valor;  
    }  
}
```

Nesse caso uma exceção unchecked. RuntimeException é a mãe de todas as exceções unchecked.

- RuntimeException é muito genérica, podemos usar uma mais específica:

```
void saca(double valor) {  
    if (this.saldo < valor) {  
        throw new IllegalArgumentException();  
    } else {  
        this.saldo-=valor;  
    }  
}
```

IllegalArgumentException diz um pouco mais: algo foi passado como argumento e seu método não gostou. Ela é uma Exception unchecked pois estende de RuntimeException.

- Para pegarmos esse erro usaremos um try/catch:

```
Conta cc = new ContaCorrente();
cc.deposita(100);

try {
    cc.saca(100);
} catch (IllegalArgumentException e) {
    System.out.println("Saldo Insuficiente");
}
```

- Poderíamos passar no construtor do método o motivo da exceção

```
void saca(double valor) {
    if (this.saldo < valor) {
        throw new IllegalArgumentException("Saldo insuficiente");
    } else {
        this.saldo-=valor;
    }
}
```

- O método **getMessage()** definido na classe Throwable (mãe de todos os tipos de erros e exceptions) vai retornar a mensagem que passamos ao construtor da IllegalArgumentException.

```
try {  
    cc.saca(100);  
} catch (IllegalArgumentException e) {  
    System.out.println(e.getMessage());  
}
```


- Nós poderíamos colocar o try/catch dentro do método saca():

```
public void saca (double valor) {  
    try {  
        if (valor > this.saldo) {  
            throw new IllegalArgumentException("Saldo Insuficiente");  
        }  
        else if (valor < 0) {  
            throw new IllegalArgumentException("Valor inválido");  
        }  
        else this.saldo -= valor;  
    }  
    catch (IllegalArgumentException erro) {  
        System.out.println(erro.getMessage());  
    }  
}
```

Criando seu próprio tipo de exceção:

- Vamos criar nossa própria classe de exceção `SaldoInsuficienteException`:

```
public class SaldoInsuficienteException extends RuntimeException {  
  
    SaldoInsuficienteException(String message) {  
        super(message);  
    }  
}
```



Perceba que `RuntimeException` é **unchecked**

Em vez de lançar um `IllegalArgumentException`, vamos lançar nossa própria exception, com uma mensagem que dirá “Saldo Insuficiente”:

- Fazendo o teste:

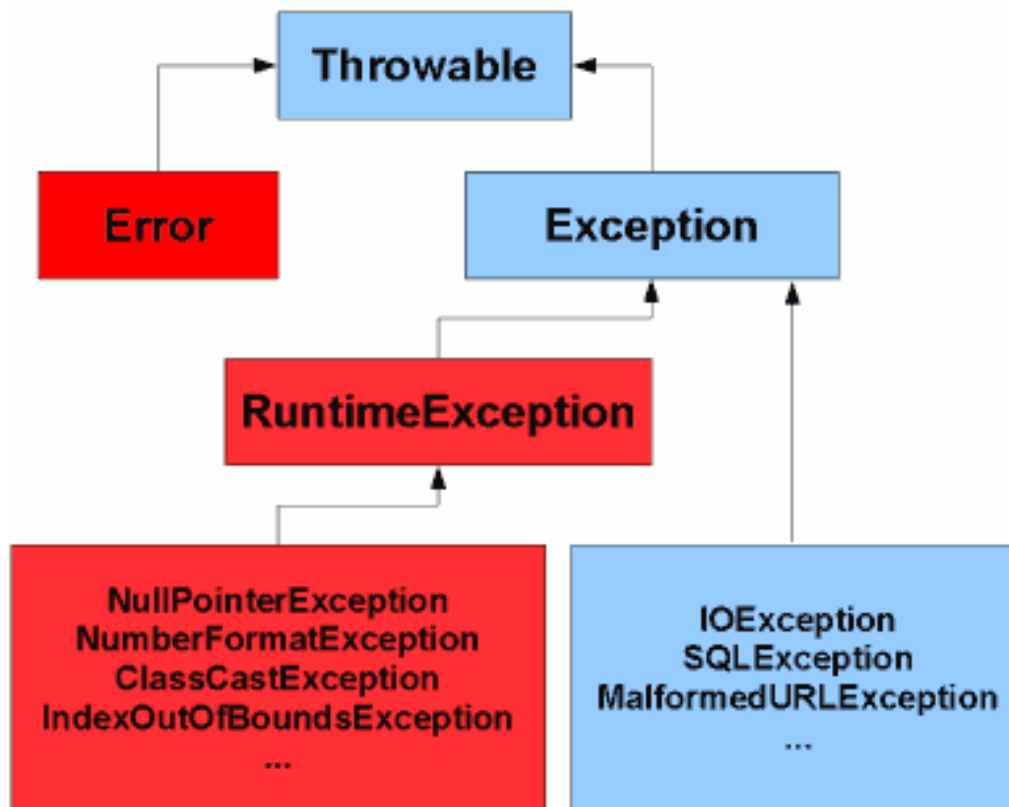
```
public static void main(String[] args) {  
    Conta cc = new ContaCorrente();  
    cc.deposita(10);  
  
    try {  
        cc.saca(100);  
    } catch (SaldoInsuficienteException e) {  
        System.out.println(e.getMessage());  
    }  
}
```

- Podemos transformar essa exceção de **unchecked** para **checked**, obrigando a quem chama esse método a dar **try/catch** ou **throw**:

```
public class SaldoInsuficienteException extends Exception {  
  
    SaldoInsuficienteException(String message) {  
        super(message);  
    }  
}
```

Perceba que Exception é **checked**

Hierarquia de herança - Throwable



Resumindo

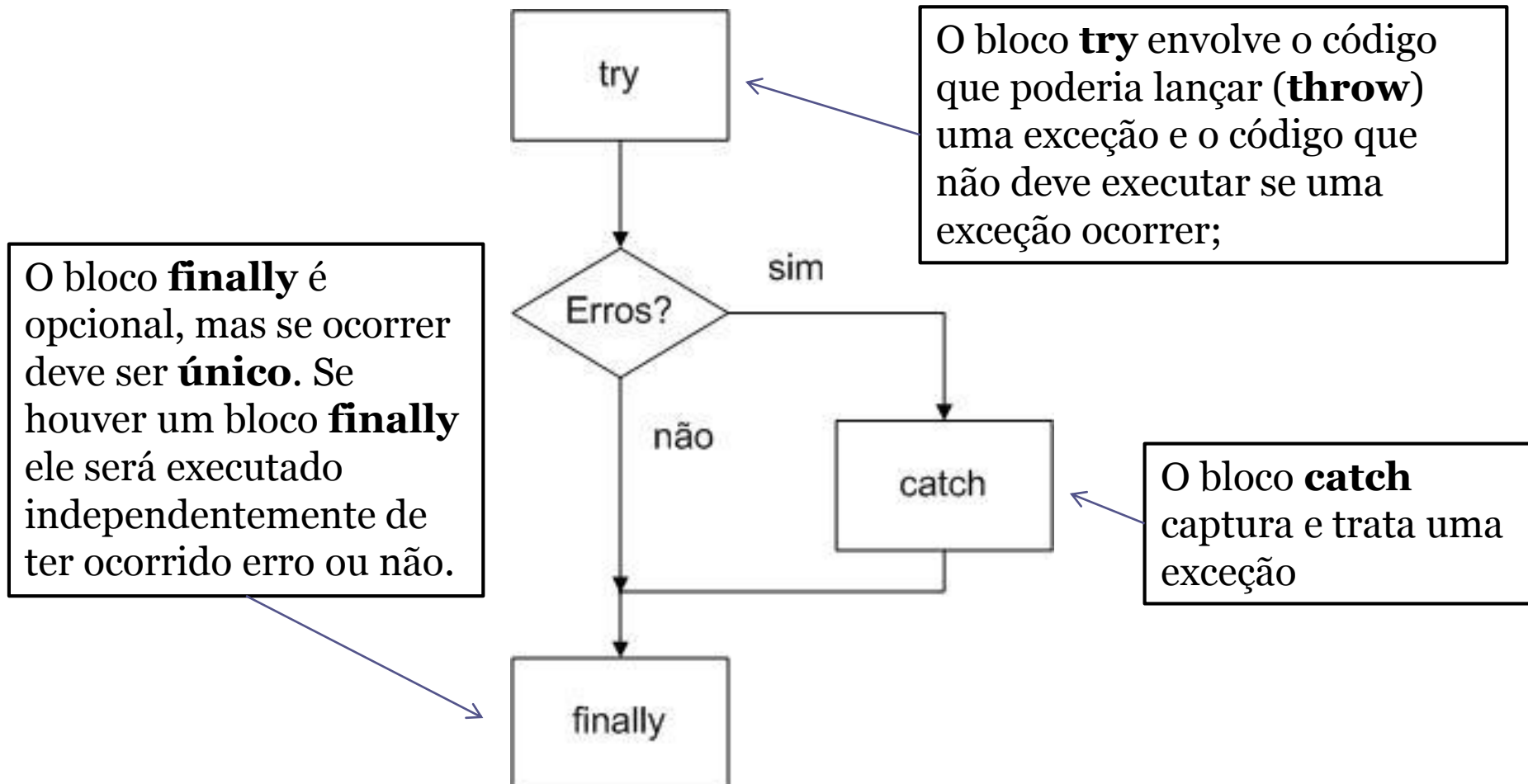
Checked Exceptions

- Representam condições inválidas em áreas fora do controle imediato do programa (problemas de entradas do usuário inválidas, banco de dados, falhas de rede, arquivos ausentes);
- São subclasses de Exception;
- Um método é obrigado a estabelecer uma política para todas as exceções verificadas lançadas por sua implementação (ou passar a exceção verificada mais acima na pilha, ou manipulá-lo de alguma forma).

Unchecked Exceptions

- Representam defeitos no programa (bugs) - muitas vezes argumentos inválidos passados para um método não privado. Na obra “A Linguagem de Programação Java”, por Gosling, Arnold, e Holmes, temos: "exceções de tempo de execução unchecked representam condições que, em geral, refletem erros na lógica do seu programa e não pode ser razoavelmente recuperados em tempo de execução.";
- São subclasses de RuntimeException, e geralmente são implementadas usando IllegalArgumentException, NullPointerException, ou IllegalStateException;
- Um método não é obrigado a estabelecer uma política para as exceções não verificadas lançadas por sua execução (e quase sempre não fazem).

Uso da estrutura try-catch-finally



Estrutura try-catch-finally

Estrutura ***try-catch***:

```
try{  
    <bloco de instruções>  
}  
catch (<nome da exceção 1>){  
    <tratamento da exceção 1>  
}  
catch (<nome da exceção 2>) {  
    <tratamento da exceção 2>  
}  
...  
catch (<nome da exceção n>) {  
    <tratamento da exceção n>  
}  
finally{  
    <instruções finais>  
}
```

Utilize o objeto **System.err**
(fluxo de erro padrão) para
imprimir mensagens de erro;



Erro comum de programação I 1.1

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



Erro comum de programação I 1.2

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

Uso da estrutura try-catch-finally

- As exceções são nomeadas segundo o seu tipo, por exemplo:
- **NumberFormatException** (erros de formato de dados)
- **ArithmeticException** (divisão por zeros entre inteiros)
- **IOException** (erros de E/S de dados)
- **ArrayIndexOutOfBoundsException** (indexação fora dos limites do vetor)
- **InterruptedException** (erro de interrupção)
- **NullPointerException** ocorre quando uma referência null é utilizada onde um objeto é esperado;

Exception (exceção genérica, isto é, não particularizada)

Algumas considerações

- Se uma exceção for lançada pelo método `main()`, a JVM termina.
- Exceções muito genéricas dificultam no entendimento do problema

Assertions

- Garantir qualidade do código, executando testes que permitem validar a lógica e as suposições sobre o programa
- São usadas em tempo de desenvolvimento e desabilitadas em produção
- Usados para testes unitário

Exemplo

- Exemplo de utilização

```
public void metodo(int arg) {  
    assert arg > 0;  
  
    ...  
}
```

- Se a assertion falhar, a JVM lançará um `AssertionError`

Mais um exemplo

- Outro exemplo

```
public void metodo(int arg) {  
    assert arg > 0 : "arg menor que 0";  
  
    // ...  
}
```

- Funciona como o exemplo anterior, mas a string fornecida será passada no construtor do `AssertionError`

Habilitando assertions

- Por padrão, as assertions ficam desabilitadas
- Para habilitá-las, é preciso passar o parâmetro `-ea` ao iniciar a JVM:

```
java -ea MinhaClasse
```

Considerações sobre Assertions

- Não use assertions para validar parâmetros de métodos públicos
 - Métodos públicos têm um comportamento bem definido sobre o que ocorre na passagem de parâmetros
 - Este comportamento deve ser o mesmo, havendo assertions ou não
- Não escreva assertions que interfiram na execução da aplicação
 - Estando a assertion habilitada ou não, a aplicação deve funcionar da mesma forma

Um pouco mais...

Outro exemplo:

```
1 // Figura 11.2: DivideByZeroWithExceptionHandling.java
2 // Tratando ArithmeticExceptions e InputMismatchExceptions.
3 import java.util.InputMismatchException;
4 import java.util.Scanner;
5
6 public class DivideByZeroWithExceptionHandling
7 {
8     // demonstra o lançamento de uma exceção quando ocorre uma divisão por zero
9     public static int quotient( int numerator, int denominator )
10         throws ArithmeticException
11     {
12         return numerator / denominator; // possível divisão por zero
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 continueLoop = true; // determina se mais entradas são necessárias
19
```

Tipo de exceção lançada por vários métodos da classe Scanner

Indica que esse método talvez lance uma ArithmeticException

Figura 11.2 | Tratando ArithmeticExceptions e InputMismatchExceptions. (Parte 1 de 4.)

do

{

try // lê dois números e calcula o quociente

{

System.out.print("Please enter an integer numerator: ");

int numerator = scanner.nextInt();

System.out.print("Please 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; // entrada bem-sucedida; fim do loop

} // fim do try

catch (InputMismatchException inputMismatchException)

{

System.err.printf("\nException: %s\n",

inputMismatchException);

scanner.nextLine(); // descarta entrada para o usuário poder tentar de novo

System.out.println(

"You must enter integers. Please try again.\n");

} // fim do catch

Inicia um bloco de código no qual uma exceção talvez ocorra; o bloco também contém código que não deveria executar se uma exceção ocorrer

Captura e processa InputMismatchExceptions

Figura 11.2 | Tratando ArithmeticExceptions e InputMismatchExceptions. (Parte 2 de 4.)

```
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 do catch
48 } while ( continueLoop ); // fim da instrução do...while
49 } // fim de main
50 } // fim da classe DivideByZeroWithExceptionHandling
```

Captura e processa
Arithmetic-
Exceptions

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7
Result: 100 / 7 = 14
```

Figura 11.2 | Tratando ArithmeticExceptions e InputMismatchExceptions. (Parte 3 de 4.)

```
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
```

Exibimos de propósito a mensagem de erro da exceção

```
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
```

Exibimos de propósito a mensagem de erro da exceção

Figura 11.2 | Tratando ArithmeticExceptions e InputMismatchExceptions. (Parte 4 de 4.)

Exemplos

- mod8
- Mod-08__Cf-03
- Mod-08__Cf-04
- Mod-08__Cf-05
- Mod-08__Cf-06
- Mod-08__Cf-07

Referências Bibliográficas

- DEITEL, Harvey M. e DEITEL, Paul J. Java - Como Programar, 8ª edição. Pearson. 2010.
- BLOCH, Joshua. Effective Java, 2ª edição. Addison-Wesley, 2008.
- CAELUM. Java e Orientação a Objetos. Disponível em: <https://www.caelum.com.br/apostila-java-orientacao-objetos/>
- SOFTBLUE. Professor Carlos Eduardo Gusso Tosin. Fundamentos de Java. <http://www.softblue.com.br/>.
- K19. Java e Orientação a Objetos. Disponível em: <http://www.k19.com.br/cursos/orientacao-a-objetos-em-java>.
- HORSTMANN, CORNELL. Core Java Volume I – Fundamentos, 8ª Edição. São Paulo, Pearson Education, 2010.
- BRAUDE, E. J. Projeto de software - da programação à arquitetura: uma abordagem baseada em Java. Porto Alegre: Bookman, 2005.
- SANTOS, R. Introdução à Programação Orientada a Objetos usando Java. São Paulo: Campus, 2003.
- Slides do Professor Doutor Horácio Fernandes da UFAM.