

CAPÍTULO 11

Exceções e controle de erros

"Quem pensa pouco, erra muito"
— Leonardo da Vinci

Ao término desse capítulo, 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".

11.1 – MOTIVAÇÃO

Voltando às Contas que criamos no capítulo 6, o que aconteceria ao tentar chamar o método `saca` com um valor fora do limite? O sistema mostraria uma mensagem de erro, mas quem chamou o método `saca` não saberá que isso aconteceu.

Como avisar aquele que chamou o método de que ele não conseguiu fazer aquilo que deveria?

Em Java, os métodos dizem qual o **contrato** que eles devem seguir. Se, ao tentar sacar, ele não consegue fazer o que deveria, ele precisa, ao menos, avisar ao usuário que o saque não foi feito.

Veja no exemplo abaixo: estamos forçando uma Conta a ter um valor negativo, isto é, estar num estado inconsistente de acordo com a nossa modelagem.

```
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.

A solução mais simples utilizada antigamente é a de marcar o retorno de um método como boolean e retornar true, se tudo ocorreu da maneira planejada, ou false, caso contrário:

```
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 ao método acima:

```
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, mas não somos obrigados a fazer isso. Esquecer de testar o retorno desse método teria consequências drásticas: a máquina de autoatendimento poderia vir a liberar a quantia desejada de dinheiro, mesmo que o sistema não tivesse conseguido efetuar o método saca com sucesso, como no exemplo a seguir:

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

Mesmo invocando o método e tratando o retorno de maneira correta, o que faríamos se fosse necessário sinalizar quando o usuário passou um valor negativo como **quantidade**? Uma solução seria alterar o retorno de boolean para int e retornar o código do erro que ocorreu. Isso é considerado uma má prática

(conhecida também como uso de "*magic numbers*").

Além de você perder o retorno do método, o valor devolvido é "mágico" e só legível perante extensa documentação, além de não obrigar o programador a tratar esse retorno e, no caso de esquecer isso, seu programa continuará rodando já num estado inconsistente.

Repare o que aconteceria se fosse necessário retornar um outro valor. O exemplo abaixo mostra um caso onde, através do retorno, não será possível descobrir se ocorreu um erro ou não, pois o método retorna um cliente.

```
public Cliente procuraCliente(int id) {  
    if (idInvalido) {  
        // avisa o método que chamou este que ocorreu um erro  
    } else {  
        Cliente cliente = new Cliente();  
        cliente.setId(id);  
        // cliente.setNome("nome do cliente");  
        return cliente;  
    }  
}
```

Por esses e outros motivos, utilizamos um código diferente em Java para tratar aquilo que chamamos de exceções: os casos onde acontece algo que, normalmente, não iria acontecer. O exemplo do argumento do saque inválido ou do id inválido de um cliente é uma **exceção** à regra.

Exceção

Uma exceção representa uma situação que normalmente não ocorre e representa algo de estranho ou inesperado no sistema.

11.2 – EXERCÍCIO PARA COMEÇAR COM OS CONCEITOS

Antes de resolvermos o nosso problema, vamos ver como a Java Virtual Machine age ao se deparar com situações inesperadas, como divisão por zero ou acesso a um índice da array que não existe.

1. Para aprendermos os conceitos básicos das exceptions do Java, teste o seguinte código você mesmo:

```
class TesteErro {  
    public static void main(String[] args) {
```

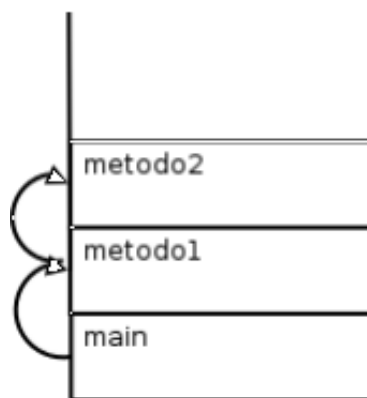
```
System.out.println("inicio do main");
metodo1();
System.out.println("fim do main");
}

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");
}
}
```

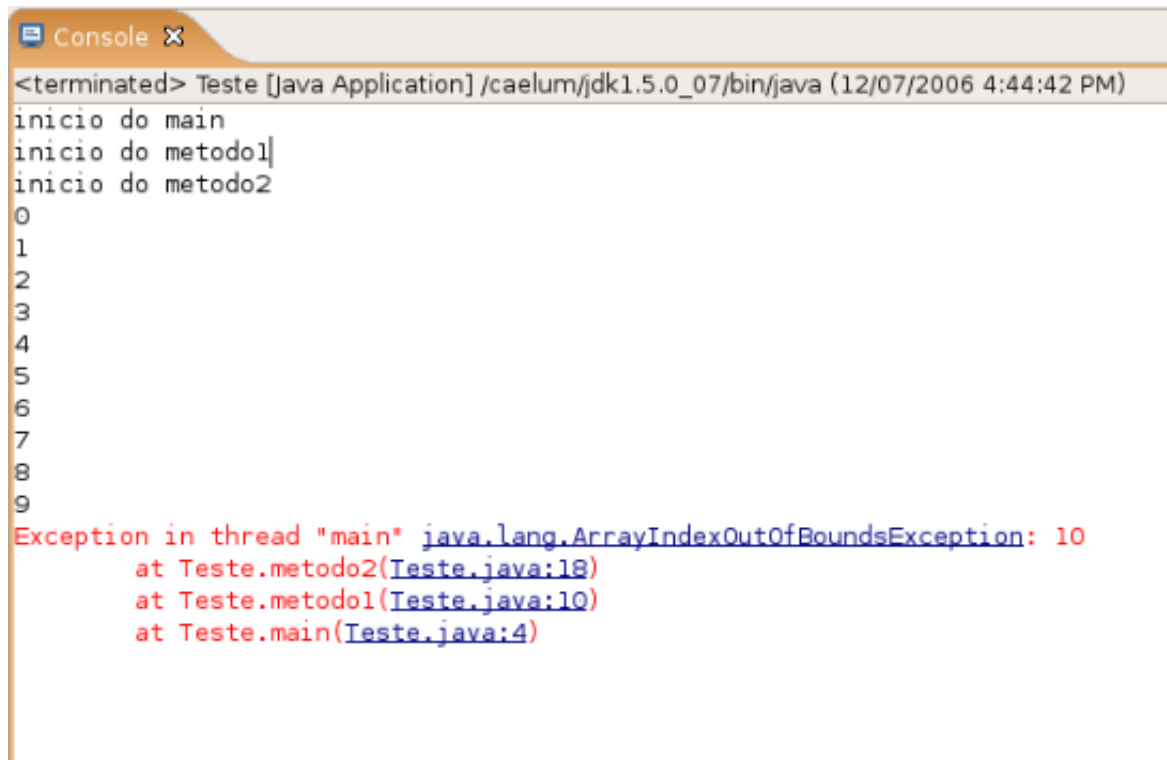
Repare o método main chamando metodo1 e esse, por sua vez, chamando o metodo2. Cada um desses métodos pode ter suas próprias variáveis locais, isto é: o metodo1 não enxerga as variáveis declaradas dentro do main e por aí em diante.

Como o Java (e muitas das outras linguagens) faz isso? 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. Ele descobre isso através da **pilha de execução** (*stack*): basta remover o marcador que está no topo da pilha:



Porém, o nosso metodo2 propositadamente possui um enorme problema: está acessando um índice de array indevido para esse caso; o índice estará fora dos limites da array quando chegar em 10!

Rode o código. Qual é a saída? O que isso representa? O que ela indica?



```
Console X
<terminated> Teste [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:44:42 PM)
inicio do main
inicio do metodo1
inicio 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)
```

Essa é o conhecido **rastro da pilha** (*stacktrace*). É uma saída importantíssima para o programador – tanto que, em qualquer fórum ou lista de discussão, é comum os programadores enviarem, juntamente com a descrição do problema, essa stacktrace. Mas por que isso aconteceu?

O sistema de exceções do Java funciona da seguinte maneira: quando uma exceção é **lançada** (*throw*), a JVM entra em estado de alerta e vai ver se o método atual toma alguma precaução ao **tentar** executar esse trecho de código. Como podemos ver, o `metodo2` não toma nenhuma medida diferente do que vimos até agora.

Como o `metodo2` não está **tratando** esse problema, a JVM pára a execução dele anormalmente, sem esperar ele terminar, e volta um *stackframe* pra baixo, onde será feita nova verificação: "o `metodo1` está se precavendo de um problema chamado `ArrayIndexOutOfBoundsException`?" "Não..." Volta para o `main`, onde também não há proteção, então a JVM morre (na verdade, quem morre é apenas a Thread corrente, veremos mais para frente).

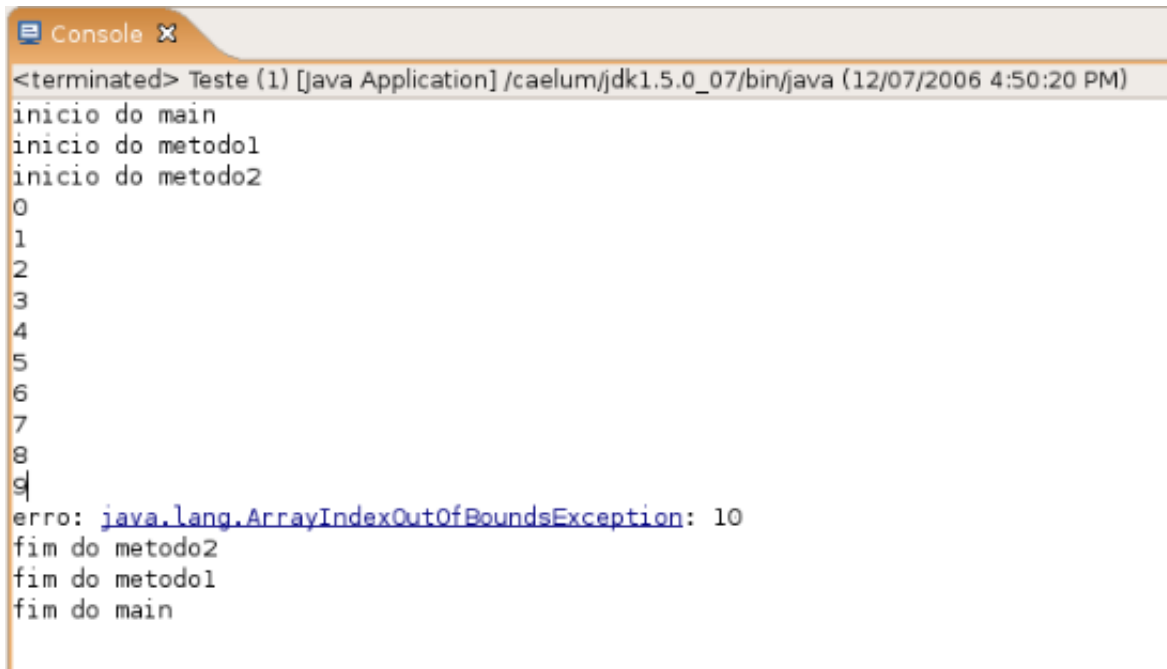
Obviamente, aqui estamos forçando esse caso e não faria sentido tomarmos cuidado com ele. É fácil arrumar um problema desses: basta percorrermos a array no máximo até o seu `length`.

Porém, apenas para entender o controle de fluxo de uma Exception, vamos colocar o código que vai **tentar** (*try*) executar o bloco perigoso e, caso o problema seja do tipo `ArrayIndexOutOfBoundsException`, ele será **pego** (*caught*). Repare que é interessante que cada exceção no Java tenha um tipo... ela pode ter atributos e

métodos.

2. Adicione um try/catch em volta do for, pegando `ArrayIndexOutOfBoundsException`. O que o código imprime?

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

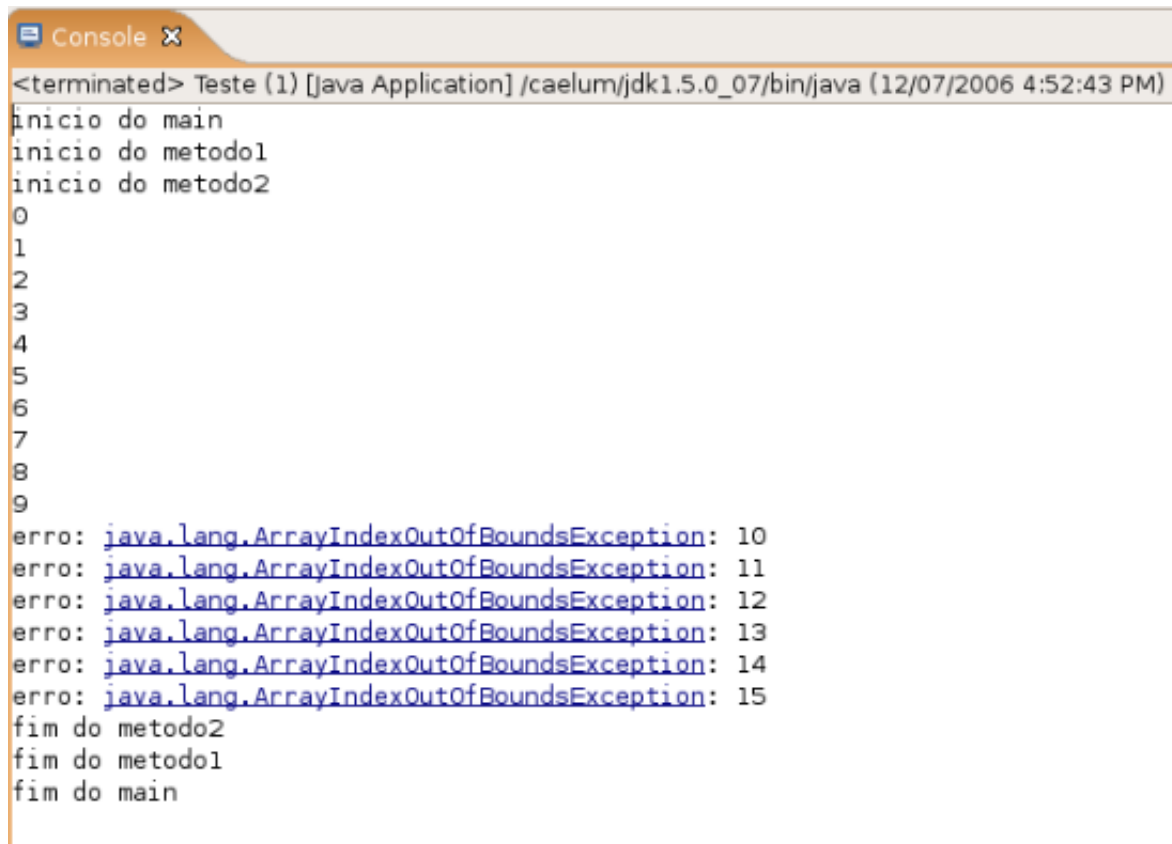


```
Console x  
<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:50:20 PM)  
inicio do main  
inicio do metodo1  
inicio do metodo2  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
erro: java.lang.ArrayIndexOutOfBoundsException: 10  
fim do metodo2  
fim do metodo1  
fim do main
```

3. Em vez de fazer o try em torno do for inteiro, tente apenas com o bloco de dentro do for:

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

Qual é a diferença?



```

<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:52:43 PM)
inicio do main
inicio do metodo1
inicio 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

```

4. Retire o try/catch e coloque ele em volta da chamada do metodo2.

```

System.out.println("inicio do metodo1");
try {
    metodo2();
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("erro: " + e);
}
System.out.println("fim do metodo1");

```



```

<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:56:54 PM)
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

```


5. Faça o mesmo, retirando o try/catch novamente e colocando em volta da chamada do metodo1. Rode os códigos, o que acontece?

```

System.out.println("inicio do main");

```

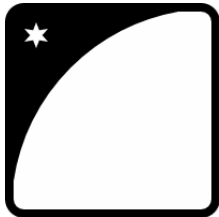
```
try {  
    metodo1();  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("Erro : "+e);  
}  
System.out.println("fim do main");
```



```
<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:59:00 PM)  
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
```

Repare que, a partir do momento que uma exception foi *caught* (pega, tratada, handled), a execução volta ao normal a partir daquele ponto.

Você não está nessa página a toa



Você chegou aqui porque a Caelum é referência nacional em cursos de Java, Ruby, Agile, Mobile, Web e .NET.

Faça curso com **quem escreveu essa apostila**.

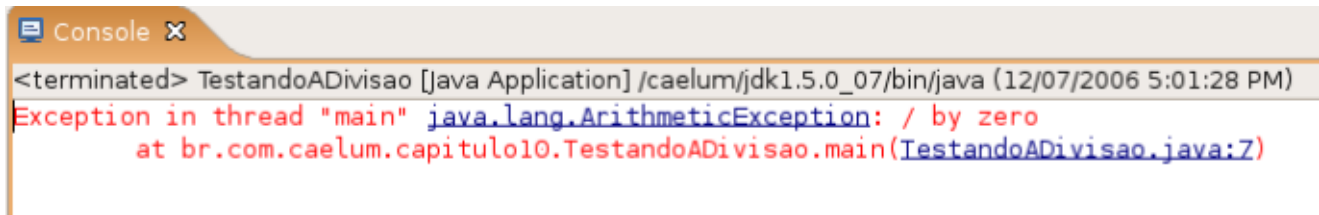
[Consulte as vantagens do curso *Java e Orientação a Objetos*.](http://www.caelum.com.br/apostila-java-orientacao-objetos/excecoes-e-controle-de-erros/)

11.3 – EXCEÇÕES DE RUNTIME MAIS COMUNS

Que tal tentar dividir um número por zero? Será que a JVM consegue fazer aquilo que nós definimos que não existe?

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

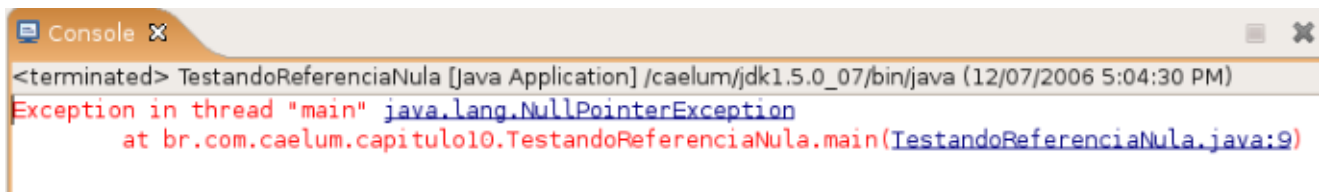

Tente executar o programa acima. O que acontece?



```
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)
```

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

Tente executar este programa. O que acontece?



```
Console X
<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)
```

Repare que um `ArrayIndexOutOfBoundsException` ou um `NullPointerException` poderia ser facilmente evitado com o `for` corretamente escrito ou com `ifs` que checariam os limites da array.

Outro caso em que também ocorre tal tipo de exceção é quando um cast errado é feito (veremos mais pra frente). Em todos os casos, tais problemas provavelmente 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*. Em outras palavras, o compilador não checa se você está tratando essas exceções.

Erros

Os erros em Java são um tipo de exceção que também podem ser tratados. Eles representam problemas na máquina virtual e não devem ser tratados em 99% dos casos, já que provavelmente o melhor a se fazer é deixar a JVM encerrar (ou apenas a Thread em questão).

11.4 – OUTRO TIPO DE EXCEÇÃO: CHECKED EXCEPTIONS

Fica claro, com os exemplos de código acima, que não é necessário declarar que você está tentando fazer algo onde um erro possa ocorrer. Os dois exemplos, com ou sem o try/catch, compilaram e rodaram. Em um, o erro terminou o programa e, no outro, foi possível tratá-lo.

Mas não é só esse tipo de exceção que existe em Java. Um outro tipo, obriga a quem chama o método ou construtor a tratar essa exceção. Chamamos esse tipo de exceção de *checked*, pois o compilador checará se ela está sendo devidamente tratada, diferente das anteriores, conhecidas como *unchecked*.

Um exemplo interessante é o de abrir um arquivo para leitura, onde pode ocorrer o erro do arquivo não existir (veremos como trabalhar com arquivos em outro capítulo, **não** se preocupe com isto agora):

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

O código acima não compila e o compilador avisa 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 compilar e fazer o programa funcionar, temos duas maneiras que podemos tratar o problema. O primeiro, é tratá-lo com o try e catch do mesmo jeito que usamos no exemplo anterior, com uma array:

```
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");  
    }  
}
```

A segunda forma de tratar esse erro, é delegar ele para quem chamou o nosso método, isto é, passar para a frente.

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

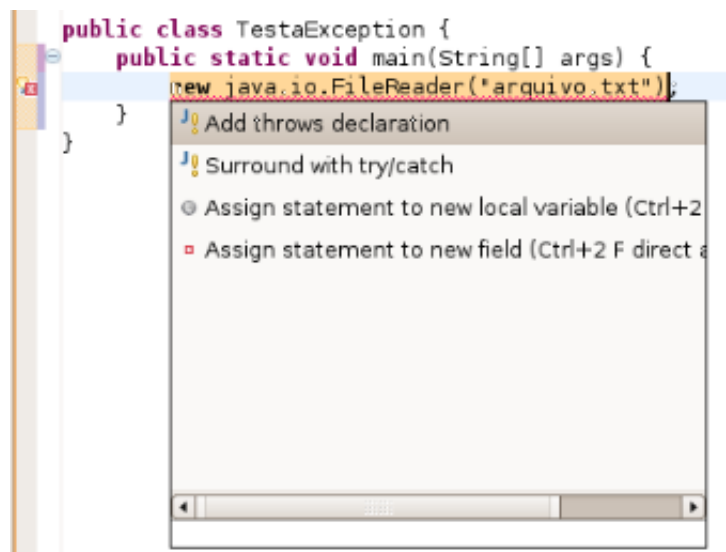
```
}
```

No Eclipse é bem simples fazer tanto um try/catch como um throws:

Tente digitar esse código no eclipse:

```
public class TestaException {
    public static void main(String[] args) {
        new java.io.FileInputStream("arquivo.txt");
    }
}
```

O Eclipse vai reclamar :



E você tem duas opções:

1. *Add throws declaration*, que vai gerar:

```
public class TestaException {
    public static void main(String[] args) throws FileNotFoundException {
        new java.io.FileInputStream("arquivo.txt");
    }
}
```

2. *Surround with try/catch*, que vai gerar:

```
public class TestaException2 {
    public static void main(String[] args) {
        try {
            new java.io.FileInputStream("arquivo.txt");
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

No início, existe uma grande tentação de sempre passar o problema pra frente para outros o tratarem. Pode ser que faça sentido, dependendo do caso, mas não até o main, por exemplo. Acontece que quem tenta abrir um arquivo sabe como lidar com um problema na leitura. Quem chamou um método no começo do programa pode não saber ou, pior ainda, tentar abrir cinco arquivos diferentes e não saber qual deles teve um problema!

Não há uma regra para decidir em que momento do seu programa você vai tratar determinada exceção. Isso vai depender de em que ponto você tem condições de tomar uma decisão em relação àquele erro. Enquanto não for o momento, você provavelmente vai preferir delegar a responsabilidade para o método que te invocou.

Um outro problema comum é quando trabalhamos com banco de dados:

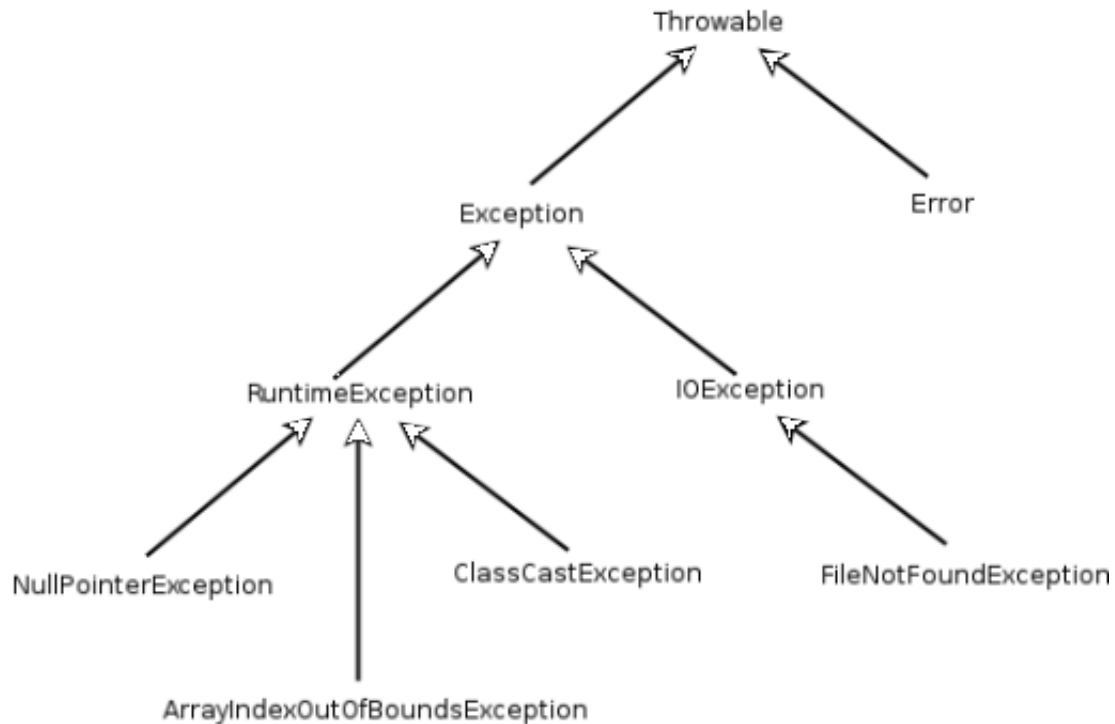
Boas práticas no tratamento de exceções

No blog da Caelum há um extenso artigo discutindo as boas práticas em relação ao tratamento de exceções.

<http://blog.caelum.com.br/2006/10/07/lidando-com-exceptions/>

11.5 – UM POUCO DA GRANDE FAMÍLIA THROWABLE

Uma pequena parte da Família Throwable:



Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

Conheça os títulos e a nova proposta, você vai gostar.

[Casa do Código, livros para o programador.](http://www.casadocodigo.com.br)

11.6 – MAIS DE UM ERRO

É possível tratar mais de um erro quase que ao mesmo tempo:

1. Com o try e catch:

```

try {
    objeto.metodoQuePodeLancarIOeSQLException();
} catch (IOException e) {
    // ..
} catch (SQLException e) {
    // ..
}
  
```

2. Com o throws:

```
public void abre(String arquivo) throws IOException, SQLException {  
    // ..  
}
```

3. Você pode, também, escolher tratar algumas exceções e declarar as outras no throws:

```
public void abre(String arquivo) throws IOException {  
    try {  
        objeto.metodoQuePodeLancarIOeSQLException();  
    } catch (SQLException e) {  
        // ..  
    }  
}
```

É desnecessário declarar no throws as exceptions que são *unchecked*, porém é permitido e às vezes, facilita a leitura e a documentação do seu código.

11.7 – LANÇANDO EXCEÇÕES

Lembre-se do método saca da nossa classe Conta. Ele devolve um boolean caso consiga ou não sacar:

```
boolean saca(double valor) {  
    if (this.saldo < valor) {  
        return false;  
    } else {  
        this.saldo -= valor;  
        return true;  
    }  
}
```

Podemos, também, lançar uma Exception, o que é extremamente útil. Dessa maneira, resolvemos o problema de alguém poder esquecer de fazer um if no retorno de um método.

A palavra chave **throw**, que está no imperativo, lança uma Exception. Isto é bem diferente de throws, que está no presente do indicativo, e que apenas avisa da possibilidade daquele método lançá-la, obrigando o outro método que vá utilizar deste de se preocupar com essa exceção em questão.

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

No nosso caso, lança uma do tipo *unchecked*. `RuntimeException` é a exception mãe de todas as exceptions *unchecked*. A desvantagem, aqui, é que ela é muito genérica; quem receber esse erro não sabe dizer exatamente qual foi o problema. Podemos então usar uma Exception 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` e já faz parte da biblioteca do java.

(`IllegalArgumentException` é a melhor escolha quando um argumento sempre é inválido como, por exemplo, números negativos, referências nulas, etc).

Para pegar esse erro, não usaremos um `if/else` e sim um `try/catch`, porque faz mais sentido já que a falta de saldo é uma exceção:

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

Podíamos melhorar ainda mais e passar para o construtor da `IllegalArgumentException` 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());  
}
```

11.8 – O QUE COLOCAR DENTRO DO TRY?

Imagine que vamos sacar dinheiro de diversas contas:

```
Conta cc = new ContaCorrente();  
cc.deposita(100);  
  
Conta cp = new ContaPoupanca();  
cp.deposita(100);  
  
// sacando das contas:  
  
cc.saca(50);  
System.out.println("consegui sacar da corrente!");  
  
cp.saca(50);  
System.out.println("consegui sacar da poupança!");
```

Podemos escolher vários lugares para colocar try/catch:

```
try {  
    cc.saca(50);  
} catch (IllegalArgumentException e) {  
    System.out.println(e.getMessage());  
}  
System.out.println("consegui sacar da corrente!");  
  
try {  
    cp.saca(50);  
} catch (IllegalArgumentException e) {  
    System.out.println(e.getMessage());  
}  
System.out.println("consegui sacar da poupança!");
```

Essa não parece uma opção boa, pois a mensagem *"consegui sacar"* será impressa mesmo que o catch seja acionado. Sempre que temos algo que depende da linha de cima para ser correto, devemos agrupá-lo no try:

```
try {  
    cc.saca(50);  
    System.out.println("consegui sacar da corrente!");  
} catch (IllegalArgumentException e) {  
    System.out.println(e.getMessage());  
}  
  
try {  
    cp.saca(50);  
    System.out.println("consegui sacar da poupança!");  
} catch (IllegalArgumentException e) {
```



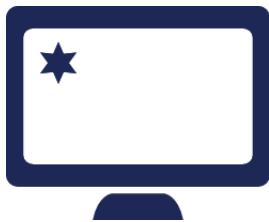
```
System.out.println(e.getMessage());  
}
```

Mas há ainda uma outra opção: imagine que, para o nosso sistema, uma falha ao sacar da conta poupança deve parar o processo de saques e nem tentar sacar da conta corrente. Para isso, agruparíamos mais ainda:

```
try {  
    cc.saca(50);  
    System.out.println("consegui sacar da corrente!");  
    cp.saca(50);  
    System.out.println("consegui sacar da poupança!");  
} catch (IllegalArgumentException e) {  
    System.out.println(e.getMessage());  
}
```

O que você vai colocar dentro do try influencia muito a execução do programa! Pense direito nas linhas que dependem uma da outra para a execução correta da sua lógica de negócios.

Agora é a melhor hora de aprender algo novo



Se você gosta de estudar essa apostila aberta da Caelum, certamente vai gostar dos novos **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum.

[Conheça a Alura.](https://www.caelum.com.br/apostila-java-orientacao-objetos/excecoes-e-controle-de-erros/)

11.9 – CRIANDO SEU PRÓPRIO TIPO DE EXCEÇÃO

É bem comum criar uma própria classe de exceção para controlar melhor o uso de suas exceções. Dessa maneira, podemos passar valores específicos para ela carregar, que sejam úteis de alguma forma. Vamos criar a nossa:

Voltamos para o exemplo das Contas, vamos criar a nossa Exceção de SaldoInsuficienteException:

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

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

```
void saca(double valor) {  
    if (this.saldo < valor) {  
        throw new SaldoInsuficienteException("Saldo Insuficiente," +  
                                              "tente um valor menor");  
    } else {  
        this.saldo -= valor;  
    }  
}
```

E, para testar, crie uma classe que deposite um valor e tente sacar um valor maior:

```
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 Exception de *unchecked* para *checked*, obrigando a quem chama esse método a dar try-catch, ou throws:

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

11.10 – PARA SABER MAIS: FINALLY

Os blocos try e catch podem conter uma terceira cláusula chamada *finally* que indica o que deve ser feito após o término do bloco try ou de um catch qualquer.

É interessante colocar algo que é imprescindível de ser executado, caso o que você queria fazer tenha dado certo, ou não. O caso mais comum é o de liberar um recurso no finally, como um arquivo ou conexão com banco de dados, para que possamos ter a certeza de que aquele arquivo (ou conexão) vá ser fechado, mesmo que algo tenha falhado no decorrer do código.

No exemplo a seguir, o bloco *finally* será sempre executado,

independentemente de tudo ocorrer bem ou de acontecer algum problema:

```
try {  
    // bloco try  
} catch (IOException ex) {  
    // bloco catch 1  
} catch (SQLException sqlex) {  
    // bloco catch 2  
} finally {  
    // bloco que será sempre executado, independente  
    // se houve ou não exception e se ela foi tratada ou não  
}
```

Há também, no Java 7, um recurso poderoso conhecido como *try-with-resources*, que permite utilizar a semântica do `finally` de uma maneira bem mais simples, como veremos no capítulo

11.11 – EXERCÍCIOS: EXCEÇÕES

1. Na classe `Conta`, modifique o método `deposita(double x)`: Ele deve lançar uma exception chamada `IllegalArgumentException`, que já faz parte da biblioteca do Java, sempre que o valor passado como argumento for inválido (por exemplo, quando for negativo).

```
public void deposita(double valor) {  
    if (valor < 0) {  
        throw new IllegalArgumentException();  
    } else {  
        this.saldo += valor;  
    }  
}
```

2. Crie uma classe `TestaDeposita` com o método `main`. Crie uma `ContaPoupanca` e tente depositar valores inválidos:

```
public static void main(String[] args) {  
    Conta cp = new ContaPoupanca();  
    cp.deposita(-100);  
}
```

O que acontece?

Atenção: estamos usando a `ContaPoupanca` porque, em capítulos anteriores, sobrescrevemos o método `deposita` da `ContaCorrente`, provavelmente sem utilizar o `super.deposita`. Assim, ela não se comportará igual à `ContaPoupanca` no caso do valor negativo! Você pode resolver isso utilizando o `super.deposita` ou fazendo apenas o teste com `ContaPoupanca`.

3. Adicione o try/catch para tratar o erro:

```
public static void main(String[] args) {  
    Conta cp = new ContaPoupanca();  
  
    try {  
        cp.deposita(-100);  
    } catch (IllegalArgumentException e) {  
        System.out.println("Você tentou depositar um valor inválido");  
    }  
}
```

4. Ao lançar a `IllegalArgumentException`, passe via construtor uma mensagem a ser exibida. Lembre que a `String` recebida como parâmetro é acessível depois via o método `getMessage()` herdado por todas as `Exceptions`.

```
public void deposita(double valor) {  
    if (valor < 0) {  
        throw new IllegalArgumentException("Você tentou depositar" +  
                                           " um valor negativo");  
    } else {  
        this.saldo += valor - 0.10;  
    }  
}
```

Com isso, você precisará fazer algumas mudanças no `TestaDeposita`. Como ficou ele, depois de alterado?

5. Crie sua própria `Exception`, `ValorInvalidoException`. Para isso, você precisa criar uma classe com esse nome que seja filha de `RuntimeException`.

```
public class ValorInvalidoException extends RuntimeException {  
  
}
```

Lance-a em vez de `IllegalArgumentException`. Quais alterações você teve que fazer nas classes `Conta` e na `TestaDeposita`?

Atenção: nem sempre é interessante criarmos um novo tipo de `exception`! Depende do caso. Neste aqui, seria melhor ainda utilizarmos `IllegalArgumentException`. A boa prática diz que devemos preferir usar as já existentes do Java sempre que possível.

6. (opcional) Coloque um construtor na classe `ValorInvalidoException` que receba valor inválido que ele tentou passar (isto é, ele vai receber um `double` valor).

Quando estendemos uma classe, não herdamos seus construtores, mas podemos acessá-los através da palavra chave `super` de dentro de um construtor. As

exceções do Java possuem uma série de construtores úteis para poder populá-las já com uma mensagem de erro. Então vamos criar um construtor em `ValorInvalidoException` que delegue para o construtor de sua mãe. Essa vai guardar essa mensagem para poder mostrá-la ao ser invocado o método `getMessage`:

```
public class ValorInvalidoException extends RuntimeException {  
  
    public ValorInvalidoException(double valor) {  
        super("Valor invalido: " + valor);  
    }  
}
```

Dessa maneira, na hora de dar o `throw new ValorInvalidoException` você vai precisar passar esse valor como argumento:

```
if (valor < 0) {  
    throw new ValorInvalidoException(valor);  
}
```

Atenção: você pode se aproveitar do Eclipse para isso: comece já passando o `valor` como argumento para o construtor da exception e o Eclipse vai reclamar que não existe tal construtor. O quick fix (`ctrl + 1`) vai sugerir que ele seja construindo, poupando-lhe tempo!

E agora, como fica a classe `TestaDeposita`?

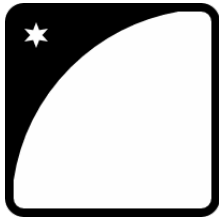
7. (opcional) Declare a classe `ValorInvalidoException` como filha direta de `Exception` em vez de `RuntimeException`. Ela passa a ser **checked**. O que isso resulta?

Você vai precisar avisar que o seu método `deposita()` throws `ValorInvalidoException`, pois ela é uma *checked* exception. Além disso, quem chama esse método vai precisar tomar uma decisão entre `try-catch` ou `throws`. Faça uso do quick fix do Eclipse novamente!

Depois, retorne a exception para *unchecked*, isto é, para ser filha de `RuntimeException`, pois utilizaremos ela assim em exercícios dos capítulos posteriores.

Você pode também fazer o curso FJ-11 dessa apostila na Caelum

Querendo aprender ainda mais sobre Java e boas práticas de orientação a objetos? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas



com um instrutor?

A Caelum oferece o **curso FJ-11** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

[Consulte as vantagens do curso *Java e Orientação a Objetos*.](#)

11.12 – DESAFIOS

1. O que acontece se acabar a memória da java virtual machine? Como forçar isso?

11.13 – DISCUSSÃO EM AULA: CATCH E THROWS EM EXCEPTION

Existe uma péssima prática de programação em java que é a de escrever o `catch` e o `throws` com `Exception`.

Existem códigos que sempre usam `Exception` pois isso cuida de todos os possíveis erros. O maior problema disso é generalizar o erro. Se alguém joga algo do tipo `Exception` para quem o chamou, quem recebe não sabe qual o tipo específico de erro ocorreu e não vai saber como tratar o mesmo.

Sim, há casos onde o tratamento de mais de uma exception pode ser feito de uma mesma maneira. Por exemplo, se queremos terminar a aplicação tanto no caso de `IOException` quanto em `SQLException`. Se fizermos `catch(Exception e)` para pegar esses dois casos, teremos um problema: a aplicação vai parar mesmo que outra exceção seja lançada. A solução correta seria ter dois catches, mas aí teríamos código repetido. Para evitar o código repetido, podemos usar o multi-catch do Java 7, que permite um mesmo catch cuidar de mais de 1 exceção, através da sintaxe `catch(IOException | SQLException e) { ... } .`

CAPÍTULO ANTERIOR:

[Interfaces](#)

PRÓXIMO CAPÍTULO:

[Pacotes – Organizando suas classes e bibliotecas](#)

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter