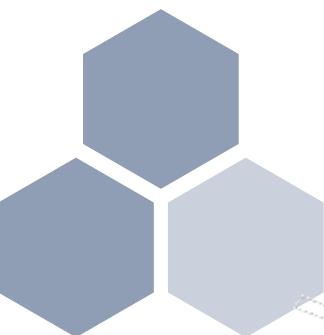




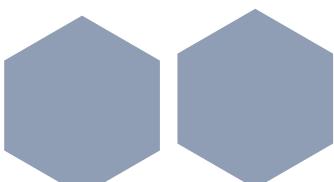
Java Programmer

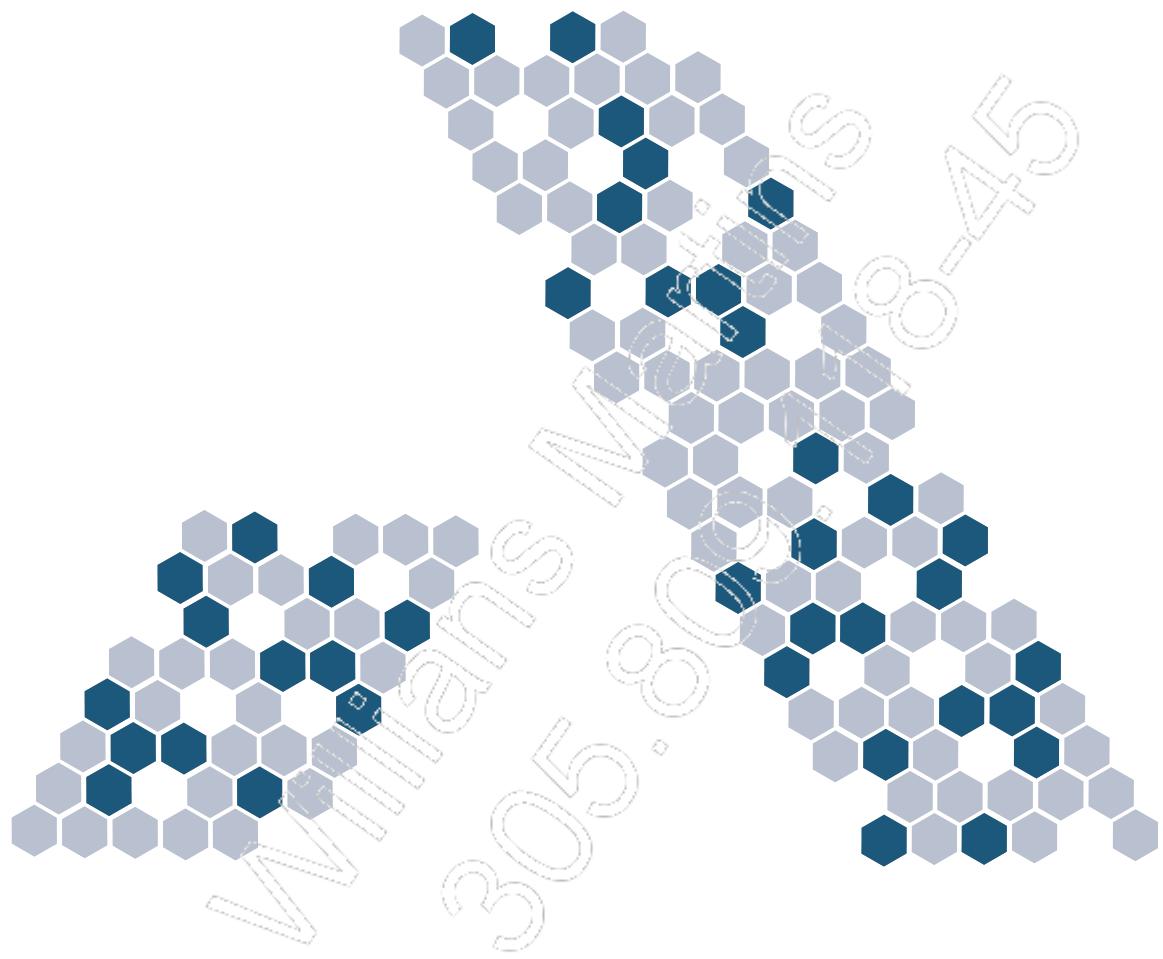
Parte II

Williams Martins
305.809.7451



Editora
IMPACTA



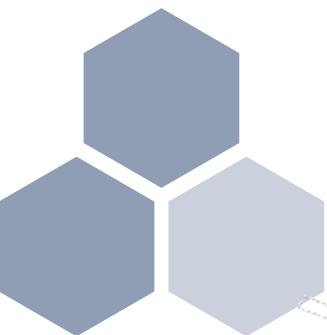




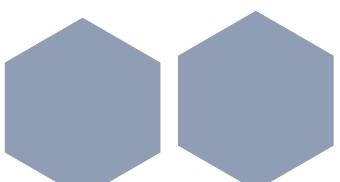
Java Programmer

Parte II

Williams Martins
305.809.7455



Editora
IMPACTA



Créditos

Copyright © Monte Everest Participações e Empreendimentos Ltda.

Todos os direitos autorais reservados. Este manual não pode ser copiado, fotocopiado, reproduzido, traduzido ou convertido em qualquer forma eletrônica, ou legível por qualquer meio, em parte ou no todo, sem a aprovação prévia, por escrito, da Monte Everest Participações e Empreendimentos Ltda., estando o contrafator sujeito a responder por crime de Violação de Direito Autoral, conforme o art.184 do Código Penal Brasileiro, além de responder por Perdas e Danos. Todos os logotipos e marcas utilizados neste material pertencem às suas respectivas empresas.

"As marcas registradas e os nomes comerciais citados nesta obra, mesmo que não sejam assim identificados, pertencem aos seus respectivos proprietários nos termos das leis, convenções e diretrizes nacionais e internacionais."

Java Programmer Parte II

Coordenação Geral

Marcia M. Rosa

Coordenação Editorial

Henrique Thomaz Bruscagin

Autoria

Braulio Consani Moura

Revisão Ortográfica e Gramatical

Marcos Cesar dos Santos Silva

Diagramação

Paloma da Silva Teixeira

Edição nº 1 | 1842_0

Março/ 2018

Este material constitui uma nova obra e é uma derivação da seguinte obra original, produzida por TechnoEdition Editora Ltda., em Set/2014: Java Programmer.

Autoria: Sandro Luiz de Souza Vieira

Sumário

Capítulo 11 - Tratamento de exceções	9
11.1. Introdução	10
11.2. Bloco try/catch.....	10
11.2.1. Manipulando mais de um tipo de exceção.....	12
11.3. throws	13
11.4. finally	13
11.5. Exceções e a pilha de métodos em Java.....	14
11.6. Hierarquia de exceções	16
11.6.1. Exceções verificadas	16
11.6.2. Exceções não verificadas.....	16
11.7. Principais exceções	17
11.7.1. Throwable.....	17
11.7.1.1.Exceções encadeadas	19
11.7.2. Error	20
11.7.3. Exception.....	20
11.7.4. NullPointerException.....	21
11.7.5. NumberFormatException.....	21
11.7.6. ArrayIndexOutOfBoundsException	22
11.7.7. ArithmeticException.....	22
11.7.8. ClassCastException.....	23
11.7.9. IOException	23
11.7.10.Classe SQLException	24
11.8. Exceções personalizadas.....	25
Pontos principais	28
Teste seus conhecimentos.....	29
Mãos Obra!.....	31
Capítulo 12 - As bibliotecas Java e o Javadoc	33
12.1. Conceito de API.....	34
12.2. Javadoc e a documentação oficial Java	34
12.3. Criação de uma documentação Javadoc.....	36
12.3.1. Geração da página de documentação	36
Pontos principais	37
Teste seus conhecimentos.....	39
Capítulo 13 - Testes unitários com Java.....	41
13.1. Conceito de teste unitário	42
13.2. Como implantar o teste unitário.....	42
13.3. Utilizando o JUnit	42
13.3.1. Criando um teste unitário	43
13.3.1.1.Ciclo de vida de um teste	47
13.3.1.2.Assertions	47
13.4. Conclusão.....	50
Pontos principais	51
Teste seus conhecimentos.....	53

Java Programmer - Parte II

Capítulo 14 - Programação funcional.....	55
14.1. Introdução	56
14.1.1. Vantagens da programação funcional.....	56
14.1.2. Um primeiro exemplo	56
14.2. Interface funcional.....	59
14.2.1. A anotação @FunctionalInterface.....	59
14.2.2. Exemplos de interface funcional.....	60
14.3. Expressões lambda	60
14.3.1. Forma geral	61
14.3.2. Expressões com parâmetros	61
14.3.3. Expressões sem parâmetros.....	62
14.3.4. Expressões com um único parâmetro.....	62
14.3.5. Corpo da expressão lambda.....	63
14.3.6. Expressões com valor de retorno	64
14.4. Referenciando métodos	65
14.5. O pacote java.util.function	68
Pontos principais	69
Teste seus conhecimentos.....	71
Mãos Obra!.....	73
Capítulo 15 - Coleções e mapas.....	77
15.1. O que são coleções	78
15.2. Principais operações de coleções	79
15.3. Principais interfaces das coleções	79
15.3.1. Características das classes de implementação.....	80
15.4. Generics	81
15.4.1. Tipos genéricos	82
15.5. Coleção List	84
15.6. Coleção Set	86
15.6.1. Classe Iterator.....	87
15.6.2. Equivalência de objetos (equals).....	89
15.6.2.1. As regras de equals().....	90
15.6.3. Hashing	91
15.6.3.1. As regras de hashCode().....	92
15.6.4. Método forEach().....	93
15.6.5. Método removeIf().....	94
15.6.6. Interface Comparable	95
15.6.7. Interface Comparator	95
15.7. Manipulando coleções com Streams	96
15.7.1. Método sorted()	98
15.7.2. Método filter()	101
15.7.3. Método limit()	102
15.7.4. Método skip()	103
15.7.5. Método map()	104
15.7.6. Método distinct()	106
15.7.7. Método count()	107
15.7.8. Métodos min() e max()	108
15.8. Interface Map	109
15.8.1. Principais métodos	109
15.9. Collections Framework	112
Pontos principais	113
Teste seus conhecimentos.....	115
Mãos Obra!.....	119
Projeto Prático - Fase 3	123

Sumário

Capítulo 16 - Arquivos - I/O e NIO.....	125
16.1. I/O.....	126
16.1.1. Classe OutputStream	126
16.1.1.1. Métodos	128
16.1.2. Classe InputStream	129
16.1.2.1. Métodos	129
16.1.3. Leitura de arquivos binários	131
16.1.4. I/O - Arquivos e diretórios (classe File)	131
16.2. try-with-resources	132
16.2.1. Exceções suprimidas.....	133
16.3. Leitura de arquivos de texto.....	133
16.3.1. Classe FileReader	133
16.3.2. Classe BufferedReader	134
16.4. NIO - Arquivos e diretórios	134
16.4.1. Visão Geral de NIO	134
16.4.2. Path, Paths e Files	135
Pontos principais	139
Teste seus conhecimentos.....	141
Mãos Obra!.....	143
Projeto Prático - Fase 4	147
 Capítulo 17 - Threads.....	149
17.1. Introdução	150
17.2. Programação multithreaded	150
17.2.1. Multitarefa baseada em processo	151
17.2.2. Multitarefa baseada em threads	152
17.3. Implementando multithreading	152
17.3.1. java.lang.Thread	153
17.3.2. java.lang.Runnable.....	155
17.4. Construtores	156
17.5. Estados da thread	158
17.6. Scheduler.....	159
17.7. Prioridades das threads.....	159
17.7.1. Método yield()	162
17.7.2. Método join()	162
17.7.3. Método isAlive()	163
17.7.4. Método sleep()	163
17.8. Sincronização.....	165
17.8.1. Palavra-chave synchronized.....	166
17.8.1.1. Race condition.....	166
17.8.2. Bloco sincronizado.....	167
17.9. Bloqueios	168
17.10. Deadlock	170
17.11. Intereração entre threads	171
Pontos principais	176
Teste seus conhecimentos.....	179
Mãos Obra!.....	183



Java Programmer - Parte II

Capítulo 18 - Geração de Pacotes - Instalação de Aplicações Java (JAR).....	187
18.1. Conceito de aplicações e bibliotecas	188
18.2. Geração de bibliotecas e executáveis	188
18.2.1. Geração de um pacote executável	189
18.2.2. Utilização de uma biblioteca em projetos	191
Pontos principais	192
 Teste seus conhecimentos.....	193
Capítulo 19 - Banco de dados com Java - JDBC.....	195
19.1. Introdução	196
19.2. Pacote java.sql	197
19.3. Conexões com banco de dados	197
19.3.1. Estabelecendo uma conexão	197
19.3.2. Interface Connection	198
19.3.3. Classe DriverManager	199
19.3.4. Estabelecendo a conexão com o banco de dados	200
19.3.5. Método Close	202
19.4. Operações na base de dados	203
19.5. Operações parametrizadas	205
19.6. Transações	207
19.7. Consultas.....	208
Pontos principais	210
 Teste seus conhecimentos.....	211
 Mãos Obra!.....	213
 Projeto Prático - Fase 5	219
Apêndice I.....	221
Apêndice II	229
Apêndice III	257
Apêndice IV.....	277

11

Tratamento de exceções

- Bloco try/catch;
- throws;
- finally;
- Exceções e a pilha de métodos em Java;
- Hierarquia de exceções;
- Principais exceções;
- Exceções personalizadas.

11.1. Introdução

Normalmente, os aplicativos que desenvolvemos possuem diversas tarefas a serem executadas. É possível que, em algum momento, uma delas gere um erro ou exceção, isto é, uma ocorrência que faça com que o fluxo normal do programa seja alterado. Isso pode ocorrer por diversos motivos, como uma falha de hardware, por exemplo. Um erro ou exceção pode ser simples ou complexo: em alguns casos, a execução do programa continua, em outros, ela pode ser interrompida.

Por isso, a detecção e a manipulação de erros são fundamentais quando desenvolvemos aplicativos. Na linguagem Java, há um mecanismo que ajuda a produzir código de manipulação organizado e eficiente: a manipulação de exceções. Com ela, os erros em tempo de execução são detectados facilmente, sem que seja necessário desenvolver código especial para que os valores retornados sejam testados.

11.2. Bloco try/catch

Encerrar o código que pode gerar exceções dentro de um bloco **try** é a primeira etapa para criar um manipulador de exceções. A estrutura de um bloco **try** é a seguinte:

```
try {  
    bloco de código  
}  
blocos catch e finally. . .
```

O código encerrado no **try** é representado por **bloco de código**.

Você pode fechar seu código dentro de um bloco **try** de duas maneiras:

- Colocando cada linha do código separada em um bloco **try** próprio, gerando, assim, um manipulador de exceção para cada linha. Veja um exemplo:

```
int i;  
  
try{  
    i = 1 / 0; // exceção por dividir por zero  
    System.out.println(i);  
}catch(Exception e){  
    System.out.println(e);  
}  
  
try{  
    i = 2;  
    System.out.println(i);  
}catch(Exception e){  
    System.out.println(e);  
}
```

- Colocando o código inteiro dentro de um único bloco **try**, gerando diversos manipuladores associados a ele. Veja um exemplo:

```
try{
    int i;
    i = 1 / 0; // exceção por dividir por zero
    i = 2;
    System.out.println(i);
}catch(Exception e){
    System.out.println(e);
}
```

Encerrando o código em um bloco **try**, as possíveis exceções que forem lançadas serão tratadas por um manipulador de exceções associado a elas.

Para associar um manipulador de exceções a um bloco **try**, é preciso inserir um bloco **catch** imediatamente após o bloco **try**, sem nenhum código entre eles. Veja a seguir:

```
try {
    }catch(tipo_de_exceção identificador){
    }catch(tipo_de_exceção identificador){
}
```

Cada bloco **catch** representa um manipulador próprio para um tipo de exceção e contém o código que será executado caso o manipulador seja chamado. O argumento **tipo_de_exceção** representa o tipo de exceção que o manipulador pode tratar. Esse argumento deve consistir no nome de uma classe que herda da classe **Throwable**. O argumento **identificador** indica o nome da própria instância da exceção.

O manipulador de exceções é chamado em tempo de execução quando ele é o primeiro na pilha de chamadas cujo argumento **tipo_de_exceção** corresponde ao tipo da exceção lançada. Com um manipulador de exceções, é possível interromper o programa, exibir mensagens de erro, fazer recuperações de erro, reproduzir o erro em um manipulador de nível mais alto por meio das exceções encadeadas e levar o usuário a tomar decisões.

Veja um exemplo de manipulação de exceções:

```
int i;

try{
    i = 1 / 0; // exceção por dividir por zero
    System.out.println("Valor de i é " + i);
}catch(ArithmetricException e){
    System.out.println("Tratando exceção:");
    e.printStackTrace();
}

System.out.println("Continuação do código.");
```

O resultado será o seguinte:

```
Tratando exceção:  
java.lang.ArithmetricException: / by zero  
    at teste.main(teste.java:8)  
Continuação do código.
```

Em um programa onde diversos tipos diferentes de exceção estão sendo capturados por meio de diversos blocos **catch**, a ordem em que esses blocos são posicionados abaixo do bloco **try** influencia no comportamento desse desvio. Caso as diversas exceções tenham relacionamento por herança entre si, devem ser declaradas do tipo mais específico para o tipo mais genérico, ou seja, as exceções que representam tipos de classes pai na árvore de exceções devem vir por último na ordem das declarações **catch**.

11.2.1. Manipulando mais de um tipo de exceção

A partir de Java 7, é possível manipular mais de um tipo de exceção com um único bloco **catch**, o que reduz a duplicação de código e as tentativas de capturar uma exceção muito extensa. Essa facilidade ajuda o desenvolvedor que precisa capturar diversos tipos de exceção, porém, aplicando o mesmo código de tratamento a todos os casos.

Para manipular diferentes exceções, basta definir os tipos de exceção, separados por uma barra vertical (conhecido como pipe em ambientes Unix-like). Veja um exemplo:

```
catch (IOException | ArithmetricException ex) {  
    System.out.println(ex);  
}
```

Quando um **catch** manipula mais de um tipo de exceção, o parâmetro do bloco é, de forma implícita, final. No exemplo, o parâmetro **ex** é final e, portanto, não aceita que sejam atribuídos valores a ele dentro do **catch**.

Caso a classe definida em **catch** possua subclasses, o **catch** captura qualquer exceção que tenha subclasses dessa classe que foi definida. Se a classe não possuir subclasses, apenas a própria classe definida em **catch** é capturada.

Recomenda-se que não seja desenvolvido apenas um manipulador para capturar todas as exceções lançadas. Além disso, tipos de exceções que possuam relação de herança não devem ser usados em declarações de blocos **multi-catch**. Caso haja a necessidade, por exemplo, de se capturar **FileNotFoundException** (herda de **IOException**) e **IOException**, não o faça em um bloco único, use blocos separados para esse fim.

11.3. throws

Para indicar que um método pode gerar uma exceção, utilizamos o comando **throws** na sua declaração. Isso é necessário, por exemplo, em uma operação com arquivos. Considere que o programa espera um nome de arquivo qualquer para abri-lo e apresentar o conteúdo na tela. Se o usuário informar o nome do arquivo errado, ou mesmo se não digitá-lo, poderá ser gerada uma interrupção do programa.

Em casos assim, é necessário declarar as exceções que podem ocorrer durante a utilização do método para que sejam tratadas. Isso é feito com o comando **throws**, cuja utilização você vê a seguir:

```
public static void abrirArquivo() throws FileNotFoundException{
    File file = new File("arquivo.txt");
    InputStream is = new FileInputStream(file);
}
```

Esse comando é colocado após a declaração do método seguido por um ou vários tipos de exceções, separados por vírgula, conforme a necessidade. Isso permitirá ao método utilizar código que potencialmente pode lançar todas as exceções declaradas.

11.4. finally

Um bloco **finally** (e as instruções contidas nele) é sempre executado após a finalização de um bloco **try**, mesmo quando ocorre uma exceção inesperada.

Uma boa prática para o bloco **finally** é colocar um código de limpeza ou depuração dentro dele. Assim, você garante que esse código será sempre executado e impede que ele seja eventualmente ignorado na execução por causa de um **return**, **break** etc. Isso pode ser feito sem problemas, mesmo quando não houver uma exceção prevista.

Há dois casos em que a execução de um bloco **finally** pode não ocorrer: um é quando a JVM sai de um bloco **try** ou **catch** enquanto ele é executado; o outro é quando a thread que executa o **try** ou **catch** é interrompida. Neste caso, o **finally** pode não executar, mesmo que a aplicação continue como um todo.

A seguir, você confere um exemplo do uso de **finally**:

```
public static void abrirArquivo() throws IOException{
    File file = null;
    InputStream is = null;
    try{
        file = new File("arquivo.txt");
        is = new FileInputStream(file);
    }finally{
        is.close(); // fecha o InputStream
    }
}
```

Usar o **finally** é importante também para evitar o desperdício de recursos, podendo ser usado para fechar um arquivo ou recuperar recursos. Basta colocar o código correspondente dentro do **finally** e assegurar que os recursos sejam sempre recuperados.

O uso do bloco **finally** é opcional. Em determinadas situações, o bloco **catch** também não é necessário, por isso, pode ser que encontremos um bloco **try** seguido de um bloco **finally** em um código.

11.5. Exceções e a pilha de métodos em Java

Para chegar ao método atual, o programa deve executar uma cadeia de métodos, que é conhecida como pilha de chamadas (ou call stack). O último método acessado é colocado no topo da pilha e, consequentemente, os métodos acessados anteriormente ficam posicionados antes, no final da pilha. Essa ordem, no entanto, pode ser invertida.

Ao exibirmos o estado da pilha em um determinado momento, realizamos o chamado rastreamento de pilha. O primeiro método é aquele que está sendo executado no momento. Quando uma exceção é lançada, o primeiro método deve capturá-la. Caso ele não a capture, a exceção passará para o método seguinte. Isso se repetirá até que a exceção chegue ao final da pilha, ou, então, até que ela seja capturada por um dos métodos da pilha. Quando isso ocorre, temos a propagação da exceção.

Se a exceção chegar ao final da pilha, a execução do programa é interrompida. Caso exista uma descrição da exceção disponível, ela é apresentada e a pilha de chamadas, então, é descartada. Essa descrição é importante porque possui informações que nos auxiliam a depurar o aplicativo.

Veja, a seguir, a pilha de chamada de métodos de Java:

- 1) A pilha de chamadas enquanto o método C() estiver sendo executado.



Método B chama o método C

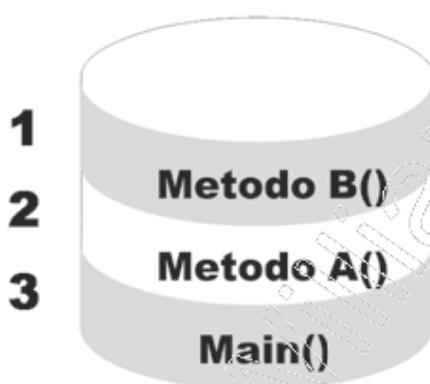
Método A chama o método B

Main chama o método A

Main é iniciado

A ordem na qual os métodos são inseridos na pilha de chamadas.

- 2) A pilha de chamadas depois da conclusão do método C()
A execução retornará ao método B()



Método B() será concluído

Método A () será concluído

Conclusão de Main() JVM sai da pilha

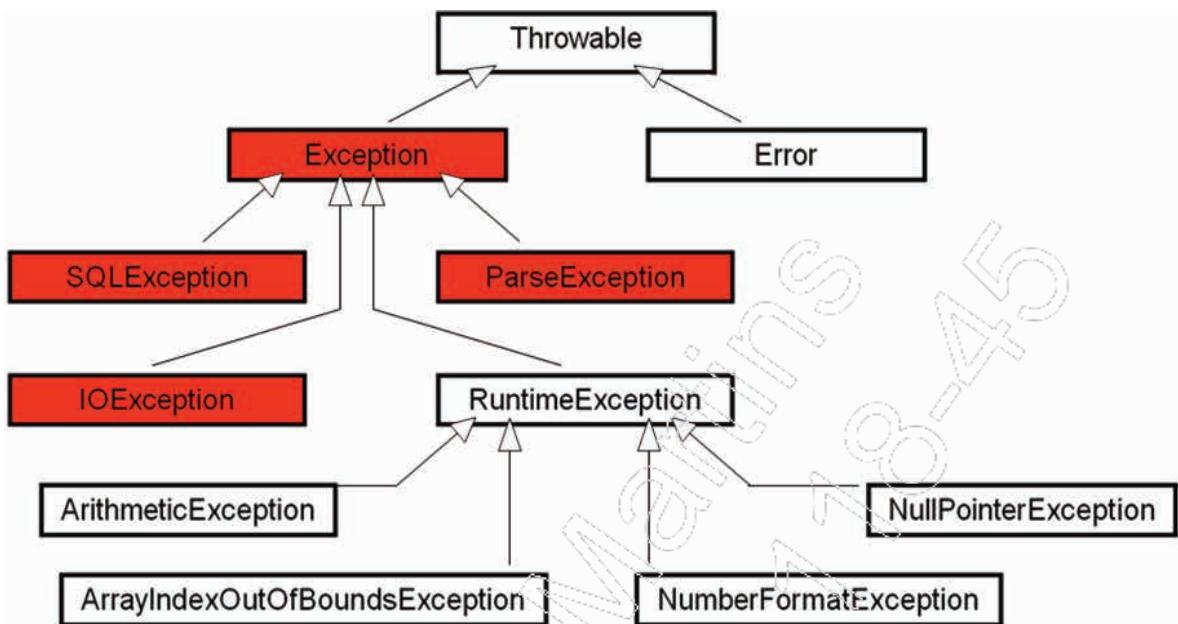
A ordem de conclusão dos métodos.

A exceção pode ser lançada para outros métodos da pilha, mas também podemos lançá-la fora do método **main()**, no final da pilha. Neste caso, a JVM é interrompida e o rastreamento da pilha é apresentado na saída.



11.6. Hierarquia de exceções

Na linguagem Java, as exceções são objetos e os tipos de exceções são representados por classes. Estas classes obedecem a uma hierarquia bem definida. Existem centenas de classes de exceções nativas do Java. O diagrama a seguir mostra algumas das principais:



Quando utilizamos o `try/catch` ou `throws` para tratar um tipo de exceção, automaticamente estamos tratando todos os seus subtipos.

No diagrama, podemos destacar dois tipos de exceções: as verificadas e as não verificadas, explicadas a seguir.

11.6.1. Exceções verificadas

Para confirmar se as exceções foram declaradas ou manipuladas, o compilador verifica o código. As exceções verificadas correspondem às aquelas derivadas da classe `java.lang.Exception` e que não derivam de `java.lang.RuntimeException`. Estas exceções precisam ser capturadas (tratadas) em alguma parte do código do programa. Caso a exceção não seja capturada ou tratada, quando um método que a lança for chamado, o código não será compilado.

11.6.2. Exceções não verificadas

Correspondem às exceções `Error`, `RuntimeException` e seus subtipos. Elas não precisam de definição ou de manipulação. Quando uma exceção `RuntimeException` é declarada em um método, ela também não precisa ser manipulada ou declarada pelo método que a chamou.

11.7. Principais exceções

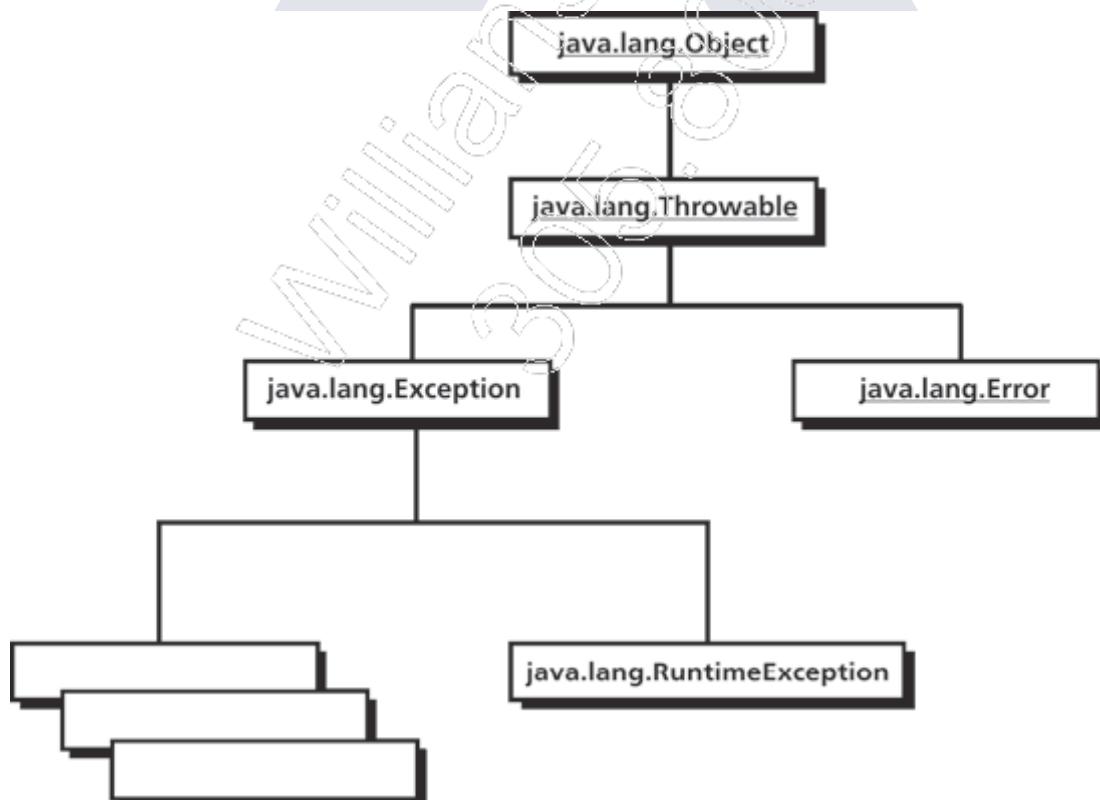
Várias exceções verificadas e não verificadas são lançadas pelos métodos existentes nas bibliotecas de Java e, conforme já mencionado, existem centenas de exceções nativas do Java. No entanto, algumas destas exceções destacam-se por serem de uso comum.

No desenvolvimento de aplicações Java, não há a necessidade de “decorar” ou conhecer todas as exceções nativas da linguagem, pois o próprio compilador Java menciona as que precisam ser manipuladas.

11.7.1. Throwable

Hierarquicamente, esta é a classe base de todas as exceções utilizadas pelo Java. Nesta classe, encontramos as subclasses **Error** e **Exception**, que são utilizadas de maneira convencional para indicar que situações incomuns ocorreram. Como você viu, em uma cláusula **catch**, um dos argumentos pode ser **Throwable** ou uma de suas subclasses.

Entre os objetos que herdam dessa classe, estão os descendentes diretos e os indiretos, que são aqueles que herdam a partir de descendentes diretos da classe. As subclasses **Error** e **Exception** são as duas descendentes imediatas de **Throwable**. Veja, a seguir, como se estrutura a hierarquia de **Throwable**:



A JVM lança somente objetos que são instâncias da própria classe **Throwable** ou de suas subclasses. Quando uma situação incomum ocorre, as instâncias são criadas para fornecerem informações importantes.

Em **Throwable**, você pode encontrar os seguintes itens:

- Uma string de mensagem, cuja função é fornecer informações a respeito do erro ocorrido;
 - Esta mensagem pode ser recuperada a partir do método **getMessage**.
- Um snapshot de pilha de execução da cadeia, que é adicionado à classe **Throwable** quando ela é criada;
 - Esta pilha pode ser impressa na saída padrão através da chamada do método **printStackTrace**.
- Um recurso **cause**, que diz respeito às exceções encadeadas.
 - A exceção causadora de outra pode ser recuperada a partir do método **getCause**.

11.7.1.1. Exceções encadeadas

Muitas vezes, uma exceção pode causar outra exceção, ou seja, um programa pode lançar uma segunda exceção como resposta à primeira. Para saber quando isso ocorre, você pode usar o recurso de encadeamento de exceções.

Veja um exemplo:



```

1  public class teste {
2      public static void main(String args[]) throws Exception{
3          try{
4              metodo1();
5          }catch(Exception e){
6              e.printStackTrace();
7          }
8      }
9
10
11     public static void metodo1() throws Exception{
12         try{
13             metodo2();
14         }catch(Exception e){
15             throw new Exception("Exceção enviada no metodo1", e);
16         }
17     }
18
19     public static void metodo2() throws Exception{
20         throw new Exception("Exceção enviada no metodo2");
21     }
22
23 }
24

```

The screenshot shows an IDE interface with two windows. The top window is titled 'teste.java' and contains the provided Java code. The bottom window is titled 'Console' and displays the output of running the program. The console output shows a stack trace starting from 'metodo1' at line 15, which then calls 'main' at line 5. It then shows a 'Caused by:' entry for 'metodo2' at line 20, which calls 'metodo1' at line 13. The error message 'Exceção enviada no metodo1' is shown in red, indicating it is the current exception being propagated.

```

Console × Problems @ Javadoc Declaration Debug
<terminated> teste (3) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe
java.lang.Exception: Exceção enviada no metodo1
    at teste.metodo1(teste.java:15)
    at teste.main(teste.java:5)
Caused by: java.lang.Exception: Exceção enviada no metodo2
    at teste.metodo2(teste.java:20)
    at teste.metodo1(teste.java:13)
    ... 1 more

```

Algumas vezes, a classe que lança um objeto **Throwable** está inserida em uma abstração de camadas baixas e uma operação em um nível mais superior falha por causa de um erro na camada inferior. Usando uma exceção que contém um **cause**, você possibilita à camada superior enviar detalhes da falha ao seu chamador, evitando a propagação de um erro da camada inferior e mantendo flexibilidade para alterar a implementação da camada superior sem alterar sua API e, particularmente, as exceções lançadas pelos seus métodos.

11.7.2. Error

Muitos erros que ocorrem durante a execução de um programa são situações bastante incomuns e, por isso, são considerados graves. Para indicar que esses erros ocorreram e que não devem ser pegos por uma aplicação adequada, existe uma subclasse de **Throwable**: a classe **Error**.

O erro **ThreadDeath** é considerado como uma situação normal. Contudo, o fato de ele não poder ser pego por uma aplicação adequada o faz ser uma subclasse de **Error**.

Veja a sintaxe da classe **Error**:

```
public class Error  
extends Throwable
```

As subclasses de **Error** podem ser lançadas durante a execução de um método, mas não podem ser pegas e geralmente representam problemas de ambiente como falta de memória, versão errada do JVM ou algum outro problema fora da alçada da aplicação que está sendo executada.

Exceções do tipo **Error** não precisam ser declaradas na cláusula de lançamento do método, caso os erros não correspondam a situações incomuns que não deveriam ocorrer.

11.7.3. Exception

A classe **Exception** e suas subclasses têm como função indicar determinadas condições que uma aplicação pode capturar em um bloco **catch**. Funcionam como uma extensão da classe **Throwable**.

A sintaxe de **Exception** é a seguinte:

```
public class Exception  
extends Throwable
```

11.7.4. NullPointerException

Trata-se de uma exceção não verificada pertencente ao pacote `java.lang`.

Exceções do tipo **NullPointerException** ocorrem ao tentar executar um método ou manipular um atributo sobre uma variável que não possui um objeto referenciado:

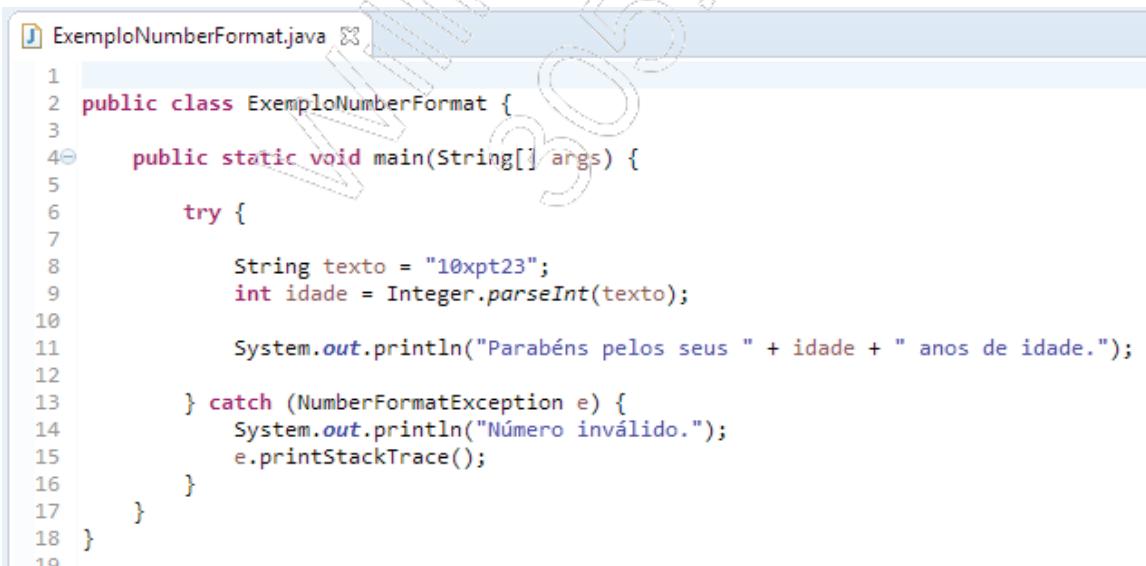


```
ExemploNullPointer.java
1
2 public class ExemploNullPointer {
3
4     public static void main(String[] args) {
5
6         try {
7
8             Cliente c = null;
9             c.setNome("Manuel da Silva");
10
11        } catch (NullPointerException e) {
12            System.out.println("Variável não instanciada.");
13            e.printStackTrace();
14        }
15    }
16
17 }
```

11.7.5. NumberFormatException

É uma exceção não verificada também pertencente ao pacote `java.lang`.

Exceções do tipo **NumberFormatException** ocorrem ao tentar realizar a conversão de uma string para um valor numérico:

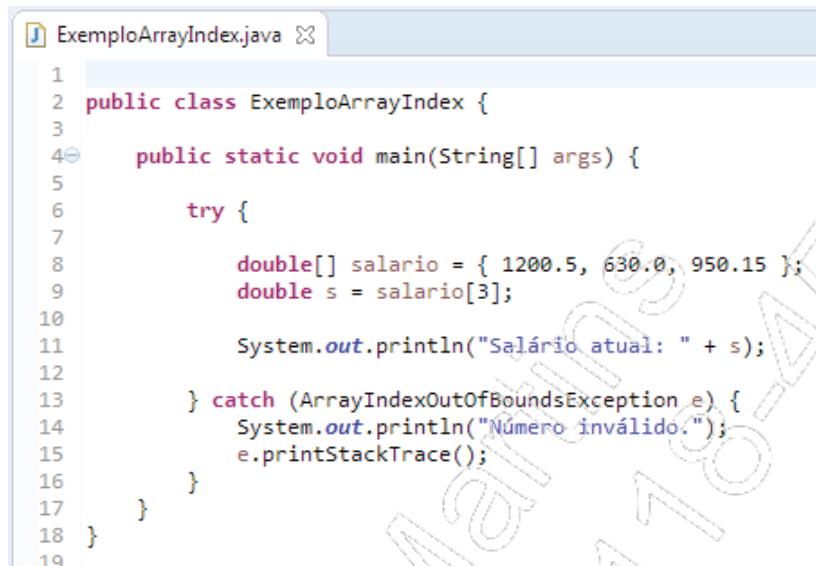


```
ExemploNumberFormatException.java
1
2 public class ExemploNumberFormatException {
3
4     public static void main(String[] args) {
5
6         try {
7
8             String texto = "10xpt23";
9             int idade = Integer.parseInt(texto);
10
11            System.out.println("Parabéns pelos seus " + idade + " anos de idade.");
12
13        } catch (NumberFormatException e) {
14            System.out.println("Número inválido.");
15            e.printStackTrace();
16        }
17    }
18 }
```

11.7.6. ArrayIndexOutOfBoundsException

Essa é outra exceção não verificada, também pertencente ao pacote **java.lang**.

Exceções do tipo **ArrayIndexOutOfBoundsException** ocorrem ao tentar obter um item inexistente em um array:

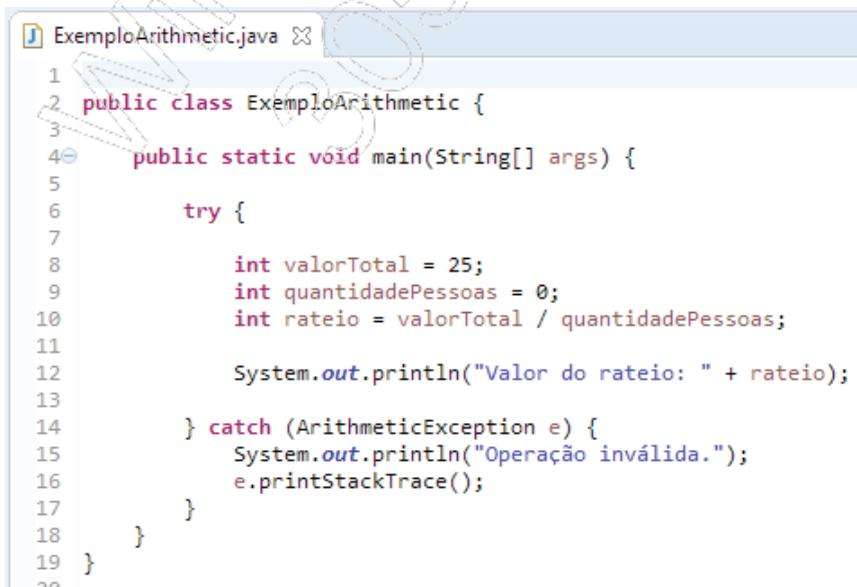


```
1  public class ExemploArrayIndex {
2
3     public static void main(String[] args) {
4
5         try {
6
7             double[] salario = { 1200.5, 630.0, 950.15 };
8             double s = salario[3];
9
10            System.out.println("Salário atual: " + s);
11
12        } catch (ArrayIndexOutOfBoundsException e) {
13            System.out.println("Número inválido.");
14            e.printStackTrace();
15        }
16    }
17
18 }
```

11.7.7. ArithmeticException

Trata-se de uma exceção não verificada, também pertencente ao pacote **java.lang**.

Exceções do tipo **ArithmeticException** ocorrem em operações aritméticas com números inteiros que possuem resultado inválido:



```
1  public class ExemploArithmetic {
2
3     public static void main(String[] args) {
4
5         try {
6
7             int valorTotal = 25;
8             int quantidadePessoas = 0;
9             int rateio = valorTotal / quantidadePessoas;
10
11            System.out.println("Valor do rateio: " + rateio);
12
13        } catch (ArithmaticException e) {
14            System.out.println("Operação inválida.");
15            e.printStackTrace();
16        }
17    }
18
19 }
```

11.7.8. ClassCastException

Outra exceção não verificada pertencente ao pacote `java.lang`.

Exceções do tipo **ClassCastException** ocorrem ao tentar realizar o cast (tipagem) para um tipo incompatível com o objeto:

```
1 public class Executando {
2     public static void main(String[] args) {
3
4         try {
5
6             Conta c = new ContaPoupanca();
7             ContaCorrente cc = (ContaCorrente)c;
8             cc.debitarTarifa(35.5);
9
10        } catch (ClassCastException e) {
11            System.out.println("Conta de tipo incompatível.");
12            e.printStackTrace();
13        }
14    }
15 }
```

11.7.9. IOException

Já esta é uma exceção verificada pertencente ao pacote `java.io`.

Este tipo de exceção ocorre em operações de input/output, como leitura de arquivos ou comunicação por rede que não são realizadas com sucesso.

Possui diversas classes filhas que representam subtipos de problemas do gênero I/O:

```
ExemploIO.java
1 import java.io.FileReader;
2 import java.io.IOException;
3
4 public class ExemploIO {
5
6     public static void main(String[] args) {
7
8         try {
9
10             FileReader doc;
11             doc = new FileReader("C:\\\\carta.txt");
12
13         } catch (IOException e) {
14             System.out.println("Não foi possível abrir o arquivo.");
15             e.printStackTrace();
16         }
17     }
18 }
```

11.7.10. Classe SQLException

Exceção verificada pertencente ao pacote `java.sql`.

Ocorre ao tentar realizar operações de acesso a bancos de dados que resultam em problemas, tais como:

- Tentar acessar um banco de dados que não está no ar;
- Tentar conectar com usuário ou senha inválidos;
- Tentar acessar uma tabela inexistente.

Veja um exemplo:



```
ExemploSQL.java
1 import java.sql.Connection;
2 import java.sql.DriverManager;
3 import java.sql.SQLException;
4
5 public class ExemploSQL {
6
7     public static void main(String[] args) {
8
9         try {
10
11             Connection cn;
12             cn = DriverManager.getConnection(
13                 "jdbc:oracle:thin:localhost:1521:orcl", "scott", "tiger");
14
15         } catch (SQLException e) {
16             System.out.println("Não foi possível realizar a conexão.");
17             e.printStackTrace();
18         }
19     }
20 }
21
```

11.8. Exceções personalizadas

Quando precisa lançar uma exceção, você tem duas opções: uma é usar uma exceção já criada, seja nativa da própria linguagem Java ou criada por terceiros; a outra é criar você mesmo sua exceção. Isso pode ser útil em muitos casos. Para saber se criar sua própria exceção é adequado em determinada situação, baseie-se nas perguntas a seguir:

- O tipo de exceção que você precisa realmente não pode ser encontrado em Java?
- Suas exceções serão úteis para outros usuários?
- O código em questão lança mais que um tipo de exceção relacionada?
- Outros usuários terão acesso às exceções criadas? Ou, em outras palavras, o pacote que conterá suas exceções será independente?

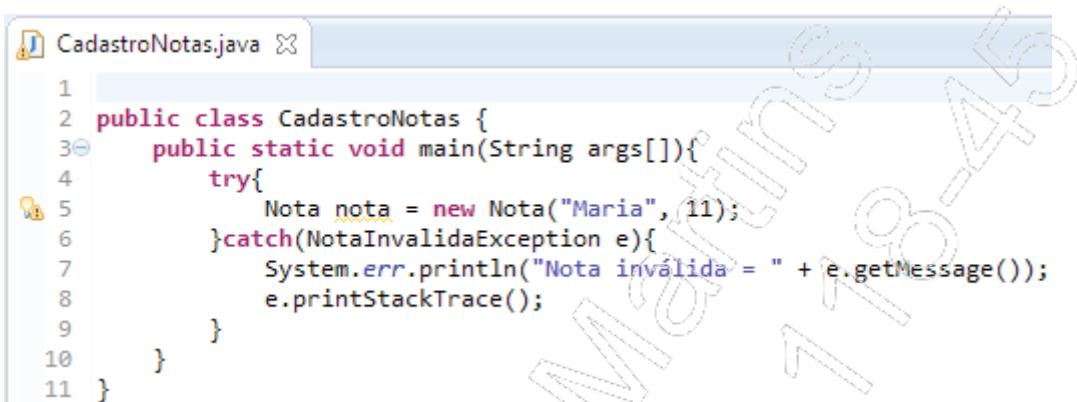
Ao criar uma exceção própria, você estende a classe **Exception** para criar uma nova classe dela. Veja no exemplo a seguir como fazer isso:

```
public class NotaInvalidaException extends Exception {  
}
```

Veja, agora, um exemplo utilizando a classe que acabamos de criar para tratar exceções. Neste exemplo, a classe **Nota** recebe uma determinada nota e verifica se ela é válida, permitindo que apenas os valores maiores ou iguais a zero e menores ou iguais a 10 sejam aceitos:

```
1  public class Nota {
2
3      private String aluno;
4      private int nota;
5
6
7      public Nota(String aluno, int nota) throws NotaInvalidaException{
8          this.setAluno(aluno);
9          this.setNota(nota);
10     }
11
12     public void setAluno(String aluno){
13         this.aluno = aluno;
14     }
15
16     public String getAluno(){
17         return this.aluno;
18     }
19
20     public void setNota(int nota) throws NotaInvalidaException{
21         if(nota < 0 || nota > 10){
22             NotaInvalidaException e = new NotaInvalidaException();
23             throw e;
24         }else{
25             this.nota = nota;
26         }
27     }
28
29     public int getNota(){
30         return this.nota;
31     }
32 }
```

No método **setNota**, observando a linha **if(nota < 0 || nota > 10)**, se esta condição for verdadeira, é gerada uma nova exceção (**NotaInvalidaException e = new NotaInvalidaException()**). Na classe **CadastroNotas**, a seguir, trataremos esta exceção:



```
1  public class CadastroNotas {  
2      public static void main(String args[]){  
3          try{  
4              Nota nota = new Nota("Maria", 11);  
5          }catch(NotaInvalidaException e){  
6              System.err.println("Nota inválida = " + e.getMessage());  
7              e.printStackTrace();  
8          }  
9      }  
10     }  
11 }
```

Após compilar o código, ao tentar executá-lo, o resultado será como o exibido na figura a seguir:



```
Nota inválida = null  
NotaInvalidaException  
    at Nota.setNota(Nota.java:22)  
    at Nota.<init>(Nota.java:9)  
    at CadastroNotas.main(CadastroNotas.java:5)
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- A primeira etapa para criar um manipulador de exceções é encerrar o código que pode gerar exceções dentro de um bloco **try**;
- Para indicar que um método pode gerar uma exceção, utilizamos o comando **throws** na sua declaração;
- Um bloco **finally** é sempre executado após a finalização de um bloco **try**, mesmo quando ocorre uma exceção inesperada. Assim, deve ser usado para conter código que deva ser sempre executado;
- Para chegar ao método atual, o programa deve executar uma cadeia de métodos, que é conhecida como pilha de chamadas. O último método acessado é colocado no topo da pilha e, consequentemente, os métodos acessados anteriormente ficam posicionados antes, no final da pilha. Essa ordem também pode ser invertida;
- Os argumentos, o retorno e as exceções que um método pode lançar devem ser sempre declarados. Todas as exceções que um método pode lançar, com exceção daquelas que fazem parte das subclasses de **RuntimeException** e **Error**, estão em uma lista na interface pública do método;
- A classe **Throwable** é uma classe específica, em Java, para erros e exceções. Nesta classe, encontramos as subclasses **Error** e **Exception**, que são utilizadas de maneira convencional para indicar situações incomuns que tenham ocorrido;
- A classe **Exception** e suas subclasses têm como função indicar determinadas condições que uma aplicação pode capturar em um bloco **catch**. Funcionam como uma extensão da classe **Throwable**;
- A classe **Error** (uma subclass de **Throwable**) indica erros que representam situações bastante incomuns e graves e que, por isso, não devem ser pegos por uma aplicação;
- Quando for gerado um erro de acesso ao banco de dados, podemos obter informações a partir da classe **SQLException**;
- Quando precisa lançar uma exceção, você tem duas opções: uma é usar uma exceção já criada, seja nativa da própria linguagem Java ou criada por terceiros; a outra é criar você mesmo sua exceção. Isso pode ser útil em muitos casos. Uma exceção personalizada sempre é uma extensão da classe **Exception**.



11

Tratamento de exceções

Teste seus conhecimentos



1. Qual cláusula utilizamos para indicar que um método pode gerar uma exceção?

- a) try
- b) catch
- c) throws
- d) finally
- e) Nenhuma das alternativas anteriores está correta.

2. Qual das alternativas a seguir está correta sobre o bloco finally?

- a) Sempre é executado após a finalização de um bloco try.
- b) É impossível colocar um código de depuração ou limpeza dentro dele.
- c) Seu uso é obrigatório.
- d) Se um bloco try aparecer logo depois de um bloco finally, será disparada uma mensagem de erro.
- e) Nenhuma das alternativas anteriores está correta.

3. Qual comando utilizamos para efetivar o lançamento de um objeto de exceção na pilha de execução de uma aplicação Java?

- a) try
- b) catch
- c) throws
- d) catches
- e) throw

4. Qual dessas exceções é uma exceção verificada?

- a) FileNotFoundException (herdeira de IOException)
- b) NullPointerException
- c) AritmeticException
- d) NumberFormatException
- e) RuntimeException



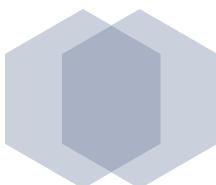
11

Tratamento de exceções



Mãos à obra!

Willians Martins
305.000.1845



Editora
IMPACTA



Laboratório 1

A – Verificando valores numéricos

1. Crie uma classe executável chamada **Exercicioldade**;
2. Através da classe **Scanner**, método **nextLine()**, solicite ao usuário que digite o ano de seu nascimento, atribuindo o valor digitado a uma variável de tipo **string**;
3. Utilize o método **Integer.parseInt()** para converter o valor inserido anteriormente para numérico e atribua este valor convertido a uma variável **int**;
4. Calcule e exiba a idade do usuário, assumindo o ano corrente;
5. Coloque todas as instruções anteriores dentro de um bloco **try** e, abaixo deste, crie um bloco **catch** realizando o tratamento sobre a exceção **NumberFormatException**. Este bloco **catch** deverá conter uma instrução **System.out.println()**, que exibirá a mensagem “Valor digitado inválido”. Esta mensagem será exibida quando o valor digitado pelo usuário não puder ser convertido para número;
6. Compile e execute o programa.

B – Tratando uma exceção verificada

1. Crie uma classe executável chamada **ExercicioGravacao**;
2. Dentro do método **main**, solicite que o usuário digite uma frase qualquer, utilizando o método **nextLine()** da classe **Scanner**;
3. Utilize o código a seguir para gravar a mensagem digitada em um arquivo texto, em que a variável **texto** é aquela que foi utilizada no passo anterior para obter a mensagem digitada pelo usuário:

```
PrintWriter writer = new PrintWriter("C:\\\\doc1.txt");
writer.println(texto);
writer.close();
```

4. Coloque todas as instruções anteriores dentro de um bloco **try** e, abaixo deste, crie um bloco **catch** realizando o tratamento sobre a exceção **IOException**. Este bloco **catch** deverá conter uma instrução **System.out.println()**, que exibirá a mensagem “Falha ao gravar as informações digitadas”. Esta mensagem será exibida quando o arquivo mencionado não puder ser utilizado para gravação;
5. Compile e execute o exercício.

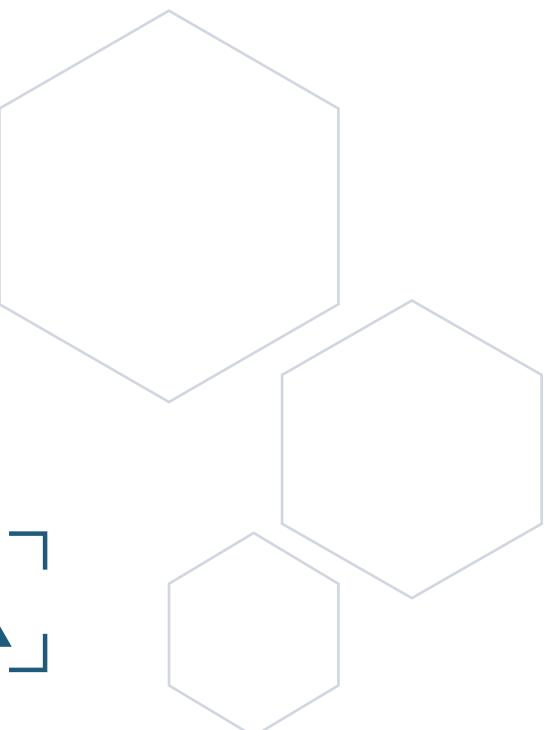
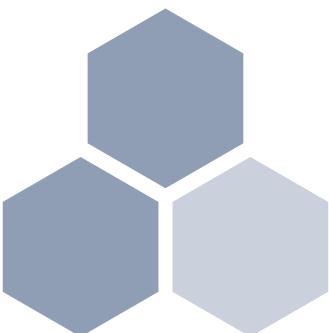
Observe que, neste exercício, o bloco **try/catch** é obrigatório para a aplicação em tempo de compilação, diferente das exceções do exercício anterior.



12

As bibliotecas Java e o Javadoc

- ◆ Conceito de API;
- ◆ Javadoc e a documentação oficial Java;
- ◆ Criação de uma documentação Javadoc.



12.1. Conceito de API

Uma API é uma biblioteca de classes, pacotes e interfaces disponibilizadas para uso dos desenvolvedores em suas aplicações. Estas APIs, de um modo geral, podem ser de diferentes tipos:

- Oficiais do Java (presentes no JDK);
- Opcionais do Java (download sob demanda);
- Não oficiais (provenientes de terceiros).

As APIs ajudam programadores na consulta a funções diversas dos pacotes, parâmetros, métodos e classes disponíveis.

12.2. Javadoc e a documentação oficial Java

Javadoc é a forma como a documentação oficial das APIs do Java está organizada. A documentação está em HTML e pode ser disponibilizada localmente na máquina do desenvolvedor, por meio de um download na página oficial ou consultada via Internet.

A página da documentação on-line da linguagem segue o padrão: [https://docs.oracle.com/javase/\[X\]/docs/api/index.html](https://docs.oracle.com/javase/[X]/docs/api/index.html), em que [X] representa a versão desejada para consulta. Por exemplo, para consultar a documentação do Java 9, basta acessar a URL adiante: <https://docs.oracle.com/javase/9/docs/api/index.html>.

A documentação apresenta todos os pacotes e classes oficiais da linguagem, como as APIs gráficas, de entrada/saída, coleções, utilitários etc. Ela está disposta em três frames, a partir da página principal da documentação, conforme pode ser observado na figura mais adiante.

O primeiro frame exibe todos os pacotes. Quando um pacote é selecionado, o segundo frame exibe todas as interfaces, classes e exceções desse pacote. O terceiro e principal frame exibe a documentação do elemento escolhido no segundo frame.



Uma novidade que surgiu a partir do Java 9 foi a possibilidade de realizar uma busca pela documentação através do campo SEARCH, o que não existia em versões anteriores.

12.3. Criação de uma documentação Javadoc

O desenvolvedor provê uma documentação Javadoc para a aplicação de uma forma muito simples, bastando que coloque, em pontos chave do código fonte, comentários Javadoc sempre imediatamente antes de declarar classes, interfaces, métodos, construtores ou atributos.

Conforme visto anteriormente, comentários Javadoc consistem em colocar um texto entre os símbolos `/**` (inicial) e `*/` (final), podendo formar uma ou mais linhas (comentário de bloco), conforme exemplo a seguir:

```
/*
 * Comentário Javadoc de uma classe
 * Aqui se encontra como a classe funciona
 */
```

É importante enfatizar que qualquer outro lugar além dos descritos no exemplo anterior (onde se coloca o comentário Javadoc) será ignorado pelo gerador.

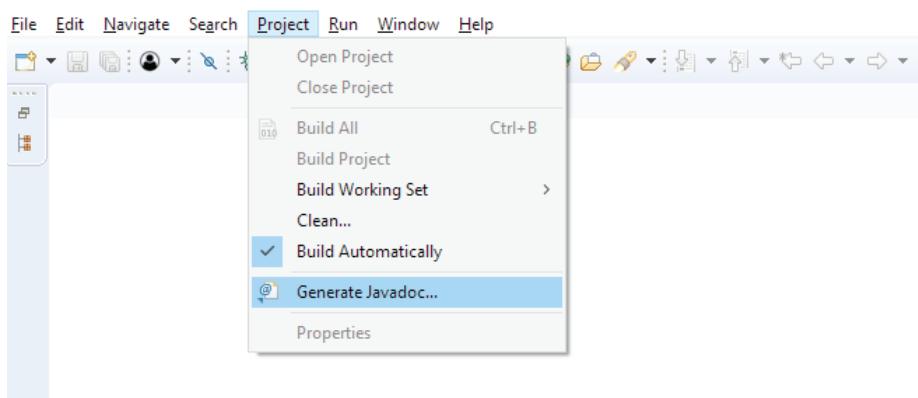
O gerador gera toda documentação a partir dos comentários no formato HTML, portanto, no texto de comentário pode-se utilizar as tags do HTML.

Além disso, tags podem ser utilizadas para documentar alguns aspectos e elementos para que tenham posicionamento e formatação especiais, como:

- `@param`: Documenta parâmetros de um método;
- `@returns`: Documenta o tipo de retorno de um método;
- `@see`: Documenta uma referência a outra classe da documentação;
- `@throws`: Documenta as exceções lançadas por um método ou construtor.

12.3.1. Geração da página de documentação

Após incluir os comentários Javadoc em todas as classes relevantes da aplicação Java, basta executar o programa de geração da documentação a partir da IDE, acessando o menu **Project / Generate Javadoc...**:



Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

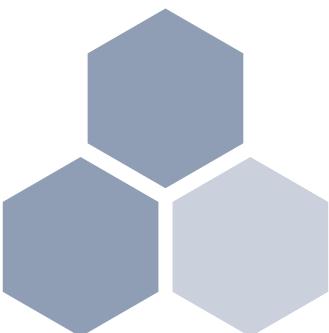
- A documentação oficial das APIs do Java estão dispostas no formato HTML e podem ser acessadas on-line ou instaladas localmente na máquina do desenvolvedor;
- A documentação Java é uma página HTML organizada e exibida em um estilo particular conhecido como **Javadoc**;
- O desenvolvedor Java pode documentar seus projetos no formato Javadoc a partir da ferramenta disponível no JDK (aplicativo Javadoc), que é o gerador da documentação a partir dos comentários de bloco do programa (iniciados por `/**` e finalizados por `*/`).



12

As bibliotecas Java e o Javadoc

Teste seus conhecimentos



1. A respeito do comentário Javadoc, indique a opção verdadeira:

- a) Comentários Javadoc só podem ser criados a partir de comentário de linha.
- b) Colocando um comentário no interior de um método, este é considerado na geração da documentação.
- c) O formato de geração da documentação Javadoc é sempre no formato .TXT.
- d) O comentário Javadoc deve ser colocado imediatamente antes da declaração de classes, métodos, construtores e atributos, para ser considerado pelo gerador de documentação.
- e) Nenhuma das alternativas anteriores está correta.

2. Indique o exemplo correto de comentário válido para a geração de documentação Javadoc:

- a) // conteúdo da documentação
- b) /* conteúdo da documentação */
- c) /** conteúdo da documentação */
- d) /* conteúdo da documentação **/
- e) /// conteúdo da documentação ///

3. Qual o formato de documento gerado pela ferramenta Javadoc?

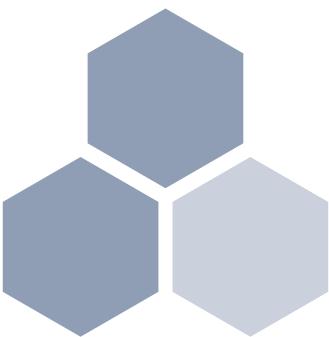
- a) HTML
- b) PDF
- c) .DOC
- d) XML
- e) .TXT



13

Testes unitários com Java

- ◆ Conceito de teste unitário;
- ◆ Como implantar o teste unitário;
- ◆ Utilizando o JUnit.



13.1. Conceito de teste unitário

Teste unitário é a prática de se testar funções ou unidades de um código. A partir dele, é possível verificar se a função funciona adequadamente. Uma função funcionar adequadamente significa que, para diferentes tipos de entrada, ela responde corretamente de acordo com o esperado e, mais do que isso, que ela se comporta bem em situações de entradas inválidas ou falhas diversas.

A finalidade do teste unitário e sua automatização no código desenvolvido é ajudar o desenvolvedor a identificar falhas nos algoritmos desenvolvidos e também melhorar a qualidade do código de uma determinada função. À medida que se inicia a criação de testes para cobrir diversas situações, naturalmente cria-se uma suíte de testes que pode ser executada durante o ciclo de vida de desenvolvimento do software continuamente, verificando constantemente a qualidade do produto desenvolvido.

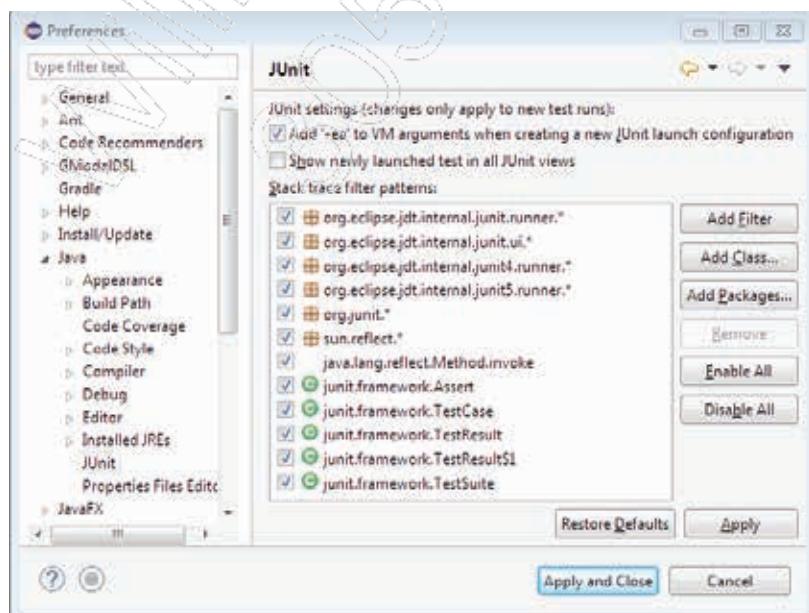
13.2. Como implantar o teste unitário

Em programação orientada a objetos, a menor unidade testável é o método. Desta forma, um determinado método pode ser alvo de diversos testes, mediante chamadas com parâmetros diferentes para cada execução.

Para auxiliar o desenvolvimento de testes unitários, utilizamos frameworks de testes. Dentre estes frameworks, destaca-se o JUnit, que está integrado na maioria das IDEs existentes no mercado.

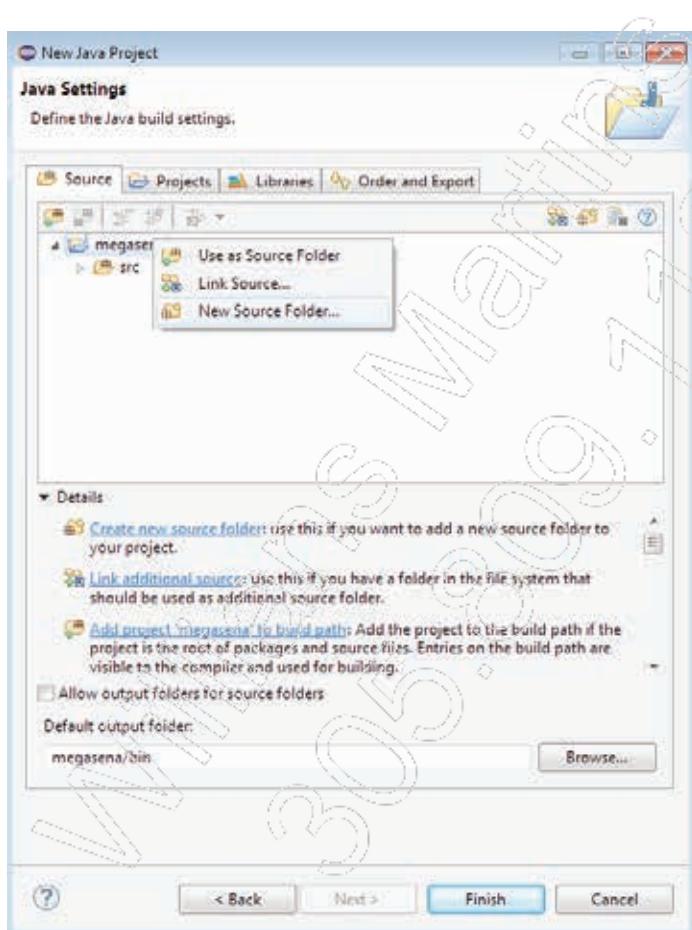
13.3. Utilizando o JUnit

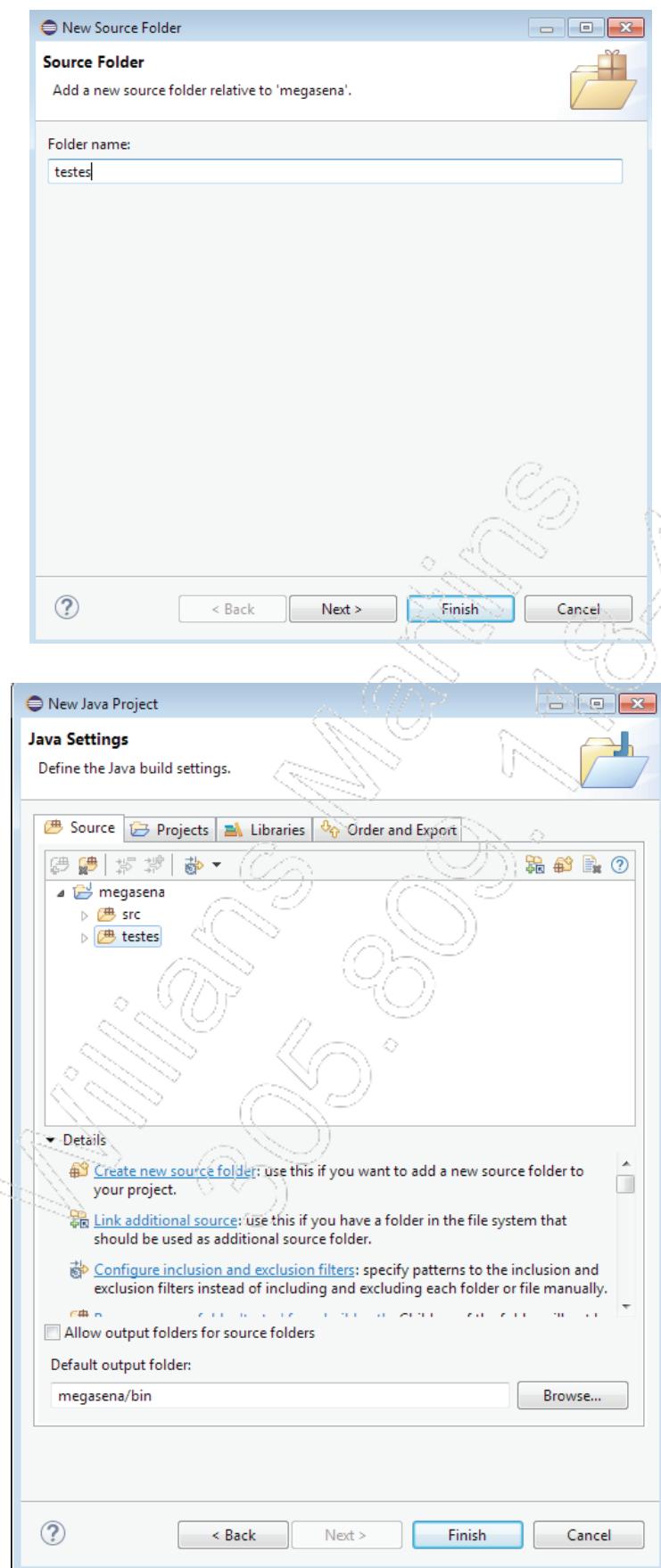
A partir da versão Oxygen do Eclipse, o JUnit já é suportado nativamente, como pode ser verificado na configuração da IDE:



13.3.1. Criando um teste unitário

Vamos considerar um cenário em que haja um projeto Java contendo uma classe utilitária relacionada ao jogo de loteria Megasena. Dentro do projeto em questão, é uma boa prática separar as classes que executam os testes unitários das classes do projeto. No Eclipse, isso é feito na configuração do projeto:

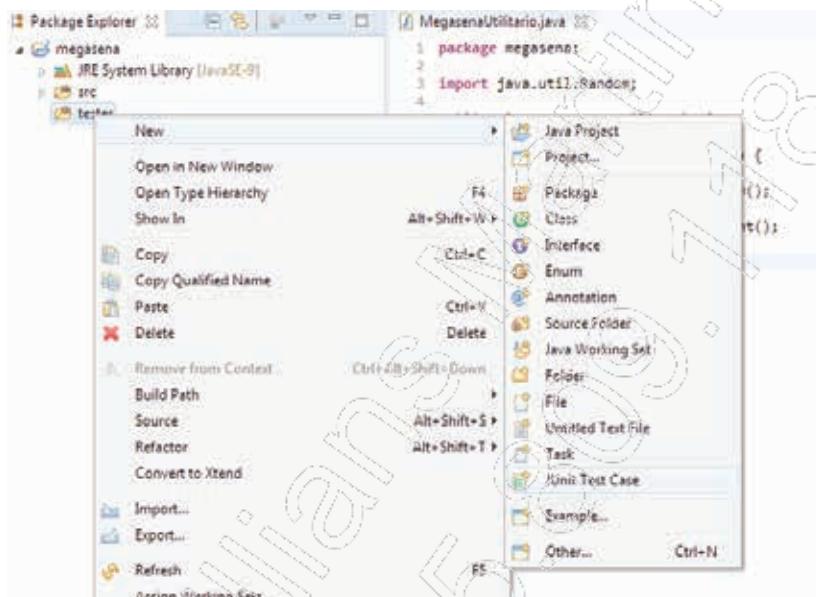


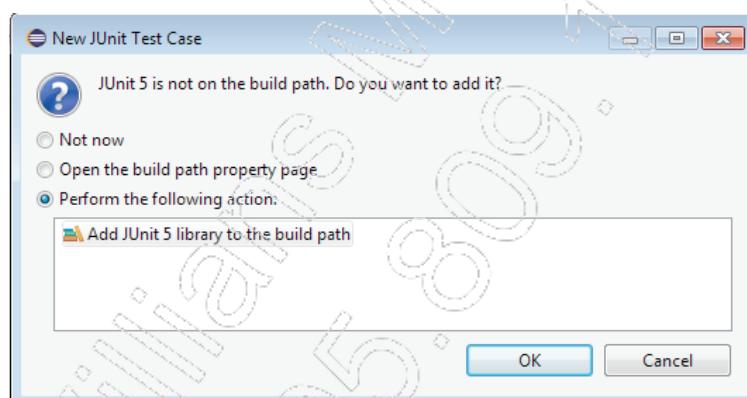
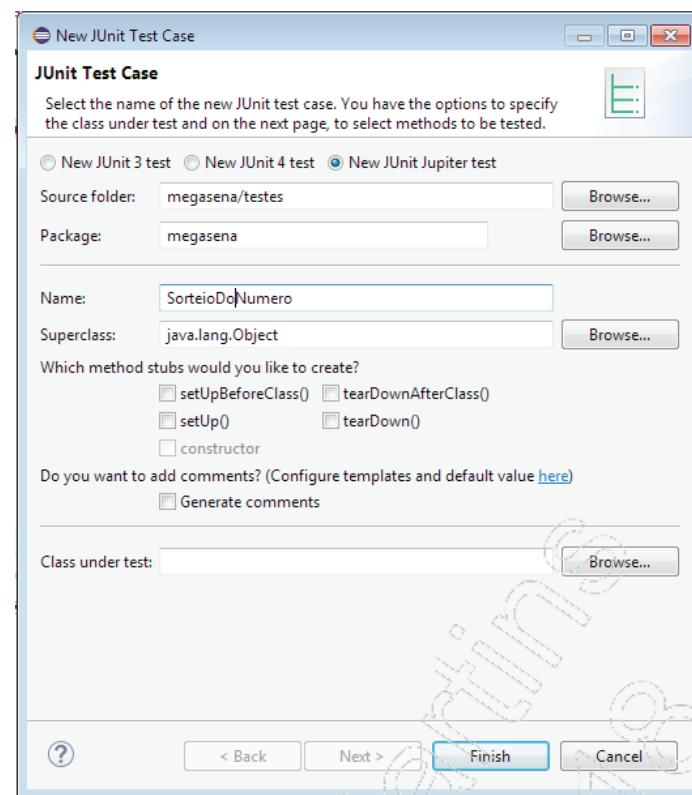


Considere a classe utilitária adiante contendo apenas um método:

```
public class MegasenaUtilitario {  
  
    public static int sortearNumero() {  
  
        Random sorteador = new Random();  
        int numero = sorteador.nextInt();  
  
        return numero;  
    }  
  
}
```

Na pasta **testes**, inclua um novo caso de teste conforme esquema a seguir:





Ao final da configuração, o seguinte template de código é gerado:

O desenvolvimento de casos de teste utilizando o JUnit é realizado através de **annotations** e **assertions**. Na sequência, serão apresentados os principais, acompanhados de uma explicação sucinta a respeito.

13.3.1.1. Ciclo de vida de um teste

- **@Test:** É a annotation mais básica do JUnit, que serve para marcar métodos que serão executados como testes.

Antes e depois:

- **@BeforeAll:** Executado uma única vez antes da execução de qualquer teste e antes de todos os métodos marcados com a annotation **@BeforeEach**;
- **@BeforeEach:** Executado antes de cada teste;
- **@AfterEach:** Executado após cada teste;
- **@AfterAll:** Executado uma única vez depois da execução de todos os testes e depois de todos os métodos marcados com a annotation **@AfterEach**.

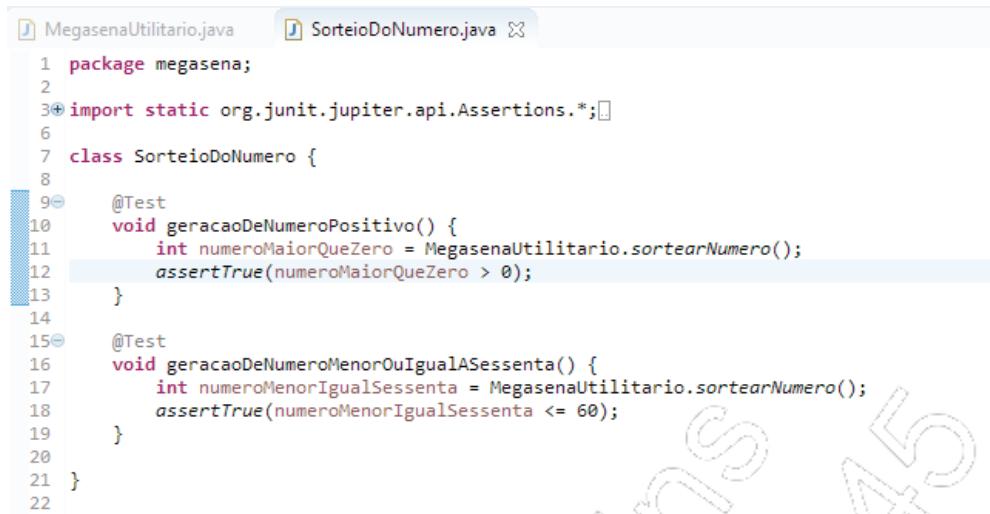
13.3.1.2. Assertions

Depois que a instância foi preparada e a funcionalidade executada, **assertions** garantem que as condições desejadas foram atingidas, caso contrário, o teste falha.

Os **assertions** básicos de testes unitários verificam basicamente duas coisas: valores esperados de atributos de uma instância (ex.: se um valor é nulo) ou comparações diversas (ex.: se duas instâncias são iguais).

- **assertEquals:** Verifica se os parâmetros informados são iguais;
- **assertNotEquals:** Verifica se os parâmetros informados são diferentes;
- **assertFalse:** Verifica se determinada condição informada no parâmetro é falsa;
- **assertTrue:** Verifica se determinada condição informada no parâmetro é verdadeira;
- **assertNotNull:** Verifica se o parâmetro não é nulo;
- **assertNull:** Verifica se o parâmetro é nulo;
- **fail:** Força a falha de um teste informando uma mensagem especificada no parâmetro.

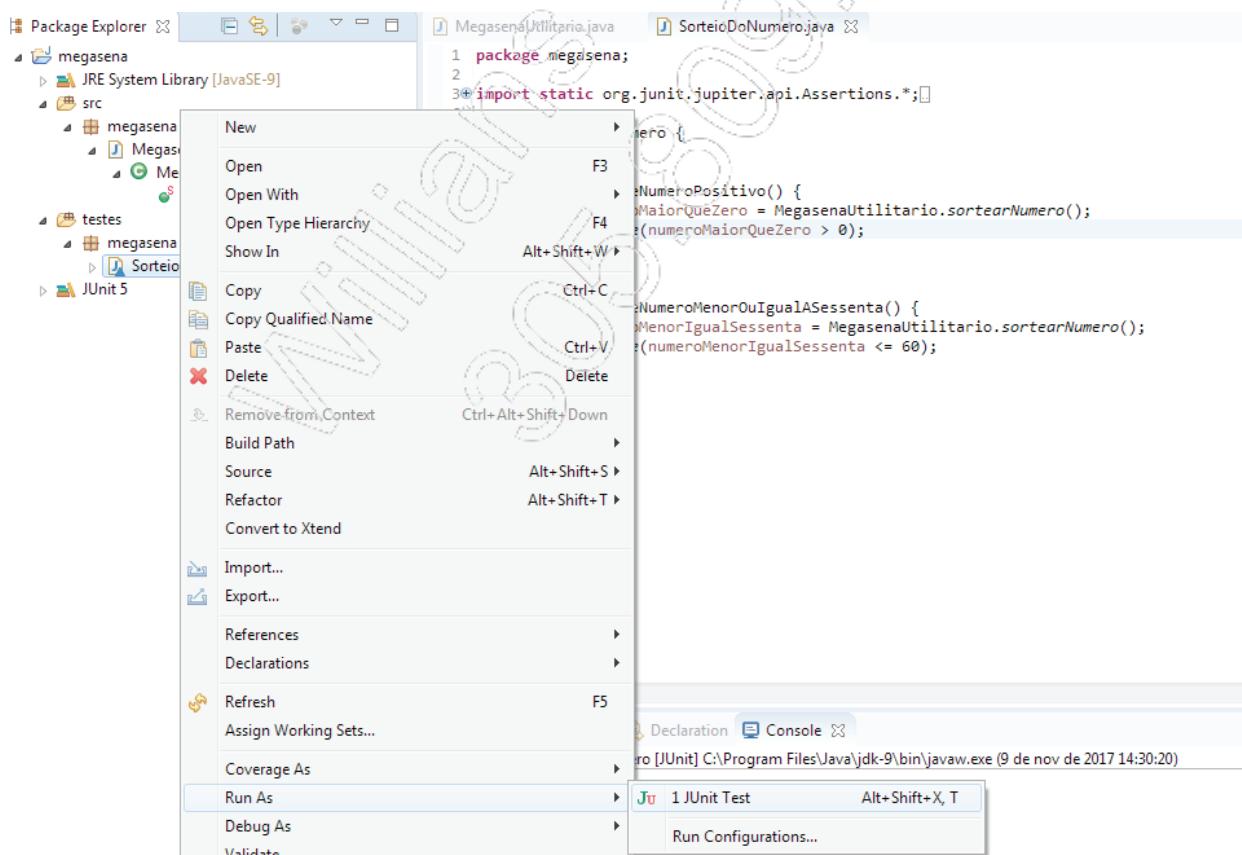
Agora que foram apresentados os elementos essenciais do JUnit, verifique como ficaram os testes da unidade (método **sortearNumero**):



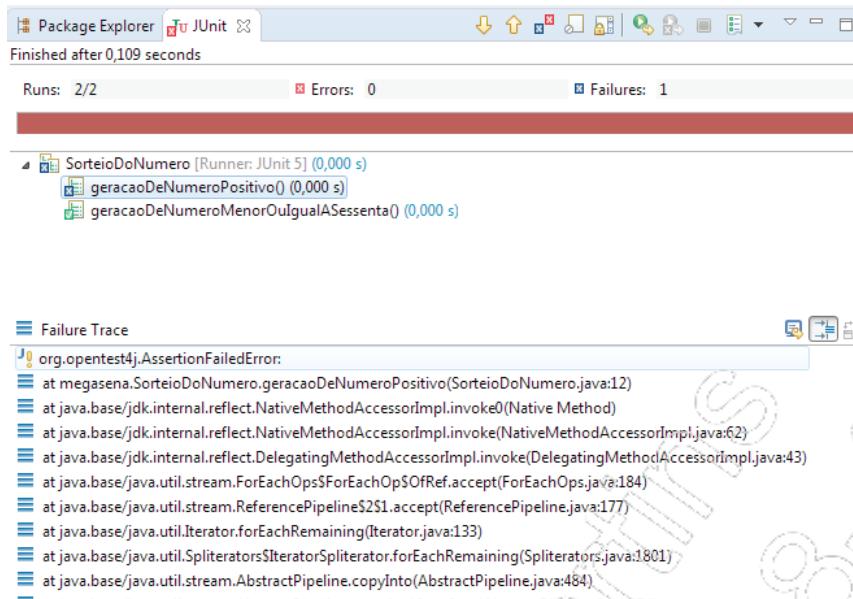
```
1 package megasena;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 class SorteoDoNumero {
6
7     @Test
8     void geracaoDeNumeroPositivo() {
9         int numeroMaiorQueZero = MegasenaUtilitario.sortearNumero();
10        assertTrue(numeroMaiorQueZero > 0);
11    }
12
13    @Test
14    void geracaoDeNumeroMenorOuIgualASessenta() {
15        int numeroMenorIgualSessenta = MegasenaUtilitario.sortearNumero();
16        assertTrue(numeroMenorIgualSessenta <= 60);
17    }
18
19 }
20
21 }
22 }
```

O teste verifica, basicamente, se o número gerado pelo utilitário atende às regras de números válidos para a Megasena: O número gerado deve estar entre 1 (inclusive) e 60 (inclusive).

Uma vez definidos os testes necessários, basta executar os testes de forma automática conforme procedimento adiante:



Após a execução de cada caso de teste definido, o framework apresenta um relatório de execução. A partir do resultado, verifica-se a necessidade de corrigir o algoritmo testado ou aferir se a unidade testada está correta.



No relatório anterior, verifica-se que há uma falha no código de geração do número, pois o caso de teste que verifica se o número gerado é positivo falhou. Na realidade, executando várias vezes o código de teste, obtém-se resultados aleatórios: ora os dois testes são bem sucedidos, ora os dois falham e ora um é bem sucedido e o outro não.

Verificando o código da classe que gera o número, percebemos que não há qualquer controle quanto ao número gerado, podendo ser gerado um número negativo em algumas execuções (já que gera um número inteiro aleatório) e maior que 60 em outras. Assim, é necessário corrigir o algoritmo e reexecutar os testes para garantir que o problema tenha sido sanado. Adiante, segue a classe com as correções realizadas:

```

public class MegasenaUtilitario {

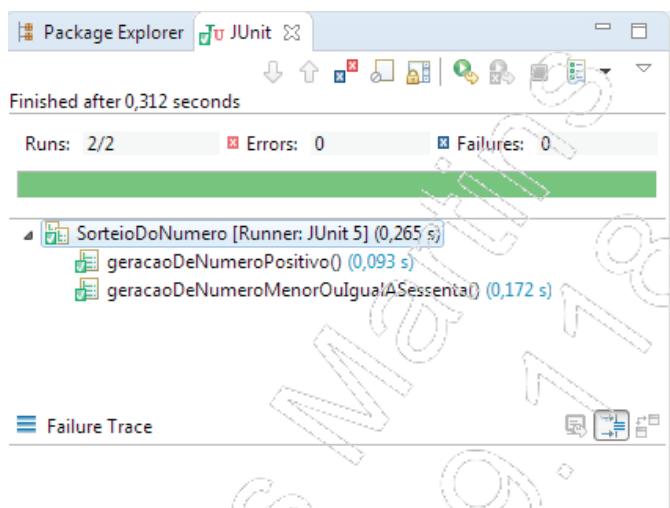
    public static int sortearNumero() {
        Random sorteador = new Random();
        int numero = sorteador.nextInt();

        while(numero <= 0 || numero > 60) {
            numero = sorteador.nextInt();
        }

        return numero;
    }
}

```

Com a correção realizada, as condições de geração são verificadas dentro do **while**. Assim, enquanto um número válido não for gerado, ele não é retornado pelo método. Com esse ajuste, executando os casos de teste diversas vezes, sempre se obtém testes executados com sucesso.



13.4. Conclusão

Teste unitário é um poderoso recurso disponível ao desenvolvedor para testar código, proporcionando meios para a melhoria da qualidade e da confiabilidade do software produzido.

Neste capítulo, foi apresentada uma visão geral do JUnit, como configurá-lo na IDE e como implementar um teste unitário simples. Para aprofundar-se em outros conceitos relacionados, aplique testes unitários em contextos e ambientes mais complexos e consulte a documentação oficial da ferramenta (versão 5 ou superior): <http://junit.org/junit5/docs/current/user-guide/>.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

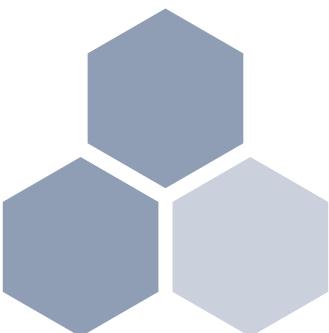
- Teste unitário é a prática de se testar funções ou unidades de um código. A partir dele, é possível verificar se a função funciona adequadamente;
- A finalidade do teste unitário e sua automatização no código desenvolvido é ajudar o desenvolvedor a identificar falhas nos algoritmos desenvolvidos e também melhorar a qualidade do código de uma determinada função;
- Para auxiliar o desenvolvimento de testes unitários, utilizamos frameworks de testes. Dentre estes frameworks, destaca-se o JUnit, que está integrado na maioria das IDEs existentes no mercado.



13

Testes unitários com Java

Teste seus conhecimentos



Editora
IMPACTA

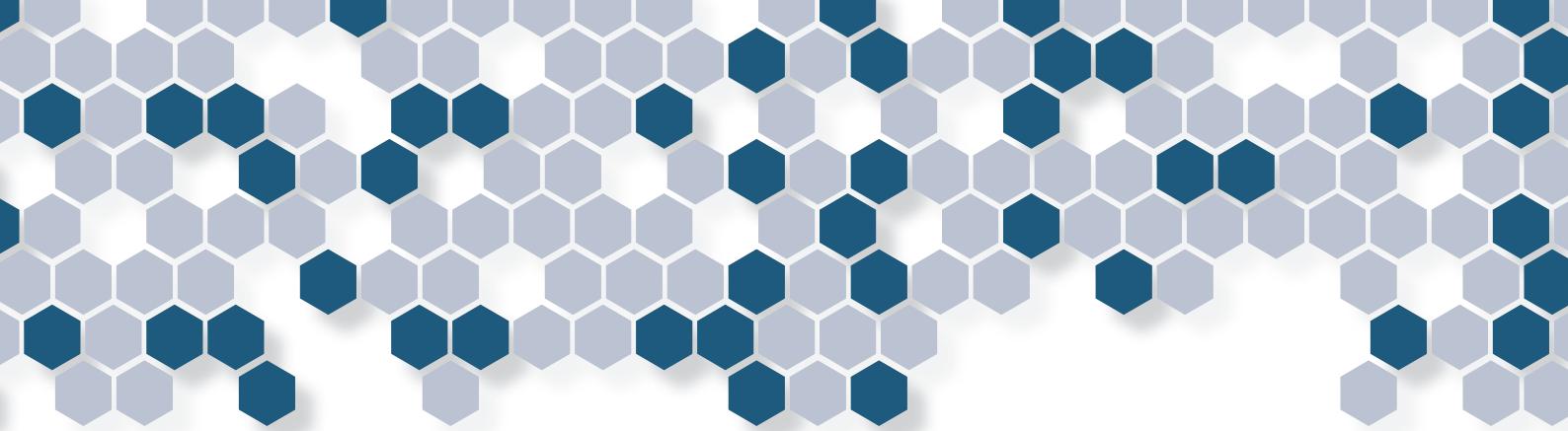


1. Conceitualmente, testes unitários visam testar:

- a) A estrutura de uma classe.
- b) Se o sistema funciona corretamente.
- c) Aspectos de usabilidade da aplicação.
- d) O funcionamento adequado de uma classe da aplicação.
- e) Nenhuma das alternativas anteriores está correta.

2. O que é um assertion no JUnit?

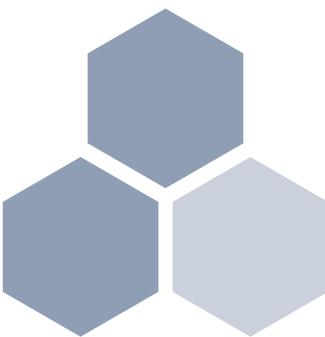
- a) É um atributo da classe Teste.
- b) É um parâmetro presente em todos os métodos de teste.
- c) É uma afirmação correta a respeito do código que deve ser atendida nos testes unitários.
- d) É um método que executa apenas no momento do carregamento do JUnit.
- e) Nenhuma das alternativas anteriores está correta.



14

Programação funcional

- Interface funcional;
 - Expressões lambda;
 - Referenciando métodos;
 - O pacote `java.util.function`.
- 



14.1. Introdução

Neste capítulo, exploraremos um novo paradigma de programação já utilizado por algumas das linguagens da atualidade e que, a partir da versão 8, também pode ser utilizado pelo Java. Trata-se da programação funcional.

O princípio básico da programação funcional é poder realizar chamadas de métodos passando funções como argumentos. Tais argumentos são pequenos objetos sem estado (sem atributos) responsáveis por uma única operação (método), que chamamos de função.

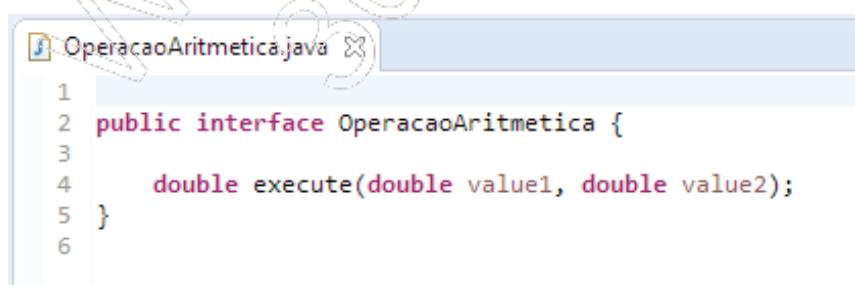
14.1.1. Vantagens da programação funcional

A seguir, veja quais são as principais vantagens da programação funcional:

- **Alto nível de abstração:** A programação funcional suprime muitos detalhes da programação e minimiza a probabilidade de erros;
- **Reusabilidade:** Como consequência da independência de estado, a utilização de objetos sem atributos, que simplesmente realizam operações, permite aos programas o reuso de procedimentos sem efeitos colaterais em diferentes partes do código;
- **Simplicidade:** A ausência de estado (atributos na classe operadora) torna a análise e manutenção de sua aplicação muito mais simples.

14.1.2. Um primeiro exemplo

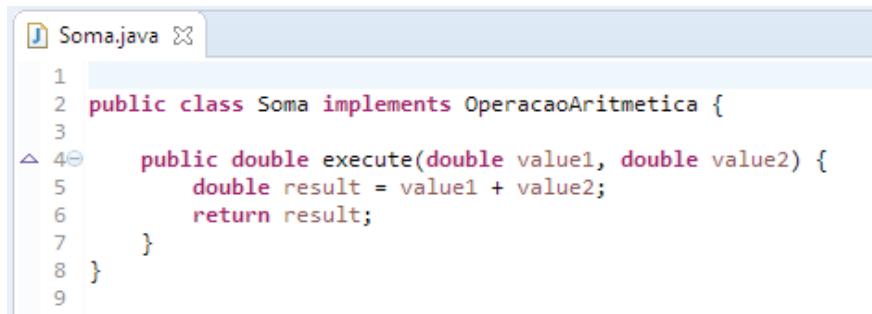
Conforme visto em capítulos anteriores, uma interface é uma estrutura composta por métodos abstratos. Neste exemplo, vamos considerar uma simples interface chamada **OperacaoAritmetica**:



```
1  OperacaoAritmetica.java
2  public interface OperacaoAritmetica {
3
4      double execute(double value1, double value2);
5
6 }
```

Tal interface abstrai a existência de um método (uma operação aritmética) que recebe dois valores **double**, realiza alguma atividade (a ser implementada por alguma classe) e, por fim, retorna um único valor **double** como resultado.

Um exemplo de implementação para esta interface pode ser visto na classe **Soma**:

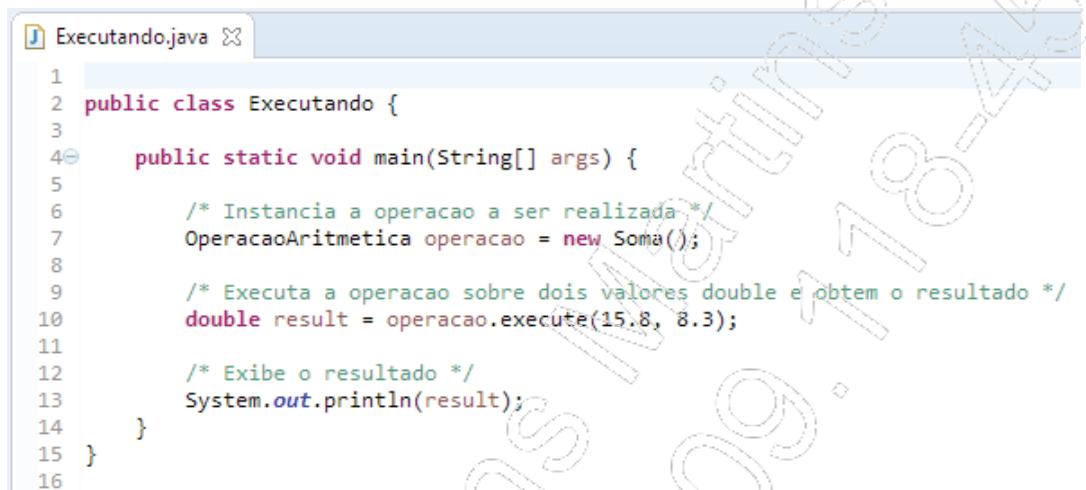


```

1  public class Soma implements OperacaoAritmetica {
2
3     public double execute(double value1, double value2) {
4         double result = value1 + value2;
5         return result;
6     }
7
8 }
9

```

E, dessa forma, podemos instanciar uma operação de soma:



```

1  public class Executando {
2
3     public static void main(String[] args) {
4
5         /* Instancia a operacao a ser realizada */
6         OperacaoAritmetica operacao = new Soma();
7
8         /* Executa a operacao sobre dois valores double e obtem o resultado */
9         double result = operacao.execute(15.8, 8.3);
10
11        /* Exibe o resultado */
12        System.out.println(result);
13    }
14
15 }
16

```

Neste exemplo, adotamos a abordagem tradicional em que abstraímos a existência de entidades (classes) responsáveis pela execução de operações aritméticas.

Criamos a classe **Soma** como uma implementação de operação aritmética e podemos instanciá-la a qualquer momento.

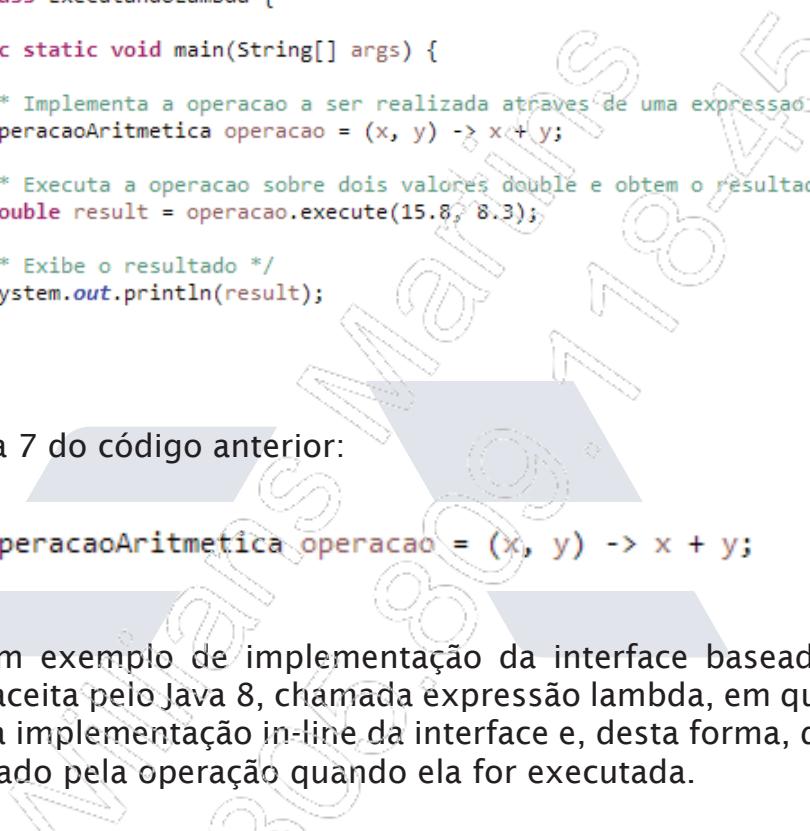
Seguindo este conceito, para cada operação aritmética que possa ser utilizada pela aplicação, devemos criar uma nova classe implementadora conforme a necessidade (soma, divisão, potência etc.).

Rpare, no última código (classe **Executando**), que o objeto **operacao** não tem atributos (objeto sem estado) e possui um único método, chamado **execute()**. É esta forma de desenvolvimento que chamamos de programação funcional. Neste formato de programação, encapsulamos as rotinas de execução (funções) em simples objetos que podem ser chamados posteriormente a qualquer momento.

Adicionalmente, o Java 8 incorporou uma nova sintaxe de comandos para elaboração destes objetos funcionais.

No código adiante, considere a mesma interface **OperacaoAritmetica** já mencionada, mas vamos ignorar a existência da classe **Soma** e supor que não existe ainda nenhuma implementação daquela interface.

Assim sendo, podemos realizar uma implementação imediata por meio da nova notação e, em seguida, utilizar o objeto funcional normalmente:



```
ExecutandoLambda.java
1  public class ExecutandoLambda {
2
3     public static void main(String[] args) {
4
5         /* Implementa a operacao a ser realizada atraves de uma expressao lambda */
6         OperacaoAritmetica operacao = (x, y) -> x + y;
7
8         /* Executa a operacao sobre dois valores double e obtém o resultado */
9         double result = operacao.execute(15.8, 8.3);
10
11        /* Exibe o resultado */
12        System.out.println(result);
13
14    }
15
16 }
```

Observe a linha 7 do código anterior:

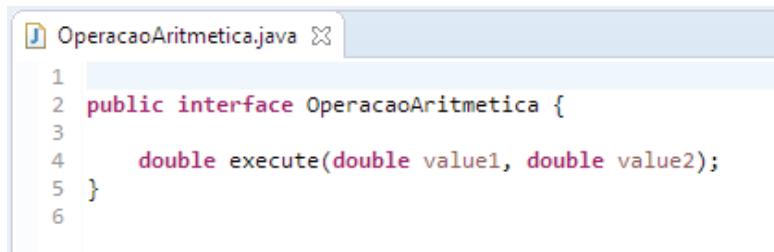
`OperacaoAritmetica operacao = (x, y) -> x + y;`

Nela, temos um exemplo de implementação da interface baseada na nova nomenclatura aceita pelo Java 8, chamada expressão lambda, em que estamos realizando uma implementação in-line da interface e, desta forma, definindo o que será realizado pela operação quando ela for executada.

No Java, a programação funcional consiste na implementação e utilização de interfaces baseadas nesta nova nomenclatura, tema que abordaremos ao longo deste capítulo.

14.2. Interface funcional

Chamamos de interface funcional aquela que possui somente um método abstrato (sem corpo), exigindo a implementação de apenas um método pela classe responsável.



```
J OperacaoAritmetica.java X
1
2 public interface OperacaoAritmetica {
3
4     double execute(double value1, double value2);
5
6 }
```

Uma interface funcional pode possuir vários métodos estáticos, privados ou default, porém, somente um único método abstrato.

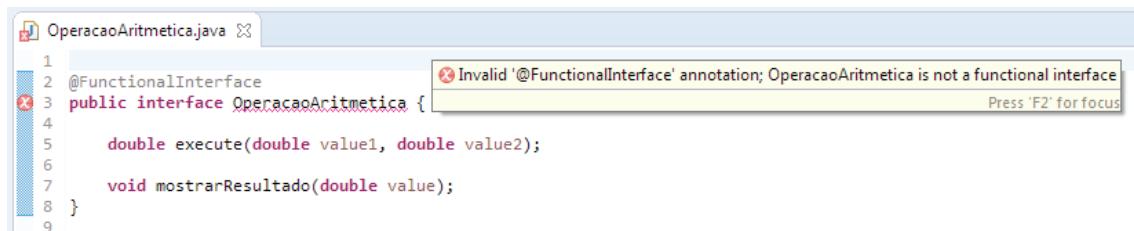
O método abstrato, por sua vez, pode possuir qualquer tipo de retorno, podendo até não haver retorno (**void**) e também possuir quaisquer quantidades e tipos de argumentos (parâmetros de entrada). O uso de genéricos também é permitido.

Esse tipo de interface é amplamente utilizado na programação funcional, pois é através dela que definimos a assinatura das funções que serão utilizadas ao longo do código fonte.

14.2.1. A anotação **@FunctionalInterface**

Embora não seja necessário, podemos utilizar a anotação **@FunctionalInterface** para que o compilador Java verifique se a interface é funcional ou não. Ela é especialmente útil em interfaces que sofrem muitas alterações durante o ciclo de desenvolvimento do projeto.

Ao tentar adicionar um segundo método abstrato em uma interface marcada como **@FunctionalInterface**, o compilador exibirá uma mensagem de erro:



```
J OperacaoAritmetica.java X
1
2 @FunctionalInterface
3 public interface OperacaoAritmetica {
4
5     double execute(double value1, double value2);
6
7     void mostrarResultado(double value);
8
9 }
```

Invalid '@FunctionalInterface' annotation; OperacaoAritmetica is not a functional interface
Press 'F2' for focus

14.2.2. Exemplos de interface funcional

Embora o conceito de interface funcional seja uma novidade no Java, algumas interfaces que já existiam na linguagem seguem esta especificação.

São exemplos de interfaces funcionais nativas da linguagem:

- **java.lang.Runnable**: Utilizada na criação de threads, assunto que será abordado posteriormente. Possui um único método, sem retorno (**void**) e sem argumentos, chamado **run()**;
- **java.util.Comparator**: Utilizada em coleções ordenadas como critério de ordenação. Possui um único método chamado **compare()** com dois argumentos genéricos e tipo de retorno **int**;
- **javax.swing.ActionListener**: Utilizada em aplicações gráficas para associar o evento de clique (ou ativação) a um componente de tela, como um botão ou caixa de texto. Possui um único método chamado **actionPerform()**, que possui um argumento auxiliar de tipo **java.awt.event.ActionEvent** e valor de retorno **void**.

Além disso, a versão 8 do Java traz o novo package **java.util.function** contendo inúmeras outras interfaces funcionais que foram criadas especificamente para auxiliar a programação funcional.

14.3. Expressões lambda

Uma expressão lambda, também chamada de função anônima, trata-se de código utilizado na implementação de uma interface funcional baseada em uma nova nomenclatura dentro da linguagem Java.

```
OperacaoAritmetica operacao = (x, y) -> x + y;
```

Uma expressão lambda sempre deve ser atribuída a uma variável de tipo funcional (interface funcional), como no exemplo anterior, ou passada como argumento na chamada de um método que aceita um tipo funcional, como no exemplo a seguir:

```
servico.realizaOperacao((x, y) -> x + y);
```

14.3.1. Forma geral

De forma geral, uma expressão lambda possui duas seções separadas pelo símbolo “->” (menos-maior):

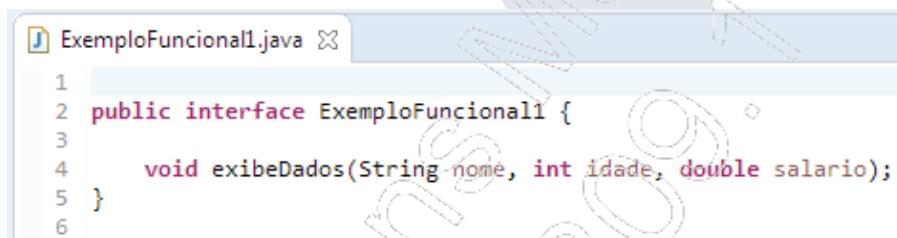
```
(arg1, arg2, ..., argn) -> <expressão de cálculo ou retorno>
```

Em que temos, ao lado esquerdo, os parâmetros a serem recebidos pelo método funcional e, ao lado direito, o conjunto de instruções que implementam este método.

14.3.2. Expressões com parâmetros

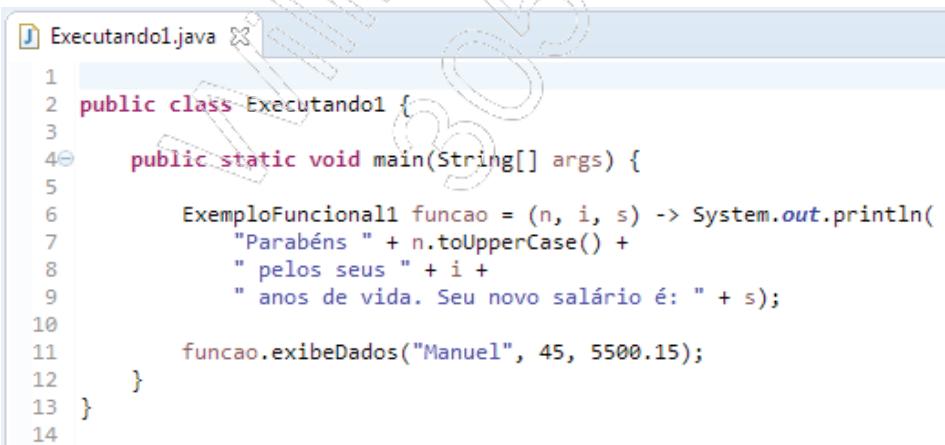
Em uma expressão lambda, todos os parâmetros definidos no método funcional da interface devem ser declarados entre parênteses, à esquerda do símbolo “->” (menos-maior), na mesma ordem em que foram definidos no método.

- **Interface**



```
ExemploFuncional1.java
1
2 public interface ExemploFuncional1 {
3
4     void exibeDados(String nome, int idade, double salario);
5
6 }
```

- **Expressão**



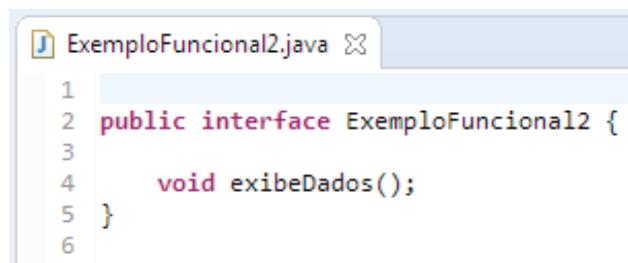
```
Executando1.java
1
2 public class Executando1 {
3
4     public static void main(String[] args) {
5
6         ExemploFuncional1 funcao = (n, i, s) -> System.out.println(
7             "Parabéns " + n.toUpperCase() +
8             " pelos seus " + i +
9             " anos de vida. Seu novo salário é: " + s);
10
11         funcao.exibeDados("Manuel", 45, 5500.15);
12     }
13 }
14 }
```

Rpare que a expressão lambda não declara os tipos dos parâmetros de entrada, representados por **(n, i, s)**, no exemplo anterior. Os tipos são sempre aqueles mesmos declarados no método da interface.

14.3.3. Expressões sem parâmetros

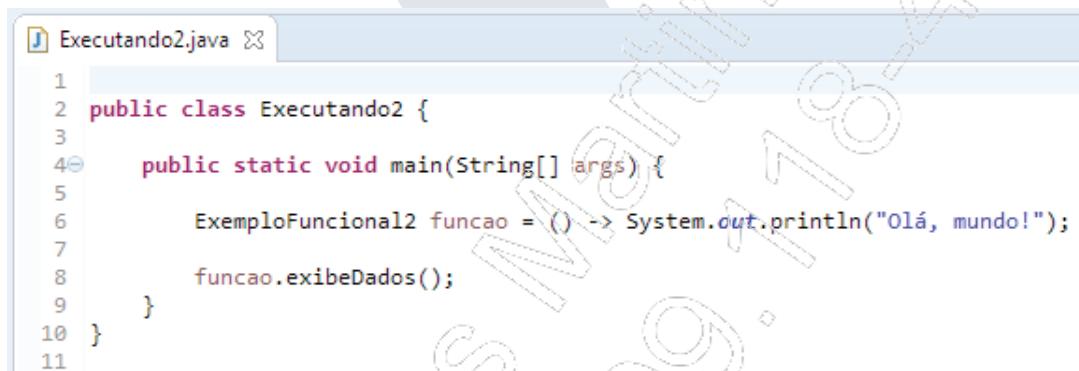
Expressões sem parâmetro devem ser declaradas com parênteses vazios.

- Interface



```
J ExemploFuncional2.java X
1
2 public interface ExemploFuncional2 {
3
4     void exibeDados();
5
6 }
```

- Expressão

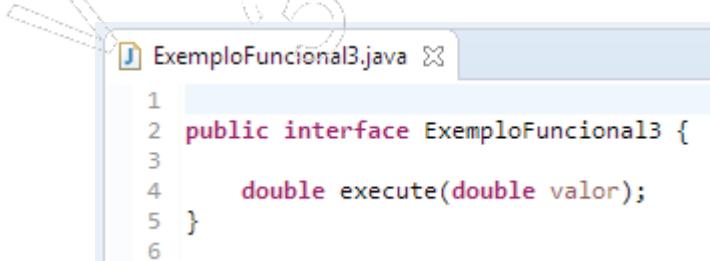


```
J Executando2.java X
1
2 public class Executando2 {
3
4     public static void main(String[] args) {
5
6         ExemploFuncional2 funcao = () -> System.out.println("Olá, mundo!");
7
8         funcao.exibeDados();
9     }
10 }
11
```

14.3.4. Expressões com um único parâmetro

Se a expressão possui apenas um parâmetro, então, os parênteses são opcionais.

- Interface



```
J ExemploFuncional3.java X
1
2 public interface ExemploFuncional3 {
3
4     double execute(double valor);
5
6 }
```

- **Expressão**

```
1 public class Executando3 {
2     public static void main(String[] args) {
3         ExemploFuncional3 funcao = d -> Math.sqrt(d);
4         double result = funcao.execute(144);
5         System.out.println("Resultado: " + result);
6     }
7 }
8
9
10
11 }
```

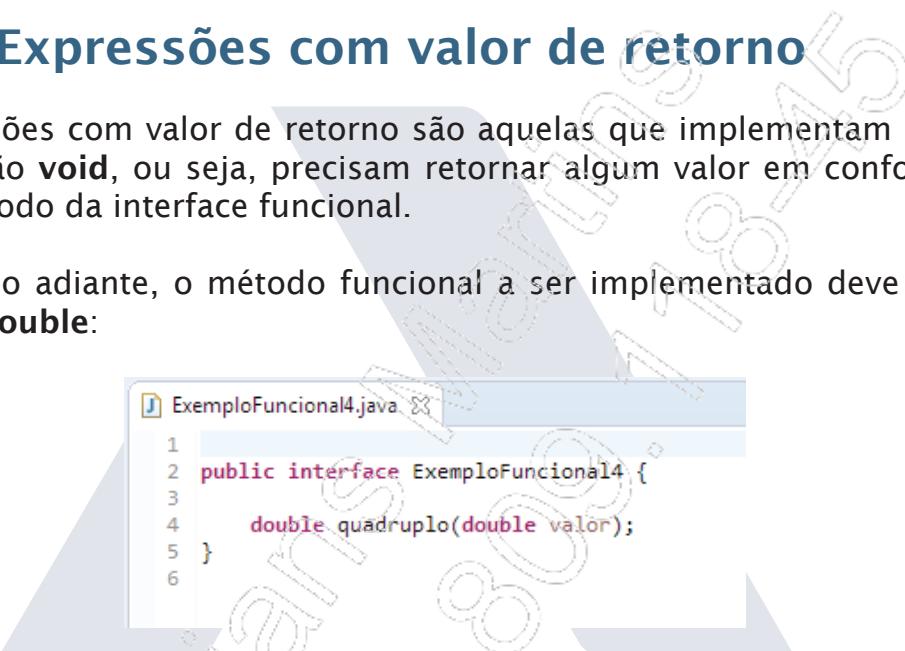
14.3.5. Corpo da expressão lambda

Todo o código após o símbolo “->” (menos-maior) é considerado como “corpo” da expressão. Aqui, podemos utilizar normalmente quaisquer instruções válidas da linguagem Java.

Quando temos duas ou mais instruções no corpo da expressão, devemos utilizar chaves em volta do corpo e ponto e vírgula ao final de cada instrução:

```
J Excutando4.java x
1
2 public class Executando4 {
3
4     public static void main(String[] args) {
5
6         ExemploFuncional1 função = (n, i, s) -> {
7             String nomeMaiusculo = n.toUpperCase();
8             if (i < 18) {
9                 System.out.println("Funcionário não pode ser menor de idade!");
10            } else {
11                System.out.println("Funcionário " + nomeMaiusculo
12                    + " recebe salário " + s);
13            }
14        };
15
16        função.exibeDados("Manuel", 45, 5500.15);
17    }
18 }
```

Expressões que possuem uma única instrução não necessitam de chaves:

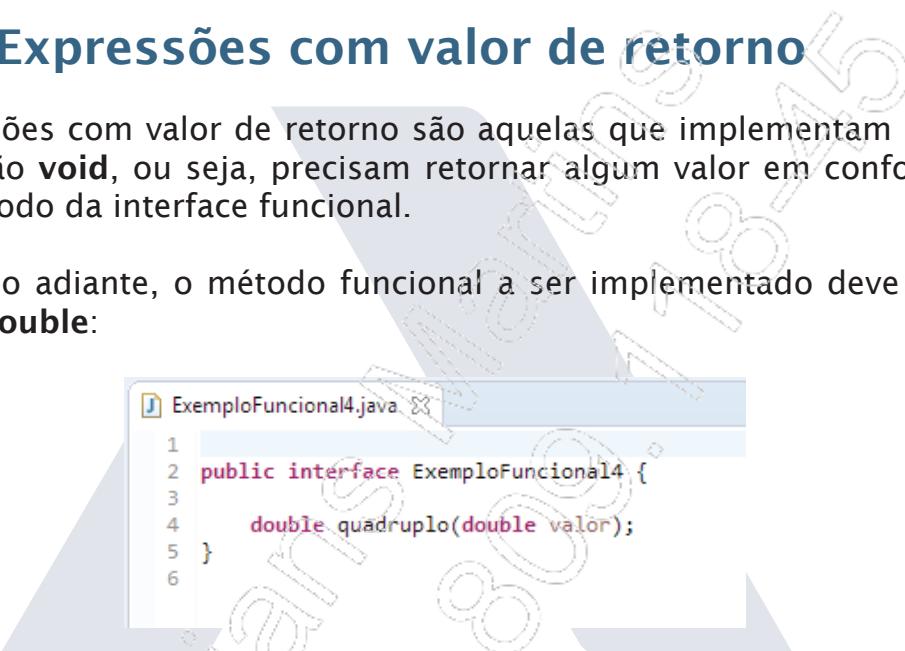


```
J Executando5.java X
1
2 public class Executando5 {
3
4     public static void main(String[] args) {
5
6         ExemploFuncional1 funcao = (n, i, s) -> System.out.println(
7             "Funcionario " + n + " recebe salário " + s);
8
9         funcao.exibeDados("Manuel", 45, 5500.15);
10    }
11 }
12 }
```

14.3.6. Expressões com valor de retorno

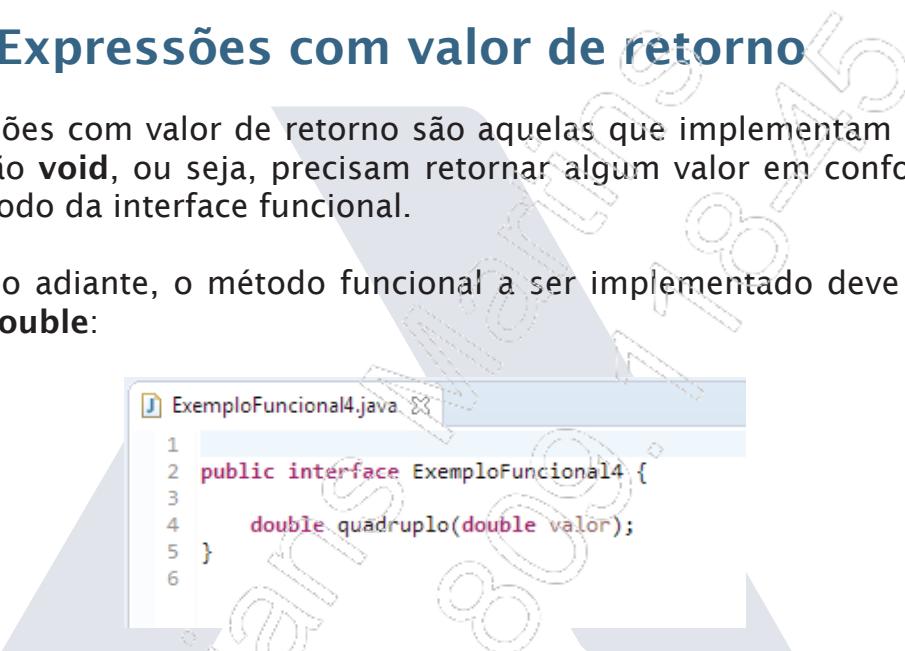
As expressões com valor de retorno são aquelas que implementam métodos que não são **void**, ou seja, precisam retornar algum valor em conformidade com o método da interface funcional.

No exemplo adiante, o método funcional a ser implementado deve retornar um valor **double**:



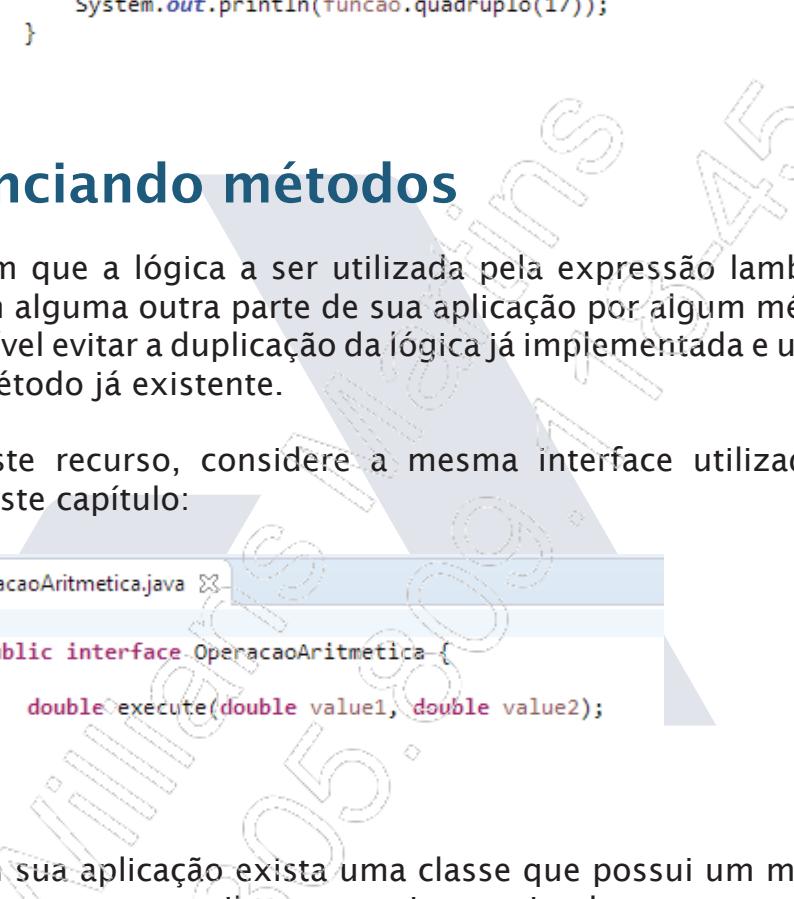
```
J ExemploFuncional4.java X
1
2 public interface ExemploFuncional4 {
3
4     double quadruplo(double valor);
5
6 }
```

Se a expressão lambda utilizada possuir diversas linhas, ela poderá ser implementada normalmente utilizando a cláusula **return** ao final do código:



```
J Executando6.java X
1
2 public class Executando6 {
3
4     public static void main(String[] args) {
5
6         ExemploFuncional4 funcao = val -> {
7             double temp = val * 4;
8             return temp;
9         };
10        System.out.println(funcao.quadruplo(8));
11    }
12 }
13 }
14 }
```

Mas, se a expressão lambda for pequena, composta por uma única instrução, podemos simplesmente indicar a instrução de retorno após o símbolo “->” (menos-maior), sem chaves e sem a cláusula **return**:



```

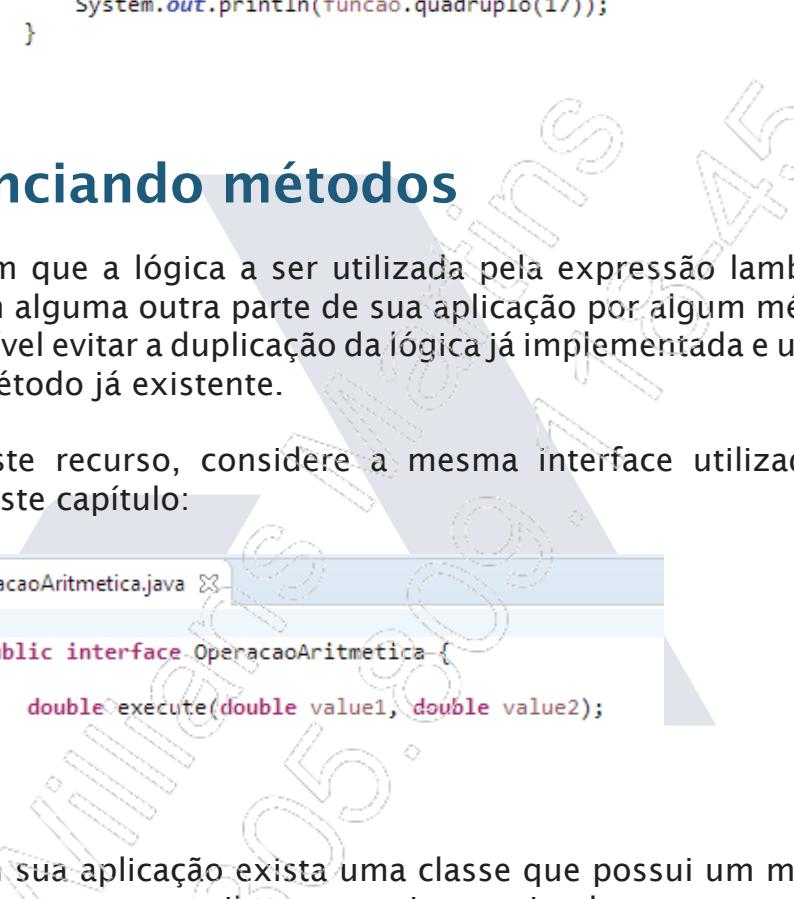
1 Executando7.java X
2 public class Executando7 {
3
4     public static void main(String[] args) {
5         ExemploFuncional4 funcao = val -> val * 4;
6         System.out.println(funcao.quadruplo(17));
7     }
8 }
9
10
11

```

14.4. Referenciando métodos

Existem situações em que a lógica a ser utilizada pela expressão lambda já foi implementada em alguma outra parte de sua aplicação por algum método. Nesses casos, é possível evitar a duplicação da lógica já implementada e utilizar uma referência ao método já existente.

Para exemplificar este recurso, considere a mesma interface utilizada no primeiro exemplo deste capítulo:

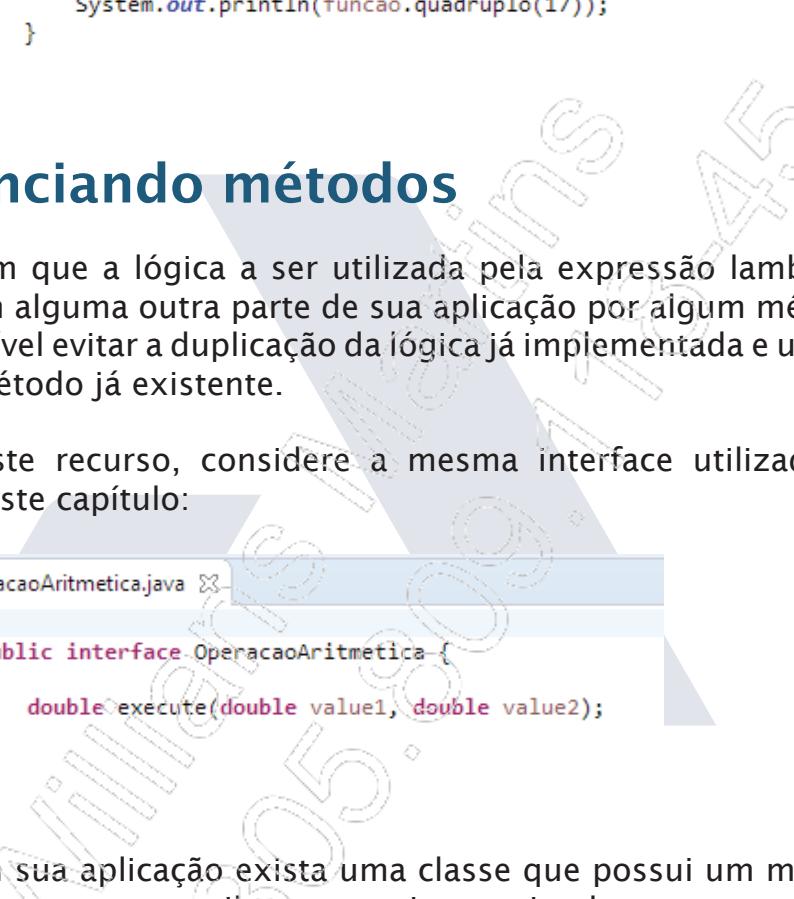


```

1 OperacaoAritmetica.java X
2 public interface OperacaoAritmetica {
3
4     double execute(double value1, double value2);
5 }
6

```

Suponhamos que em sua aplicação exista uma classe que possui um método pronto que realiza exatamente aquilo que precisamos implementar em nossa função:



```

1 FinancialUtils.java X
2 public class FinancialUtils {
3
4     public static double calculaJurosCompostos(double valorInicial, double taxaJuros, int prazo) {
5
6     public static double calculaJuros(double valorInicial, double taxaJuros) {
7         double razao = taxaJuros / 100;
8         double juros = valorInicial * razao;
9         return valorInicial + juros;
10    }
11
12    public static double calculaAmortizacao(double valorTotal, double valorPrestacao, int prazo) {
13
14    public static int calculaPrazo(double valorTotal, double valorPrestacao, double taxaJuros) {
15
16
17
18
19
20

```

Em situações como esta podemos “linkar”, ou criar uma referência deste método, para que ela seja utilizada como a função que queremos implementar:

```
Executando8.java X
1
2 public class Executando8 {
3
4@  public static void main(String[] args) {
5
6     OperacaoAritmetica funcao = FinancialUtils::calculaJuros;
7
8     System.out.println("Valor final: " + funcao.execute(1200.0, 8.3));
9 }
10}
11
```

No exemplo anterior, utilizamos a seguinte notação para criar uma referência funcional a um método estático de alguma classe pré-existente de nossa aplicação:

```
<nome da classe>::<nome do método>
```

Para referenciar um método de instância (não estático), devemos utilizar a seguinte notação, conforme o próximo exemplo:

```
<nome do objeto>::<nome do método>
```

Considere que estamos utilizando a mesma interface funcional **OperacaoAritmetica**, mas queremos referenciar, desta vez, um método não estático de alguma outra classe:

```
Produto.java X
1
2 public class Produto {
3
4     private String descricao;
5     private int tamanho;
6
7@  public Produto(String descricao, int tamanho) {}
8
9@  public double aumentarPreco(double precoAtual, double taxaAumento) {
10    double aumento = precoAtual * taxaAumento / 100;
11    double precoAumentado = precoAtual + aumento;
12    return precoAumentado;
13}
14
15@  public String getDescricao() {}
16
17@  public void setDescricao(String descricao) {}
18
19@  public int getTamanho() {}
```

Assim sendo, podemos referenciar o método desejado da seguinte forma:

```
Executando9.java X
1  public class Executando9 {
2      public static void main(String[] args) {
3          /* Cria uma instancia da classe que contem o metodo desejado. */
4          Produto produto = new Produto("Sapato social", 42);
5
6          /* Utiliza o metodo aumentarPreco() como referencia para nossa funcao. */
7          OperacaoAritmetica funcao = produto::aumentarPreco;
8
9          /* Executa a funcao e exibe o resultado. */
10         System.out.println("Valor final: " + funcao.execute(65.5, 8.5));
11     }
12 }
13 }
```

Referenciando métodos

O método a ser referenciado (estático ou não) precisa ter exatamente a mesma assinatura exigida pela interface funcional, isto é, os mesmos tipos de parâmetros de entrada e mesmo tipo de retorno.

14.5. O pacote `java.util.function`

As expressões lambda (EL) foram criadas para auxiliar diversas APIs do Java no processamento de coleções, programação concorrente e em muitos outros aspectos.

A fim de suportar o uso de ELs, o Java 8 traz uma vasta quantidade de interfaces funcionais predefinidas que facilitam a implementação destas rotinas de processamento. Tais interfaces foram disponibilizadas em um novo package denominado `java.util.function`.

Citamos, a seguir, algumas das principais interfaces funcionais deste pacote:

Interface	Descrição	Método funcional
<code>Consumer<T></code>	Uma função sem retorno que recebe um parâmetro de tipo especificado.	<code>void accept(T)</code>
<code>BiConsumer<T,U></code>	Uma função sem retorno que recebe dois parâmetros de tipos especificados.	<code>void accept(T, U)</code>
<code>Predicate<T></code>	Uma função que recebe um parâmetro de tipo especificado e retorna um <code>boolean</code> .	<code>boolean test(T)</code>
<code>Supplier<T></code>	Uma função sem parâmetros que retorna um valor de tipo especificado.	<code>T get()</code>
<code>UnaryOperator<T></code>	Uma função que recebe um parâmetro de tipo especificado e retorna um valor do mesmo tipo.	<code>T apply(T)</code>
<code>Function<T,R></code>	Uma função que recebe um parâmetro de tipo especificado e retorna um valor de um outro tipo especificado.	<code>R apply(T)</code>
<code>BinaryOperator<T></code>	Uma função que recebe dois parâmetros de mesmo tipo e retorna um valor também do mesmo tipo.	<code>T apply(T, T)</code>
<code>BiFunction<T,U,R></code>	Uma função que recebe dois parâmetros de tipos especificados e retorna um valor de tipo diferente.	<code>R apply(T, U)</code>

Muitas das APIs do Java 8 que utilizam o conceito de programação funcional fazem uso das interfaces mencionadas na tabela, conforme veremos nos próximos capítulos.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- A programação funcional traz um novo paradigma no desenvolvimento de aplicações, permitindo elaborar pequenas rotinas de uso comum a diversos pontos da aplicação, podendo, inclusive, executar métodos passando estas funções como argumentos;
- Chamamos de interface funcional aquela que contém apenas um método a ser implementado e chamamos de função uma implementação sem estado de uma interface funcional;
- O Java 8 permite a utilização de uma metalinguagem denominada notação lambda que permite uma construção mais simplificada de funções em sua aplicação;
- Podemos utilizar a notação lambda para criar referências a métodos, sejam eles estáticos ou não, já existentes na aplicação;
- O novo pacote **java.util.function** traz uma série de interfaces funcionais de uso comum por diversas outras APIs da linguagem.

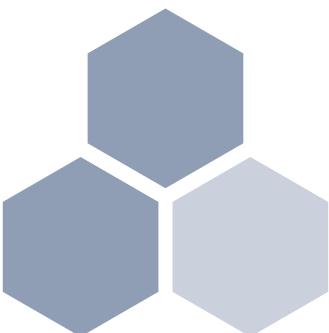


14

Programação funcional

Teste seus conhecimentos

Willian Martins
305.000
45



Editora
IMPACTA



1. Qual das alternativas a seguir define o que é a programação funcional?

- a) Consiste na ampla utilização de entidades que possuem métodos e atributos, em que os atributos representam as suas características e os métodos representam as ações que dependem de seus atributos.
- b) Consiste na elaboração de instruções a serem executadas de forma linear pelo interpretador da linguagem.
- c) Trata-se de uma nova linguagem de programação.
- d) É uma nova metodologia de desenvolvimento em que isolamos rotinas de uso geral em pequenas entidades sem estado (funções), a fim de serem executadas de qualquer ponto da aplicação.
- e) Trata-se de um processo, em segundo plano, responsável pelo gerenciamento de memória das aplicações desenvolvidas em Java.

2. O que é uma interface funcional?

- a) Um conjunto de janelas com botões, caixas de texto e outros componentes gráficos utilizados pela aplicação.
- b) Uma interface contendo diversos métodos abstratos.
- c) Uma interface que contém um único método abstrato.
- d) Uma interface que possui métodos privados de apoio a outros métodos.
- e) Nenhuma das alternativas anteriores está correta.



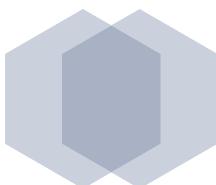
14

Programação funcional



Mãos à obra!

Willian Martins
305.000
45



Editora
IMPACTA



Laboratório 1

Neste laboratório, vamos criar um array contendo diversos salários (array de valores **double**) e utilizar a classe auxiliar **DoubleArrayUtils** para manipular os salários através de expressões lambda.

A – Descontando 10% dos valores salariais

1. Copie a classe **DoubleArrayUtils** fornecida pelo instrutor;
2. Crie uma segunda classe chamada **DescontoSalarial** contendo o método **main** e, dentro dele, crie a variável **salariosBrutos** (do tipo array de valores **double**) contendo os seguintes salários:

Salários
1.350,00
4.320,15
8.235,25
2.500,55
1.830,00
850,26
3.614,29
12.500,00

3. Ainda no método **main()**, crie outra variável array de **double** chamada **salariosLiquidos**;
4. Chame o método estático **DoubleArrayUtils.transformaValores()** passando os seguintes argumentos separados por vírgula:
 - O array de salários brutos criado no passo 2;
 - Uma expressão lambda que recebe um salário e retorna o salário descontado de 10%.
5. O resultado da chamada deste método deverá ser atribuído à variável **salariosLiquidos**. Exiba cada um dos elementos do array **salariosLiquidos**.

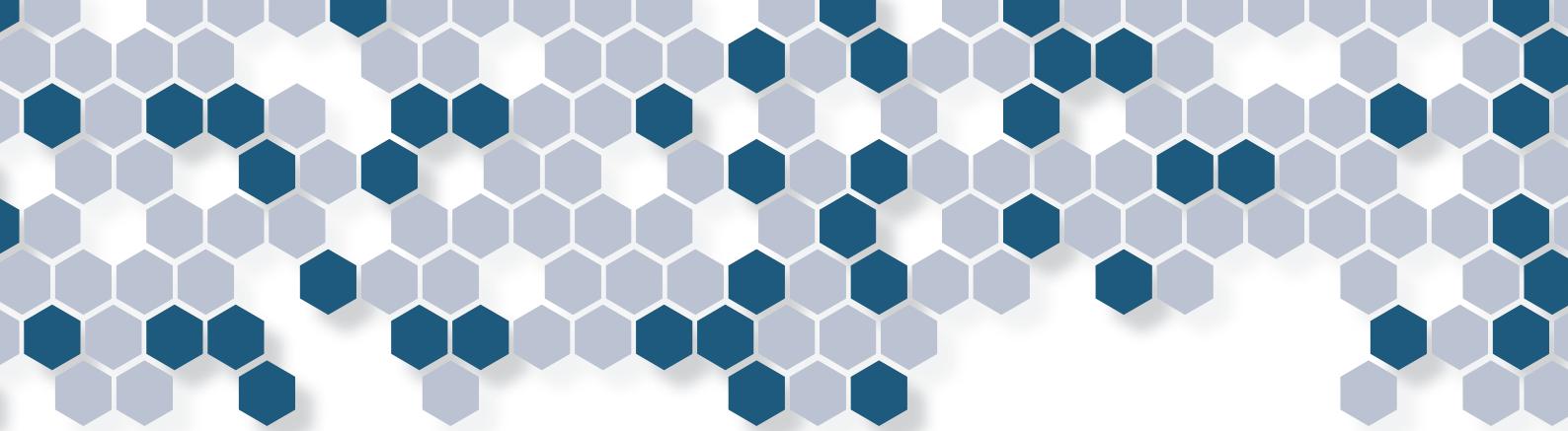
B – Obtendo os salários mais altos

1. Crie outra classe chamada **MaioresSalarios** e, dentro dela, crie o método **main()** com o mesmo array de salários brutos do exercício anterior;
2. Declare um segundo array de **double** chamado **salariosTop**;
3. Chame o método estático **DoubleArrayUtils.filtraValores()**, passando os seguintes argumentos separados por vírgula:
 - O array de salários brutos;
 - Uma expressão lambda que recebe um salário e retorna um valor **booleano**:
 - **true** quando o salário foi maior ou igual a 3.000,00;
 - **false** em caso contrário.
4. Exiba cada um dos elementos do array **salariosTop**.

C – Usando a expressão lambda para exibir os resultados

! Modifique o último passo dos exercícios A e B para utilizar a expressão lambda também na exibição de valores.

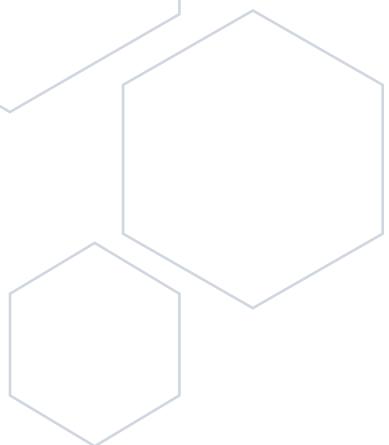
1. Apague o loop utilizado na exibição dos resultados;
2. Chame o método estático **DoubleArrayUtils.processaValores()**, passando os seguintes argumentos separados por vírgula:
 - O array de salários brutos;
 - Uma expressão lambda que recebe um salário e imprime, utilizando a instrução **System.out.println()**.



15

Coleções e mapas

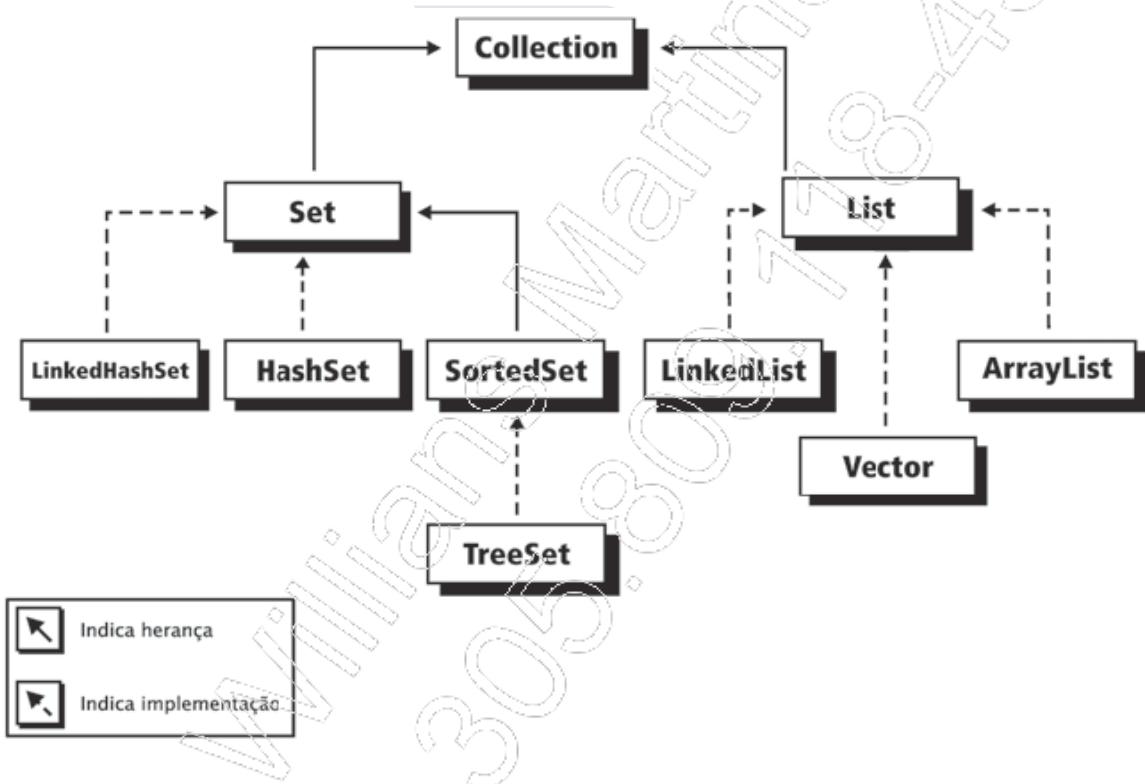
- O que são coleções;
- Principais operações de coleções;
- Principais interfaces das coleções;
- Generics;
- Coleção List;
- Coleção Set;
- Manipulando coleções com Streams;
- Interface Map;
- Collections Framework.



15.1. O que são coleções

A estrutura de coleções foi lançada na primeira versão da linguagem Java 2 (JDK1.2) e se expandiu nas versões 1.4 (Java 4) e Java 8. Ela contém mapas, conjuntos e listas que são utilizados na codificação para suprir as necessidades das aplicações Java. Para personalizar essa aplicação, o pacote `java.util` disponibiliza diversos utilitários e interfaces.

As coleções estão divididas em três hierarquias básicas: listas (**List**), mapas (**Map**) e conjuntos (**Set**). As listas referem-se às listas de itens, nas quais estão as classes que implementam a interface **List**. Em mapas, estão os itens que possuem identificação exclusiva, cujas classes implementam a interface **Map**. Já conjuntos referem-se a itens exclusivos, cujas classes implementam a interface **Set**.



15.2. Principais operações de coleções

Além das operações básicas, como adição e remoção de objetos na coleção, outras operações são realizadas:

- Verificação da existência de um grupo de objetos, ou somente um objeto, na coleção;
- Iteração pela coleção, sendo que cada objeto ou elemento é verificado, um em seguida do outro;
- Recuperação de um objeto na coleção sem que, para isso, ele seja removido.

O processo de iteração refere-se a percorrer os elementos de uma coleção seguindo-se uma sequência iniciada pelo primeiro elemento.

15.3. Principais interfaces das coleções

A hierarquia aplicada à API de coleções apresenta diversas interfaces, algumas são apresentadas a seguir:

- **Collection**;
- **List**;
- **Set**;
- **Map**;
- **Sorted Set**;
- **Sorted Map**.

As interfaces **Set** e **List** derivam de **Collection**, porém, o mesmo não acontece com as interfaces e classes relacionadas a mapas. Com relação às classes de implementação, elas são divididas nas categorias: conjuntos, mapas e listas.

15.3.1. Características das classes de implementação

A seguir, são apresentadas características das classes de implementação de interfaces de coleções:

- **Mapa (Map)**

Classe	Ordenada	Classificada
Hashtable	Não	Não
HashMap	Não	Não
LinkedHashMap	Ordenada por ordem de último acesso ou de inserção.	Não
TreeMap	Classificada	Classificada pelas regras personalizadas de comparação ou pela ordem natural.

- **Conjunto (Set)**

Classe	Ordenada	Classificada
HashSet	Não	Não
LinkedHashSet	Ordenada por ordem de último acesso ou de inserção.	Não
TreeSet	Classificada	Classificada pelas regras personalizadas de comparação ou pela ordem natural.

- **Lista (List)**

Classe	Ordenada	Classificada
ArrayList	Por índice	Não
LinkedList	Por índice	Não
Vector	Por índice	Não

Em uma implementação, a classificação corresponde a um tipo de ordenamento específico. Uma classe de implementação pode receber uma das seguintes características:

- Indexada e classificada;
- Não indexada e não classificada;
- Indexada e não classificada.

Considere, por exemplo, a classe **LinkedHashSet**, que é indexada. A ordem dos elementos na coleção é definida pela ordem de inserção dos elementos, ou seja, o último elemento inserido no conjunto ocupa a última posição.

A classificação define de que forma os elementos estão ordenados na coleção (sequência de recuperação dos elementos). A classificação ou ordenamento natural de um objeto é definida pela sua classe. Dessa forma, a ordem natural de objetos **Integer** é definida pelo valor numérico, em que 1 vem antes de 2, 5 vem antes de 10, e assim por diante.

As coleções que no momento da inserção aplicam uma determinada classificação nos elementos são chamadas de coleções classificadas.

15.4. Generics

Generics (ou genéricos) são um recurso que permite tornar tipos em parâmetros para a definição de classes, métodos e interfaces. Isso permite que você reutilize o mesmo código em diferentes entradas. Código construído com Generics possui as seguintes vantagens em relação a estruturas de código que não os utilizem:

- A verificação de tipos em tempo de compilação é mais forte e, dessa forma, transfere-se um grande problema, cuja recorrência é comum em tempo de execução, para o tempo de compilação;
- A eliminação de casts;
- A possibilidade de implementar algoritmos genéricos, de fácil leitura, tipagem segura e que podem ser usados em coleções de diferentes tipos.

15.4.1. Tipos genéricos

Tipos genéricos são classes ou interfaces parametrizadas de acordo com tipos. Para entender melhor o conceito de tipo genérico, acompanhe o exemplo a seguir:

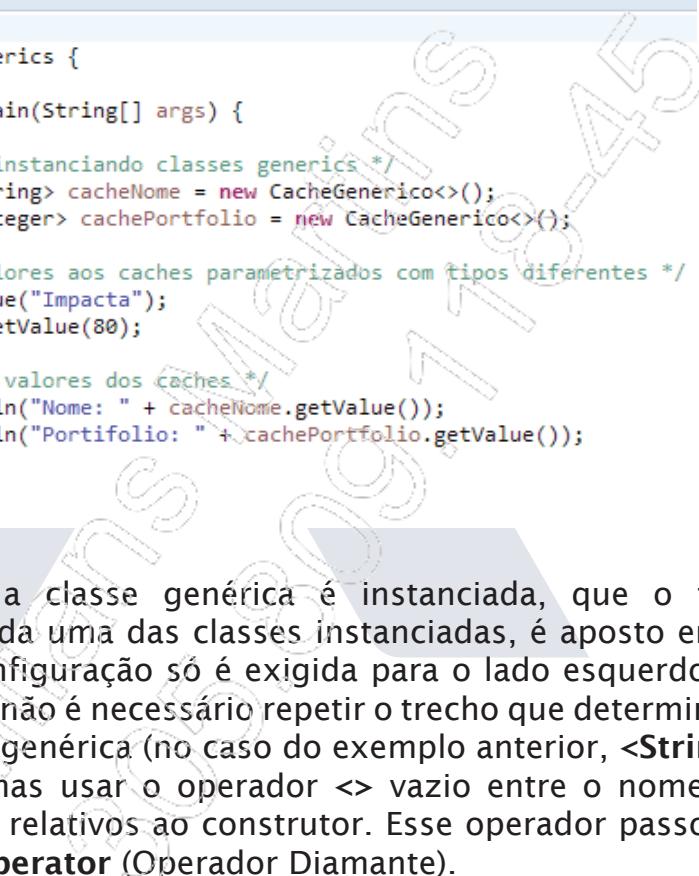
```
1  public class CacheGenerico<T> {
2
3     private T value;
4
5     public T getValue() {
6         return value;
7     }
8
9     public void setValue(T value) {
10        this.value = value;
11    }
12 }
13
14
```

Os nomes de parâmetros de tipo seguem uma convenção para diferenciá-los das variáveis de tipo. Eles são sempre compostos por uma única letra maiúscula. Os nomes de parâmetros mais comuns são os seguintes: E – Element; K – Key; N – Number; T – Type e V – Value.

Os tipos genéricos podem ser invocados quando for necessário referenciar uma classe genérica de dentro do código. Invocar um tipo genérico é como invocar um método, a única diferença é que, em vez de passar um argumento a um método, você passa um argumento de tipo à classe genérica em questão.

Classes compostas por um tipo genérico são conhecidas como tipos parametrizados.

Para criar uma instância da classe genérica, você deve usar a palavra-chave **new** e declarar o tipo (fechado com sinais de maior e menor, **<>**) entre o nome da classe e os parênteses, da seguinte maneira:



```
1  public class TestandoGenerics {
2
3     public static void main(String[] args) {
4
5         /* Declarando e instanciando classes genéricas */
6         CacheGenerico<String> cacheNome = new CacheGenerico<>();
7         CacheGenerico<Integer> cachePortfolio = new CacheGenerico<>();
8
9
10        /* Atribuindo valores aos caches parametrizados com tipos diferentes */
11        cacheNome.setValue("Impacta");
12        cachePortfolio.setValue(80);
13
14        /* Imprimindo os valores dos caches */
15        System.out.println("Nome: " + cacheNome.getValue());
16        System.out.println("Portifolio: " + cachePortfolio.getValue());
17    }
18 }
19
```

Note, nas linhas em que a classe genérica é instanciada, que o tipo parametrizado, usado em cada uma das classes instanciadas, é aposto entre os sinais “<” e “>” e essa configuração só é exigida para o lado esquerdo da atribuição. A partir do Java 7, não é necessário repetir o trecho que determina o tipo a ser utilizado na classe genérica (no caso do exemplo anterior, **<String>** e **<Integer>**), bastando apenas usar o operador **<>** vazio entre o nome da classe e o par de parênteses relativos ao construtor. Esse operador passou a ser chamado de **Diamond Operator** (Operador Diamante).

15.5. Coleção List

Os objetos que implementam a interface **List** permitem obter controle do local em que os elementos são inseridos. Isso é possível porque a interface **List** define uma sequência, ou seja, uma coleção indexada.

Com isso, um índice inteiro é utilizado para acessar os elementos. Estes podem receber valor **null** e podem ser duplicados, o que é permitido pelas classes que implementam a interface **List**.

Na coleção **List**, o índice é muito importante e, por isso, existem diversos métodos relacionados a ele. Ao adicionar um elemento em uma lista sem informar um índice, este é posicionado após o último elemento, porém é possível adicionar elementos em um índice específico.

Confira, a seguir, três implementações importantes de **List**, ordenadas pela posição do índice:

- **ArrayList**

Esta lista foi implementada na versão 1.4 de Java e diz respeito a um array que pode aumentar de tamanho dinamicamente. Ele não é classificado, mas é indexado.

Você deve utilizar **ArrayList** quando a demanda por percorrer frequentemente todos os elementos for maior que a demanda por inserções ou exclusões (principalmente nos primeiros elementos da lista).

- **LinkedList**

No **LinkedList**, o encadeamento duplo existente entre os elementos fornece métodos novos para que as tarefas de inserção e remoção no início ou no fim do conjunto sejam realizadas.

Para realizar inserções, bem como exclusões com rapidez, utilize esse tipo de lista. Contudo, é preciso ressaltar que o processo de iteração de seus elementos pode não ser tão rápido se comparado ao **ArrayList**.

- **Vector**

O **Vector** é uma coleção bem antiga, lançada com a primeira versão da linguagem Java. Esta coleção é semelhante ao **ArrayList**, porém, em **Vector()**, os métodos são sincronizados para que as **threads** sejam seguras.

Em razão do impacto que os métodos sincronizados causam no desempenho, é melhor utilizar o **ArrayList** em vez do **Vector**, quando o acesso concorrente não for um problema relevante.

Observe o exemplo a seguir, que ilustra alguns métodos presentes em coleções que implementam a interface **List**:

```
J ExemploArrayList.java X
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class ExemploArrayList {
5
6     public static void main(String[] args) {
7
8         List<String> paises = new ArrayList<>();
9
10        paises.add("Brasil");
11        paises.add("Portugal");
12        paises.add("Estados Unidos");
13        paises.add("França");
14        paises.add("Inglaterra");
15        paises.add(2, "Itália");
16
17        System.out.println("Países na ordem original:");
18        paises.forEach(curso -> System.out.println(" - " + curso));
19
20        System.out.println("===== ");
21
22        /* Verifica se esta lista possui o String "Estados Unidos" */
23        System.out.println(paises.contains("Estados Unidos"));
24
25        /* Imprime o item na posição 2 */
26        System.out.println(paises.get(2));
27
28        /* Imprime o tamanho desta lista */
29        System.out.println(paises.size());
30
31        /* Imprime em que posição da lista está a "França" */
32        System.out.println(paises.indexOf("França"));
33
34        /* Substitui todos os elementos pelo seu conteúdo em maiúsculo */
35        paises.replaceAll(s -> s.toUpperCase());
36
37        /* Reordena todo a lista utilizando a ordem natural de Strings */
38        paises.sort((x, y) -> x.compareTo(y));
39
40        System.out.println("===== ");
41
42        System.out.println("Países ordenados e em maiúsculo:");
43        paises.forEach(curso -> System.out.println(" - " + curso));
44    }
45 }
```

O resultado é mostrado a seguir:

```
Console Problems Javadoc
<terminated> ExemploArrayList [Java Application]
Países na ordem original:
- Brasil
- Portugal
- Itália
- Estados Unidos
- França
- Inglaterra
=====
true
Itália
6
4
=====
Países ordenados e em maiúsculo:
- BRASIL
- ESTADOS UNIDOS
- FRANÇA
- INGLATERRA
- ITÁLIA
- PORTUGAL
```

15.6. Coleção Set

Consiste em um conjunto que não possui elementos duplicados. Isto quer dizer que, se dois objetos verificados pelo método `equals()` forem considerados equivalentes, somente um deles poderá permanecer na coleção. A coleção **Set** possui diversas implementações. A seguir, apresentamos as características das mais importantes:

- **HashSet**

A classe **HashSet** é um conjunto não indexado e não classificado, que não permite duplicações entre seus elementos. Você deve usá-la se a ordem na iteração não for necessária. A **HashSet** utiliza o código de hashing do objeto que está sendo inserido, fazendo com que o desempenho obtido na busca de um elemento seja tanto melhor quanto melhor for a implementação da geração de um código hash, realizado através da reescrita do método `hashCode()` herdado da classe **Object**.

Não é aconselhável utilizar **HashSet** se a quantidade de elementos for muito grande e for necessário realizar navegações sequenciais pelos elementos frequentemente.

- **LinkedHashSet**

Quando for necessário estabelecer uma ordem na iteração, utilize a classe **LinkedHashSet**, na qual os elementos são iterados na mesma ordem em que foram inseridos. Esta classe é uma versão indexada de **HashSet**.

A **LinkedHashSet** mantém uma lista com encadeamento duplo para todos os elementos. Os elementos podem ser iterados pela ordem do último elemento acessado, o que significa que a ordem de iteração dos elementos pode ser invertida.

- **TreeSet**

Este é um conjunto classificado, em que os elementos permanecem em sequência ascendente, conforme a ordem natural definida pela classe dos elementos pertencentes à árvore. Para alterar a ordem dos elementos, basta utilizar um construtor que permita definir uma classificação diferente da natural para seus elementos.

15.6.1. Classe Iterator

Para navegar pelos elementos, utilize a classe **Iterator**. Ela interage com classes não indexadas e possui um método chamado **iterator()**, o qual retorna um objeto da classe **Iterator** e é definido pela interface **Set**. Veja os métodos da classe **Iterator**:

- **boolean hasNext()**

Este método retorna **true** se a iteração tiver mais elementos e **false** caso contrário.

- **E next()**

Este método retorna o próximo elemento na iteração. Se a iteração não tiver mais elementos, lança **NoSuchElementException**. Perceba que seu retorno é parametrizado com “E”, ou seja, o tipo é o que foi definido no momento da declaração da coleção.

- **void remove()**

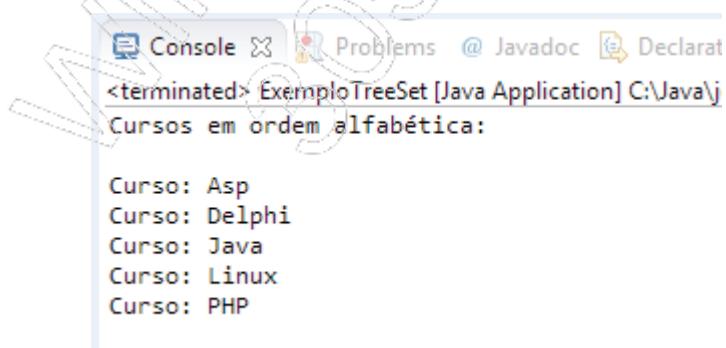
O elemento corrente da iteração é removido da coleção.

O exemplo adiante utiliza a classe **TreeSet** que garante que a ordem dos elementos seja ascendente, tomando por base sua ordem natural:



```
1 import java.util.Iterator;
2 import java.util.TreeSet;
3
4 public class ExemploTreeSet {
5
6     public static void main(String[] args) {
7
8         TreeSet<String> cursos = new TreeSet<>();
9
10        cursos.add("PHP");
11        cursos.add("Linux");
12        cursos.add("Asp");
13        cursos.add("Java");
14        cursos.add("Delphi");
15
16        Iterator<String> iterator = cursos.iterator();
17        System.out.println("Cursos em ordem alfabética: \n");
18
19        while (iterator.hasNext()) {
20            String curso = iterator.next();
21            System.out.println("Curso: " + curso);
22        }
23    }
24 }
25
```

Depois de compilado e executado o código anterior, o resultado será como o exibido na figura a seguir:



```
Console Problems @ Javadoc Declarat
<terminated> ExemploTreeSet [Java Application] C:\Java\j
Cursos em ordem alfabética:

Curso: Asp
Curso: Delphi
Curso: Java
Curso: Linux
Curso: PHP
```

15.6.2. Equivalência de objetos (`equals()`)

Para comparar o conteúdo de dois objetos, ou seja, para comparar se dois objetos são significativamente equivalentes, utilize o método `equals()`. Use o operador `==` apenas para verificar se um mesmo objeto é referenciado por duas variáveis distintas, ou seja, se apontam para um mesmo endereço de memória.

No momento em que as classes são criadas, é preciso definir o que determina que dois objetos distintos sejam significativamente equivalentes. Para fazer a comparação entre dois objetos distintos, o método `equals()` da classe deve ser invocado. Desse modo, é possível comparar as instâncias de uma classe.

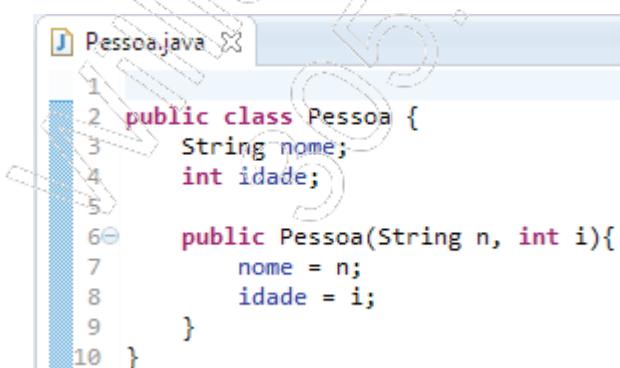
O método `equals()` está presente em todas as classes, pois é herdado da classe `Object`, porém seu comportamento deve ser reescrito caso seja necessário comparar igualdade entre as instâncias de uma classe. A implementação herdada compara apenas endereço de memória.

O método `equals()` retorna `true` se os argumentos forem iguais entre si, caso contrário, retorna `false`. A sua sintaxe é a seguinte:

```
public boolean equals(Object a)
```

Este método apresenta como parâmetro o objeto `a`, que representa o objeto com o qual o objeto que reescreve o método será comparado.

O exemplo a seguir demonstra a utilização deste método para testar a equivalência de objetos, considerando apenas o comportamento herdado da classe `Object`:



```
Pessoa.java
1
2 public class Pessoa {
3     String nome;
4     int idade;
5
6     public Pessoa(String n, int i){
7         nome = n;
8         idade = i;
9     }
10 }
```

```
J ExemploEquals.java X
1
2 public class ExemploEquals {
3     public static void main(String args[]){
4         Pessoa a = new Pessoa("Rodrigo", 23);
5         Pessoa b = new Pessoa("Claudio", 20);
6
7         System.out.println(a.nome + " é igual a " + b.nome + "? " + a.equals(b));
8
9         a.nome = b.nome; // Nomes iguais, mas objetos diferentes
10
11        System.out.println(a.nome + " é igual a " + b.nome + "? " + a.equals(b));
12
13        a = b; // Objetos iguais
14
15        System.out.println(a.nome + " é igual a " + b.nome + "? " + a.equals(b));
16    }
17 }
```

O resultado é mostrado a seguir:

```
Console X Problems @ Javadoc E
<terminated> ExemploEquals [Java Application]
Rodrigo é igual a Cláudio? false
Cláudio é igual a Cláudio? false
Cláudio é igual a Cláudio? true
```

15.6.2.1. As regras de equals()

Para desenvolver aplicações em Java, você deve seguir um conjunto de regras, que estão especificadas no contrato da linguagem Java. No contrato de **equals()**, está definido que este método é:

- **Consistente:** Considere as variáveis **a** e **b**. Caso não sejam realizadas alterações nas informações utilizadas na comparação do objeto de **equals()**, são retornadas múltiplas chamadas de **a.equals(b)** consistentemente **false** ou **true** para qualquer valor de referência aplicado às variáveis;
- **Reflexivo:** Considere a variável **a**. O valor **true** é retornado por **a.equals()** para qualquer valor de referência atribuído à variável **a**;
- **Simétrico:** Considere as variáveis **a** e **b**. Somente se **b.equals(a)** retornar o valor **true**, **a.equals(b)** retornará **true** para qualquer valor de referência atribuído a ambas as variáveis;
- **Transitivo:** Considere as variáveis **a**, **b** e **c**. Se **a.equals(b)** e **b.equals(c)** retornar **true**, então, **a.equals(c)** também retornará **true** para qualquer valor de referência atribuído às variáveis.

O método **equals()** recebe também como argumento o valor **null**. Considerando a variável **a**, por exemplo, a operação **a.equals(null)** retorna **false** para qualquer valor de **a** que seja diferente de **null**.

Você deve saber que o método **hashCode()** e o método **equals()** estão vinculados por um contrato de associação, no qual fica determinado que os valores de **hashing** de dois objetos devem ser iguais, caso o método **equals()** considere esses objetos equivalentes. Por isso, para substituir o método **equals()**, é preciso substituir também o método **hashCode()**.

15.6.3. Hashing

Ao armazenar um tipo de elemento em um endereço de alguns tipos de estrutura de dados, você realiza um processo chamado **hashing**. Esse endereço da estrutura de dados é gerado a partir de um algoritmo que toma por base o valor a ser armazenado.

Quando já existe um elemento em uma determinada posição da tabela de **hashing** e você tenta inserir outro elemento nesta posição, acontece uma **colisão**, ou seja, a mesma posição na tabela foi originada para elementos diferentes, gerando, a partir de uma colisão, uma lista encadeada de elementos, formando o conceito de agrupamento baseado no código hash.

Para melhor distribuição dos elementos nos agrupamentos formados (quanto mais equilibrado, melhor é o desempenho), os métodos **hashing** são criados conforme o tamanho e o tipo de estrutura de dados utilizados. Algumas classes de conjuntos utilizam o valor de **hashing** de um objeto. Esse valor não é exclusivo, contudo, ele pode ser considerado como um número por meio do qual o objeto é agrupado na coleção.

Nos conjuntos **HashSet** e **HashMap**, o local em que um objeto é armazenado é determinado pelo seu código de **hashing**.

Você deve saber que uma classe pode conter um método de código de **hashing** ineficiente (com distribuição ruim dos objetos nos grupos), apesar de retornar um código **hash** válido de acordo com o contrato entre **equals** e **hashCode**.

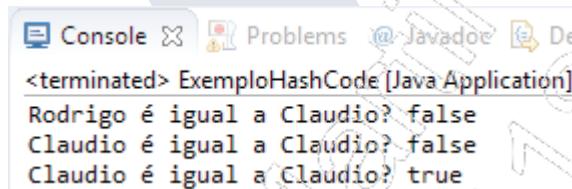
O método **hashCode()**, cujo parâmetro é o objeto **o**, retorna o código de **hashing** para um argumento que não seja nulo e **0** para um argumento nulo. Sua sintaxe é a seguinte:

```
public int hashCode()
```

O exemplo a seguir demonstra a utilização deste método:

```
ExemploHashCode.java
1  public class ExemploHashCode {
2      public static void main(String args[]){
3          Pessoa a = new Pessoa("Rodrigo", 23);
4          Pessoa b = new Pessoa("Claudio", 20);
5
6          System.out.println(a.nome + " é igual a " + b.nome + "? " + (a.hashCode() == b.hashCode()));
7
8          a.nome = b.nome; // Nomes iguais, mas objetos diferentes
9
10         System.out.println(a.nome + " é igual a " + b.nome + "? " + (a.hashCode() == b.hashCode()));
11
12         a = b; // Objetos iguais
13
14         System.out.println(a.nome + " é igual a " + b.nome + "? " + (a.hashCode() == b.hashCode()));
15     }
16 }
17 }
```

O resultado é mostrado a seguir:



```
Console X Problems @ Javadoc De
<terminated> ExemploHashCode [Java Application]
Rodrigo é igual a Claudio? false
Claudio é igual a Claudio? false
Claudio é igual a Claudio? true
```

15.6.3.1. As regras de hashCode()

Assim como o método **equals(Object)**, o método **hashCode()** possui um contrato com regras definidas. Veja o que está definido no contrato de **hashCode()**:

- Quando o método **equals(Object)** reconhece dois objetos como iguais, o resultado referente à chamada do método **hashCode()** nos dois objetos é o mesmo inteiro;
- É possível chamar o método **hashCode()** mais de uma vez no mesmo objeto durante a execução de um aplicativo Java. Neste caso, o método retorna o mesmo inteiro, o qual deve ser constante em todas as execuções do aplicativo, desde que não tenham sido realizadas alterações nas informações utilizadas na comparação do método **equals()** do mesmo objeto;
- Quando o método **equals(Object)** considerar que dois objetos são distintos, a chamada ao método **hashCode()** nesses objetos não precisa necessariamente produzir resultados inteiros distintos;
- Uma sobrescrita (**overriding**) válida do método **hashCode()** pode ser feita quando se deseja alterar a estratégia de distribuição dos objetos na tabela de **hashing**.

15.6.4. Método forEach()

Outra forma de realizar a iteração entre os itens de uma coleção é através do método **forEach()**.

O método **forEach()** permite a utilização de uma expressão lambda que interage com cada um dos itens da coleção, sem que seja necessário criar um loop no código.

O código a seguir produzirá o mesmo resultado que o código do exemplo anterior:

```
J ExemploForEach.java X
1 import java.util.TreeSet;
2
3 public class ExemploForEach {
4
5     public static void main(String[] args) {
6
7         TreeSet<String> cursos = new TreeSet<>();
8
9         cursos.add("PHP");
10        cursos.add("Linux");
11        cursos.add("Asp");
12        cursos.add("Java");
13        cursos.add("Delphi");
14
15        System.out.println("Cursos em ordem alfabética: \n");
16
17        cursos.forEach(c -> System.out.println("Curso: " + c));
18    }
19
20 }
```

! O método **forEach()** aplicado a coleções (sets e lists) deve sempre implementar um **Consumer** (uma das interfaces funcionais do pacote `java.util.function`). Em outras palavras, o **forEach()** precisa de uma expressão lambda que recebe um objeto (do mesmo tipo genérico usado pela coleção) e não retorna nada (**void**).

15.6.5. Método removeIf()

Outro método utilitário que faz uso de expressões lambda é o método **removeIf()**.

Através deste método, podemos remover de nossa coleção todos os itens que não satisfaçam a um determinado critério.

O exemplo a seguir remove todos os produtos cujo preço seja superior a R\$ 2,50:

```
ExemploRemoveIf.java
1 import java.util.HashSet;
2 import java.util.Set;
3
4 public class ExemploRemoveIf {
5
6     public static void main(String[] args) {
7
8         Set<Produto> cursos = new HashSet<>();
9
10        cursos.add(new Produto("Leite em pó", "Laticínios", 5.80));
11        cursos.add(new Produto("Cerveja em lata", "Bebidas", 1.89));
12        cursos.add(new Produto("Detergente", "Limpeza", 2.49));
13        cursos.add(new Produto("Manteiga", "Laticínios", 2.90));
14        cursos.add(new Produto("Pasta de dente", "Limpeza", 1.75));
15
16        cursos.removeIf(p -> p.getPreco() > 2.5);
17
18        System.out.println("Lista de produtos:\n");
19        cursos.forEach(p -> System.out.println("Produto: " + p));
20    }
21 }
```

O método **removeIf()** deve sempre implementar um **Predicate** (outra interface funcional do pacote `java.util.function`). Em outras palavras, o **removeIf()** precisa de uma expressão lambda que recebe um objeto (do mesmo tipo genérico usado pela coleção) e retorna um **boolean** que indica se o objeto da vez deverá ou não ser removido da lista.

15.6.6. Interface Comparable

O método de comparação `int compareTo(Object o)` é definido pela interface **Comparable**. O resultado deste método pode ser um número negativo, positivo ou zero.

Veja:

- Número negativo indica que o objeto é menor do que o objeto passado como parâmetro;
- Número positivo indica que o objeto é maior do que o objeto passado como parâmetro;
- Zero indica que o objeto e o objeto passado como parâmetro são iguais.

A classe **Collections** ordena as implementações e os arrays da interface **List** que possuem objetos que implementam a interface **Comparable**. Essa ordenação é realizada de forma automática a partir do método `sort()`.

As chaves de **SortedSet** ou de **SortedMap** podem ser os objetos que implementam a interface **Comparable**. Para tanto, não há a necessidade de que um determinado objeto da classe **Comparator** seja passado.

Para o uso de **Comparable**, todos os elementos da coleção devem implementar a interface. Lembre-se que, nesse caso, somente uma forma de ordenação torna-se possível. Caso haja a necessidade de mais de uma forma de ordenação, use a interface **Comparator**.

15.6.7. Interface Comparator

O método de comparação `int compare(Object o1, Object o2)` é definido por esta interface. Como na interface **Comparable**, o método do **Comparator** resulta no mesmo número positivo, negativo ou zero.

O uso da interface **Comparator** possibilita a realização de múltiplas formas de ordenação, pois é possível criar várias classes diferentes implementando **Comparator**, cada qual com um algoritmo de ordenação diferente.

Diferentemente da interface **Comparable**, os elementos da coleção não precisam implementar a interface, bastando passar uma classe que a implemente como referência para o método `Collections.sort(Comparator)`, e a ordenação será feita com base na classe. Para alterar o método de ordenação, basta criar nova classe que implemente **Comparator** e passá-la como argumento a `Collections.sort(Comparator)` novamente para se obter nova ordenação.

15.7. Manipulando coleções com Streams

Com o surgimento das expressões lambda e o novo paradigma de programação funcional, novos métodos de manipulação de coleções foram implementados a partir do Java 8.

Além dos métodos **forEach()** e **removeIf()** já vistos neste capítulo, a API de coleções traz muitos outros métodos utilitários que podem ser aplicados a coleções na realização de filtragem, ordenação e muitas outras tarefas.

A fim de garantir a compatibilidade da linguagem com versões anteriores, as interfaces já existentes nesta API foram preservadas, e os novos métodos utilitários foram agrupados na nova interface **java.util.stream.Stream**.

Um **Stream** trata-se de um manipulador dos dados de uma coleção. Ele está sempre associado a uma coleção e acoplamos a ele instruções de busca, filtragem ou quaisquer outros tipos de processamento de coleções.

Um stream possui um genérico do mesmo tipo da coleção associada. Para obter um stream a partir de uma coleção, basta executar o método **stream()**:

```
Stream<tipo> meuStream = colecao.stream();
```

A partir de um stream, podemos executar instruções de processamento sobre a coleção a fim de obter outro stream com as instruções já processadas:

```
Stream<tipo> outroStream = meuStream.algumMetodo(lambda);
```

Em geral, as operações com streams são realizadas em cascata, a fim de produzir um resultado final após sucessivos processamentos:

```
colecao.stream()
    .algumMetodo(lambda1)
    .outroMetodo(lambda2)
    .maisUmMetodo(lambda3);
```

É importante ressaltar que a manipulação de streams não afeta a coleção associada. A execução de métodos sucessivos sobre um stream apenas afeta a forma com que os dados são buscados sobre a coleção.

Veja o exemplo a seguir:

```
ExemploStream.java
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class ExemploStream {
5
6     public static void main(String[] args) {
7
8         /* Cria uma lista vazia de funcionários. */
9         List<Funcionario> list = new ArrayList<>();
10
11         /* Adiciona alguns funcionários a lista. */
12         list.add(new Funcionario(2005, "Manuel da Silva", "Desenvolvedor", 3580.0));
13         list.add(new Funcionario(3018, "Joaquim Batista", "Desenvolvedor", 5500.0));
14         list.add(new Funcionario(1045, "Maria das Dores", "Analista", 6250.0));
15         list.add(new Funcionario(1780, "João Ricardo", "Desenvolvedor", 7100.0));
16         list.add(new Funcionario(5200, "Ana-Maria", "Vendedora", 4100.0));
17         list.add(new Funcionario(3420, "André de Souza", "Desenvolvedor", 5890.0));
18         list.add(new Funcionario(3999, "Robson Gusmão", "Analista", 6500.0));
19         list.add(new Funcionario(1920, "Rubens Vieira", "Coordenador", 12300.5));
20         list.add(new Funcionario(2389, "Eduardo Alves", "Desenvolvedor", 3200.0));
21         list.add(new Funcionario(6300, "Joana Paiva", "Atendente", 1350.0));
22
23         /* Executa sucessivas operações com os dados da lista. */
24         list.stream()
25             .filter(f -> "Desenvolvedor".equals(f.getCargo()))
26             .sorted((f1, f2) -> f1.getNome().compareTo(f2.getNome()))
27             .skip(1)
28             .limit(2)
29             .forEach(System.out::println);
30     }
31 }
```

A seguir, veremos os métodos mais comuns pertencentes à interface Stream.

15.7.1. Método sorted()

Obtém um stream ordenado dos itens da coleção:

```
J ExemploSorted.java X
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class ExemploSorted {
5
6     public static void main(String[] args) {
7
8         List<Funcionario> lista = new ArrayList<>();
9         lista.add(new Funcionario(3018, "Joaquim Batista", "Desenvolvedor", 5500.0));
10        lista.add(new Funcionario(1045, "Maria das Dores", "Analista", 6250.0));
11        lista.add(new Funcionario(1780, "João Ricardo", "Desenvolvedor", 7100.0));
12        lista.add(new Funcionario(5200, "Ana Maria", "Vendedor", 4100.0));
13        lista.add(new Funcionario(3999, "Robson Gusmão", "Analista", 6500.0));
14        lista.add(new Funcionario(2389, "Eduardo Alves", "Desenvolvedor", 3200.0));
15
16        lista.stream()
17            .sorted()
18            .forEach(System.out::println);
19    }
20 }
```

O resultado será o seguinte:

```
Console X Problems @ Jayadoc Declaration Task List
<terminated> ExemploSorted [Java Application] C:\Java\jdk1.8.0_05\bin\java
1045 Maria das Dores           Analista      6.250,00
1780 João Ricardo              Desenvolvedor 7.100,00
2389 Eduardo Alves             Desenvolvedor 3.200,00
3018 Joaquim Batista           Desenvolvedor 5.500,00
3999 Robson Gusmão            Analista      6.500,00
5200 Ana Maria                 Vendedor     4.100,00
```

Este método funciona corretamente só quando os itens da coleção implementam a interface **java.lang.Comparable**, que obriga a existência de uma “ordenação natural” definida em cada item da coleção.

No exemplo anterior, a classe **Funcionario** possui um critério natural de ordenação por matrícula.

Caso desejemos uma ordenação diferente da natural ou mesmo se os elementos não possuírem um critério natural, podemos chamar este método passando uma expressão lambda de ordenação:

```
J ExemploSortedByName.java X
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class ExemploSortedByName {
5
6     public static void main(String[] args) {
7
8         List<Funcionario> lista = new ArrayList<>();
9         lista.add(new Funcionario(3018, "Joaquim Batista", "Desenvolvedor", 5500.0));
10        lista.add(new Funcionario(1045, "Maria das Dores", "Analista", 6250.0));
11        lista.add(new Funcionario(1780, "João Ricardo", "Desenvolvedor", 7100.0));
12        lista.add(new Funcionario(5200, "Ana Maria", "Vendedor", 4100.0));
13        lista.add(new Funcionario(3999, "Robson Gusmão", "Analista", 6500.0));
14        lista.add(new Funcionario(2389, "Eduardo Alves", "Desenvolvedor", 3200.0));
15
16        lista.stream()
17            .sorted((f1, f2) -> f1.getNome().compareTo(f2.getNome()))
18            .forEach(System.out::println);
19    }
20 }
```

Veja o resultado:

Id	Nome	Função	Salário
5200	Ana Maria	Vendedor	4.100,00
2389	Eduardo Alves	Desenvolvedor	3.200,00
3018	Joaquim Batista	Desenvolvedor	5.500,00
1780	João Ricardo	Desenvolvedor	7.100,00
1045	Maria das Dores	Analista	6.250,00
3999	Robson Gusmão	Analista	6.500,00

Neste exemplo, na linha 17, criamos uma expressão lambda que implementa o critério de comparação entre dois funcionários (representados por **f1** e **f2**) e realizamos a comparação pelo nome (método **getNome()**).

Outra forma de especificar o critério de ordenação é através da interface **Comparator**, método **comparing()**, na qual podemos simplesmente passar como argumento o atributo de comparação (método **get**).

A linha 17 do código anterior pode ser substituída por:

```
.sorted(Comparator.comparing(Funcionario::getNome))
```

A vantagem do método **Comparator.comparing()** é que ele também pode ser utilizado em cascata:

```
J ExemploSortedEmCascata.java X
1+ import java.util.ArrayList;...
4
5 public class ExemploSortedEmCascata {
6
7@   public static void main(String[] args) {
8
9      List<Funcionario> lista = new ArrayList<>();
10     lista.add(new Funcionario(3018, "Joaquim Batista", "Desenvolvedor", 5500.0));
11     lista.add(new Funcionario(1045, "Maria das Dores", "Analista", 6250.0));
12     lista.add(new Funcionario(1780, "João Ricardo", "Desenvolvedor", 7100.0));
13     lista.add(new Funcionario(5200, "Ana Maria", "Vendedor", 4100.0));
14     lista.add(new Funcionario(3999, "Robson Gusmão", "Analista", 6500.0));
15     lista.add(new Funcionario(2389, "Eduardo Alves", "Desenvolvedor", 3200.0));
16
17     lista.stream()
18       .sorted(Comparator.comparing(Funcionario::getCargo)
19                  .reversed()
20                  .thenComparing(Funcionario::getSalario))
21       .forEach(System.out::println);
22   }
23 }
```

Confira o resultado:

Funcionario ID	Nome	Cargo	Salário
5200	Ana Maria	Vendedor	4.100,00
2389	Eduardo Alves	Desenvolvedor	3.200,00
3018	Joaquim Batista	Desenvolvedor	5.500,00
1780	João Ricardo	Desenvolvedor	7.100,00
1045	Maria das Dores	Analista	6.250,00
3999	Robson Gusmão	Analista	6.500,00

Neste exemplo, estamos ordenando inversamente por cargo e, em seguida, diretamente por salário.

15.7.2. Método filter()

Obtém um stream contendo alguns dos elementos do stream original.

O método **filter()** permite a utilização de uma expressão lambda de filtragem, que desconsidera os itens que não atendam ao critério especificado.

O exemplo adiante filtra uma coleção de funcionários, exibindo somente aqueles que são desenvolvedores e que possuem salário superior a R\$ 5.000,00:

```
J ExemploFilter.java X
1④ import java.util.ArrayList;⑤
3
4 public class ExemploFilter {
5
6⑥     public static void main(String[] args) {
7
8         List<Funcionario> lista = new ArrayList<>();
9
10        lista.add(new Funcionario(3018, "Joaquim Batista", "Desenvolvedor", 5550.0));
11        lista.add(new Funcionario(1045, "Maria da Dores", "Analista", 6250.0));
12        lista.add(new Funcionario(1780, "João Ricardo", "Desenvolvedor", 7100.0));
13        lista.add(new Funcionario(5200, "Ana Maria", "Vendedor", 4100.0));
14        lista.add(new Funcionario(3999, "Robson Gusmão", "Analista", 6500.0));
15        lista.add(new Funcionario(2389, "Eduardo Alves", "Desenvolvedor", 3200.0));
16
17        lista.stream()
18            .filter(f -> f.getCargo().equals("Desenvolvedor"))
19            .filter(f -> f.getSalario() > 5000)
20            .forEach(System.out::println);
21    }
22 }
```

O resultado será o seguinte:



The screenshot shows an IDE interface with the following details:

- Console:** Shows the output of the Java application.
- Output:**

```
<terminated> ExemploFilter [Java Application] C:\Java\jdk1.8.0_05\bin\javav
3018 Joaquim Batista      Desenvolvedor      5.550,00
1780 João Ricardo        Desenvolvedor      7.100,00
```

15.7.3. Método limit()

Limita a quantidade de itens contidos no stream original gerando um stream com, no máximo, a quantidade especificada.

O exemplo a seguir exibe os três funcionários mais bem remunerados:

```
J ExemploLimit.java X
1 import java.util.ArrayList;
2 import java.util.Comparator;
3 import java.util.List;
4
5 public class ExemploLimit {
6
7     public static void main(String[] args) {
8
9         List<Funcionario> lista = new ArrayList<>();
10
11        lista.add(new Funcionario(3018, "Joaquim Batista", "Desenvolvedor", 5550.0));
12        lista.add(new Funcionario(1045, "Maria da Dores", "Analista", 6250.0));
13        lista.add(new Funcionario(1780, "João Ricardo", "Desenvolvedor", 7100.0));
14        lista.add(new Funcionario(5200, "Ana Maria", "Vendedor", 4100.0));
15        lista.add(new Funcionario(3999, "Robson Gusmão", "Analista", 6500.0));
16        lista.add(new Funcionario(2389, "Eduardo Alves", "Desenvolvedor", 3200.0));
17
18        lista.stream()
19            .sorted(Comparator.comparing(Funcionario::getSalario)
20                  .reversed())
21            .limit(3)
22            .forEach(System.out::println);
23    }
24 }
```

Confira o resultado:

```
Console X Problems Javadoc Declaration Task List
<terminated> ExemploLimit [Java Application] C:\Java\jdk1.8.0_05\bin\java
1780 João Ricardo      Desenvolvedor    7.100,00
3999 Robson Gusmão   Analista       6.500,00
1045 Maria da Dores   Analista       6.250,00
```

15.7.4. Método skip()

Este método desconsidera os primeiros n itens do stream original. Tem o efeito oposto ao do método `limit()`.

O exemplo adiante exibe todos os funcionários ordenados inversamente por salário, com exceção dos três primeiros:

```
J ExemploSkip.java X
1 import java.util.ArrayList;
2 import java.util.Comparator;
3 import java.util.List;
4
5 public class ExemploSkip {
6
7     public static void main(String[] args) {
8
9         List<Funcionario> lista = new ArrayList<>();
10
11        lista.add(new Funcionario(3018, "Joaquim Batista", "Desenvolvedor", 5550.0));
12        lista.add(new Funcionario(1045, "Maria da Dóres", "Analista", 6250.0));
13        lista.add(new Funcionario(1780, "João Ricardo", "Desenvolvedor", 7100.0));
14        lista.add(new Funcionario(5200, "Ana Maria", "Vendedor", 4100.0));
15        lista.add(new Funcionario(3999, "Robson Gusmão", "Analista", 6500.0));
16        lista.add(new Funcionario(2389, "Eduardo Alves", "Desenvolvedor", 3200.0));
17
18        lista.stream()
19            .sorted(Comparator.comparing(Funcionario::getSalario)
20                    .reversed())
21            .skip(3)
22            .forEach(System.out::println);
23    }
24 }
```

Veja o resultado:



```
Console X Problems @ Javadoc Declaration Task List
<terminated> ExemploSkip [Java Application] C:\Java\jdk1.8.0_05\bin\java
3018 Joaquim Batista      Desenvolvedor      5.550,00
5200 Ana Maria           Vendedor          4.100,00
2389 Eduardo Alves       Desenvolvedor      3.200,00
```

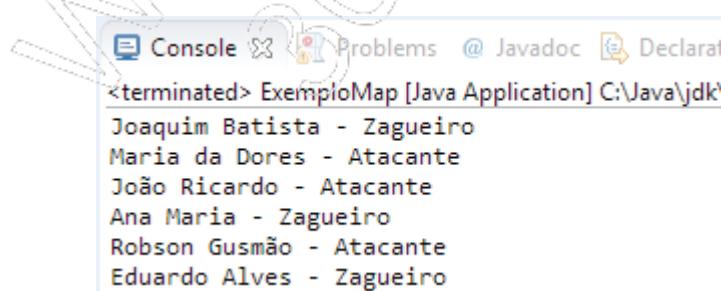
15.7.5. Método map()

Utiliza uma expressão lambda de transformação para gerar um stream de elementos de um tipo diferente.

No exemplo adiante, suponha que a classe **JogadorFutebol** possui os atributos **nome** e **posicao**, bem como um construtor que permita passar estas informações no momento da instanciação. Podemos, então, criar um stream de **JogadorFutebol** a partir de um stream de **Funcionario**:

```
J ExemploMap.java X
1 import java.util.ArrayList;
2 import java.util.List;
3 import java.util.stream.Stream;
4
5 public class ExemploMap {
6
7     public static void main(String[] args) {
8
9         List<Funcionario> lista = new ArrayList<>();
10
11         lista.add(new Funcionario(3018, "Joaquim Batista", "Desenvolvedor", 5550.0));
12         lista.add(new Funcionario(1045, "Maria da Dores", "Analista", 6250.0));
13         lista.add(new Funcionario(1780, "João Ricardo", "Desenvolvedor", 7100.0));
14         lista.add(new Funcionario(5200, "Ana Maria", "Vendedor", 4100.0));
15         lista.add(new Funcionario(3999, "Robson Gusmão", "Analista", 6500.0));
16         lista.add(new Funcionario(2389, "Eduardo Alves", "Desenvolvedor", 3200.0));
17
18         Stream<Funcionario> streamFunc = lista.stream();
19         Stream<JogadorFutebol> streamJogador =
20             streamFunc.map(f -> new JogadorFutebol(
21                 f.getNome(),
22                 f.getSalario() > 6000 ? "Atacante" : "Zagueiro"
23             ));
24
25         streamJogador.forEach(System.out::println);
26     }
27 }
```

O resultado será o seguinte:



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the following text:
<terminated> ExemploMap [Java Application] C:\Java\jdk
Joaquim Batista - Zagueiro
Maria da Dores - Atacante
João Ricardo - Atacante
Ana Maria - Zagueiro
Robson Gusmão - Atacante
Eduardo Alves - Zagueiro

Veja um outro exemplo, com a mesma lista de funcionários. Mas, desta vez, utilizamos o stream de **Funcionario** para obter um stream de **String**, contendo simplesmente os seus cargos:

```
J ExemploMap2.java ✘
1④ import java.util.ArrayList;⑤
4
5 public class ExemploMap2 {
6
7④     public static void main(String[] args) {
8
9         List<Funcionario> lista = new ArrayList<>();
10
11         lista.add(new Funcionario(3018, "Joaquim Batista", "Desenvolvedor", 5550.0));
12         lista.add(new Funcionario(1045, "Maria da Dores", "Analista", 6250.0));
13         lista.add(new Funcionario(1780, "João Ricardo", "Desenvolvedor", 7100.0));
14         lista.add(new Funcionario(5200, "Ana Maria", "Vendedor", 4100.0));
15         lista.add(new Funcionario(3999, "Robson Gusmão", "Analista", 6500.0));
16         lista.add(new Funcionario(2389, "Eduardo Alves", "Desenvolvedor", 3200.0));
17
18         Stream<Funcionario> streamFunc = lista.stream();
19         Stream<String> streamCargo = streamFunc.map(Funcionario::getCargo);
20
21         streamCargo.forEach(System.out::println);
22     }
23 }
```

O resultado será o seguinte:

```
Console ✘ Problems @
<terminated> ExemploMap2 [Java A]
Desenvolvedor
Analista
Desenvolvedor
Vendedor
Analista
Desenvolvedor
```

15.7.6. Método `distinct()`

Este método desconsidera os elementos repetidos do stream original. Em outras palavras, os itens iguais são considerados apenas uma vez. O critério de igualdade é estabelecido pelo método `equals()` de cada objeto da coleção.

O exemplo adiante exibe os diferentes cargos (sem repetições) de todos os funcionários de uma lista:

```
J ExemploDistinct.java X
1+ import java.util.ArrayList;
2
3 public class ExemploDistinct {
4
5     public static void main(String[] args) {
6
7         List<Funcionario> lista = new ArrayList<>();
8
9
10        lista.add(new Funcionario(3018, "Joaquim Batista", "Desenvolvedor", 5550.0));
11        lista.add(new Funcionario(1045, "Maria da Dores", "Analista", 6250.0));
12        lista.add(new Funcionario(1780, "João Ricardo", "Desenvolvedor", 7100.0));
13        lista.add(new Funcionario(5200, "Ana Maria", "Vendedor", 4100.0));
14        lista.add(new Funcionario(3999, "Robson Gusmão", "Analista", 6500.0));
15        lista.add(new Funcionario(2389, "Eduardo Alves", "Desenvolvedor", 3200.0));
16
17        lista.stream()
18            .map(Funcionario::getCargo)
19            .distinct()
20            .forEach(System.out::println);
21    }
22 }
```

15.7.7. Método count()

Este método retorna um número inteiro longo (**long**), informando a quantidade de itens que está sendo considerada no stream.

O exemplo adiante exibe a **quantidade** de funcionários que recebem salário maior que R\$ 5.000,00:

```
J ExemploCount.java X
1④ import java.util.ArrayList;⑤
2
3
4 public class ExemploCount {
5
6⑥ public static void main(String[] args) {
7
8     List<Funcionario> lista = new ArrayList<>();
9
10    lista.add(new Funcionario(3018, "Joaquim Batista", "Desenvolvedor", 5550.0));
11    lista.add(new Funcionario(1045, "Maria da Dores", "Analista", 6250.0));
12    lista.add(new Funcionario(1780, "João Ricardo", "Desenvolvedor", 7100.0));
13    lista.add(new Funcionario(5200, "Ana Maria", "Vendedor", 4100.0));
14    lista.add(new Funcionario(3999, "Robson Gusmão", "Analista", 6500.0));
15    lista.add(new Funcionario(2389, "Eduardo Alves", "Desenvolvedor", 3200.0));
16
17    System.out.println(lista.stream()
18                      .filter(f -> f.getSalario() > 5000)
19                      .count());
20
21 }
```

Veja o resultado:

```
Console X Problems @
<terminated> ExemploCount [Java Aplication]
4
```

15.7.8. Métodos min() e max()

Os métodos **min()** e **max()** retornam, respectivamente, uma referência ao menor e ao maior elemento do stream, e o critério de comparação é especificado por uma expressão lambda.

Veja um exemplo:

```
J ExemploMinMax.java X
1 import java.util.ArrayList;
2 import java.util.Comparator;
3 import java.util.List;
4
5 public class ExemploMinMax {
6
7     public static void main(String[] args) {
8
9         List<Funcionario> lista = new ArrayList<>();
10
11        lista.add(new Funcionario(3018, "Joaquim Batista", "Desenvolvedor", 5550.0));
12        lista.add(new Funcionario(1045, "Maria da Dores", "Analista", 6250.0));
13        lista.add(new Funcionario(1780, "João Ricardo", "Desenvolvedor", 7100.0));
14        lista.add(new Funcionario(5200, "Ana Maria", "Vendedor", 4100.0));
15        lista.add(new Funcionario(3999, "Robson Gusmão", "Analista", 6500.0));
16        lista.add(new Funcionario(2389, "Eduardo Alves", "Desenvolvedor", 3200.0));
17
18        Funcionario minFunc =
19            lista.stream()
20                .min(Comparator.comparing(Funcionario::getMatricula))
21                .get();
22
23        Funcionario maxFunc =
24            lista.stream()
25                .max(Comparator.comparing(Funcionario::getMatricula))
26                .get();
27
28        System.out.println("Funcionario de MENOR matricula: \n" + minFunc);
29        System.out.println("Funcionario de MAIOR matricula: \n" + maxFunc);
30    }
31 }
```

O resultado será o seguinte:

```
Console X Probléms @ Javadoc Declaration Task List
<terminated> ExemploMinMax [Java Application] C:\Java\jdk\jdk1.8.0_05\bin\j
Funcionario de MENOR matricula:
 1045 Maria da Dores           Analista      6.250,00
Funcionario de MAIOR matricula:
 5200 Ana Maria               Vendedor     4.100,00
```

A fim de evitar exceções de **null pointer (NullPointerException)**, os métodos **min()** e **max()** retornam um **Optional**, que é uma entidade que pode conter um valor ou pode estar vazia, dependendo da operação que foi realizada.

Para obter o possível valor contido nesta entidade, utilize o método **get()**.

15.8. Interface Map

Esta interface define uma estrutura de dados que mapeia pares chave-valor. Um mapa não pode conter chaves duplicadas, pois cada chave deve mapear, no máximo, um valor.

Além disso, o conteúdo de um mapa pode ser visualizado como uma coleção de chaves, uma coleção de valores ou um conjunto de mapeamentos chave-valor. A ordem de um mapa é a ordem em que os iteradores retornam seus elementos na visualização de coleção do mapa.

Enquanto algumas implementações de mapas têm garantias específicas quanto à ordem de iteração em seus elementos, como a classe `TreeMap` (ordem natural), outras não apresentam garantias, como a classe `HashMap`.

15.8.1. Principais métodos

Três de suas implementações importantes são `HashMap`, `HashTable` e `TreeMap`.

Para adicionar um par chave-valor a uma implementação de `Map`, utilize o método adiante, em que `V` é o tipo do valor associado a `K`, que representa a chave:

```
V put(K chave, V valor)
```

Para verificar se um determinado valor ou uma chave foi inserida na classe na estrutura do `Map`, utilize uma das seguintes sintaxes:

- Para encontrar uma chave (`key`):

```
boolean containsKey(Object key)
```

Se a chave for encontrada, o valor `true` é retornado.

- Para encontrar um valor (`value`):

```
boolean containsValue(Object value)
```

Se o valor for encontrado, o valor `true` é retornado.

Você também pode realizar no **Map** as seguintes tarefas:

- **Recuperação de um valor do conjunto Map**

Para que seja possível retornar o valor associado à chave passada como parâmetro, utilize a sintaxe a seguir:

```
V get(Object key)
```

Para verificar se existe uma chave no **Map**, utilize o método **containsKey(Object Key)**. Esta chave pode estar associada a um valor **null**, o qual é retornado quando a chave não é encontrada.

- **Remoção de elementos do conjunto Map**

Para remover o par chave-valor localizado no **Map**, utilize a seguinte sintaxe:

```
V remove(Object key)
```

- **Verificação da quantidade ou do tamanho de elementos do Map**

Para retornar o número de pares chave-valor armazenados no conjunto **Map**, utilize a seguinte sintaxe:

```
int size()
```

Observe o exemplo a seguir:

```
J ExemploHashMap.java X
1 import java.util.HashMap;
2
3 public class ExemploHashMap {
4     public static void main(String[] args) {
5
6         /* Cria um mapa de produtos chaveado por strings. */
7         HashMap<String, Produto> map = new HashMap<>();
8
9         /* Adiciona alguns produtos ao mapa. */
10        map.put("leite",      new Produto("Leite em pó",      "Laticínios", 5.80));
11        map.put("cerveja",    new Produto("Cerveja em lata",  "Bebidas",   1.89));
12        map.put("detergente", new Produto("Detergente",       "Limpeza",   2.49));
13        map.put("manteiga",   new Produto("Manteiga",        "Laticínios", 2.90));
14        map.put("pasta",      new Produto("Pasta de dente",  "Limpeza",   1.75));
15
16        /* Verifica se o mapa possui algum item com chave "cerveja" */
17        System.out.println(map.containsKey("cerveja"));
18
19        /* Obtem o item com chave "pasta" */
20        System.out.println(map.get("pasta"));
21
22        /* Remove o item com chave "detergente" */
23        map.remove("detergente");
24
25        /* Mostra a quantidade de itens remanescentes no mapa */
26        System.out.println(map.size());
27
28        System.out.println("=====");
29
30        /* Exibe todos os itens do mapa. */
31        map.forEach((k, v) -> System.out.println(v));
32    }
33 }
```

O resultado é mostrado a seguir:

```
Console X Problems @ Javadoc E
<terminated> ExemploHashMap [Java Application]
true
Pasta de dente - Limpeza - 1.75
4
=====
Leite em pó - Laticínios - 5.8
Cerveja em lata - Bebidas - 1.89
Pasta de dente - Limpeza - 1.75
Manteiga - Laticínios - 2.9
```

! O método **forEach()** aplicado a mapas deve sempre implementar um **BiConsumer** (outra das interfaces funcionais do pacote `java.util.function`). Em outras palavras, o **forEach()** precisa de uma expressão lambda que recebe dois valores: **chave** e **conteúdo** de cada item contido no mapa, e não retorna nada (**void**).

15.9. Collections Framework

Todo conteúdo aprendido neste capítulo faz parte do framework de coleções do Java conhecido como **Collections Framework**.

Um framework de coleções é uma arquitetura unificada para representar e manipular coleções, permitindo que estas sejam utilizadas de forma independente dos detalhes da implementação.

As principais vantagens de um framework são:

- Reduzir o esforço de programação, fornecendo estruturas de dados e algoritmos para que não seja necessário escrevê-los;
- Aumentar a performance, implementando estruturas de dados e algoritmos de alto desempenho;
- Promover a reutilização de software, manipulando coleções e algoritmos por meio de uma interface padronizada.

O framework de coleções é baseado em interfaces de coleções, que representam diferentes tipos de coleções, como conjuntos, listas e mapas, e também possui implementações primárias dessas interfaces.

Além disso, o framework utiliza:

- **Algoritmos:** São métodos com funções úteis nas coleções, como ordenar uma lista;
- **Infraestrutura:** Composta de interfaces que fornecem suporte essencial para a interface de coleção;
- **Utilitários de array:** São funções utilitárias para arrays de tipos primitivos e objetos de referência. Esta característica foi adicionada à plataforma Java junto com o framework de coleções e se baseia em algumas estruturas do framework.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- A estrutura de coleções foi lançada na primeira versão da linguagem Java 2 (JDK1.2) e se expandiu nas versões 1.4 (Java 4) e Java 8. Ela contém mapas, conjuntos e listas que são utilizados na codificação para suprir as necessidades das aplicações Java. Para personalizar essa aplicação, o pacote **java.util** disponibiliza diversos utilitários e interfaces;
- Coleções do tipo **Set** consistem em conjuntos que não possuem elementos duplicados. Isto quer dizer que se dois objetos verificados pelo método **equals()** forem considerados equivalentes, somente um deles poderá permanecer no conjunto. O conjunto **Set** possui diversas implementações;
- Em coleções do tipo **List**, o índice é muito importante e, por isso, existem diversos métodos relacionados a ele. Para definir a posição do índice, é preciso configurar um objeto em um determinado índice. Outra possibilidade é adicionar esse objeto ao índice sem que a posição seja definida. Neste caso, o objeto é inserido no final do índice;
- A interface **java.util.stream.Stream** oferece um vasto conjunto de métodos que permite a manipulação dos dados de uma coleção, como filtragem e ordenação, sem que estes dados sejam alterados;
- Coleções do tipo **Map** substituíram a classe **Dictionary**, que era uma classe totalmente abstrata. Você deve saber que esta coleção mapeia pares chave-valor. Lembre-se que um mapa não pode conter chaves duplicadas, pois cada chave pode mapear, no máximo, um valor;
- A plataforma Java inclui um framework de coleções (**Collections Framework**), que é uma arquitetura unificada para representar e manipular coleções, permitindo que estas sejam manipuladas de forma independente de detalhes da implementação.

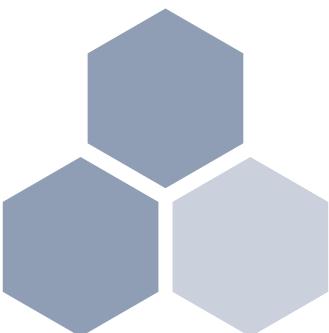


15

Coleções e mapas

Teste seus conhecimentos

Williams Martins 305.009.1845



Editora
IMPACTA



1. Qual das alternativas a seguir NÃO é uma operação de uma coleção?

- a) Adição de objetos.
- b) Verificação de existência de objeto ou grupo de objetos.
- c) Iteração pelo conjunto.
- d) Alteração de endereço de objetos.
- e) Recuperação de um objeto no conjunto.

2. Qual das alternativas a seguir NÃO é uma interface presente no Collections Framework?

- a) Stack
- b) Set
- c) Map
- d) List
- e) Collection

3. Sobre a coleção Set, qual das afirmativas está correta?

- a) Não possui elementos duplicados.
- b) Possui uma única implementação.
- c) A classe HashSet cria conjuntos ordenados.
- d) Representa uma coleção indexada.
- e) Todas as alternativas anteriores estão corretas.

4. Sobre a estrutura do tipo Map, qual das afirmativas está correta?

- a) Funciona em conjunto com a classe Dictionary.
- b) Mapeia pares chave-valor.
- c) Aceita chaves duplicadas.
- d) Não pode conter elementos nulos.
- e) Todas as alternativas anteriores estão corretas.

5. Sobre a coleção List, qual das afirmativas está correta?

- a) Três de suas implementações importantes são ArrayList, LinkedList e Vector.
- b) Existem diversos métodos relacionados ao índice do conjunto.
- c) Para adicionar um elemento na lista, podemos determinar a posição dele informando um índice de inserção.
- d) Permite elementos duplicados na lista.
- e) Todas as alternativas anteriores estão corretas.

6. Sobre o uso de streams, qual das afirmativas está correta?

- a) Os streams podem ser utilizados para filtrar, ordenar e realizar diversos outros tipos de operações sobre uma coleção.
- b) A partir de um stream, podemos utilizar o método map() para gerar dados de um formato diferente da coleção original.
- c) O método filter() desconsidera todos os elementos da coleção que não satisfaçam o critério especificado pela expressão lambda.
- d) O método skip() desconsidera os primeiros n elementos contidos na stream.
- e) Todas as alternativas anteriores estão corretas.



15

Coleções e mapas



Mãos à obra!

Willian Martins
305.009.118-45



Editora
IMPACTA

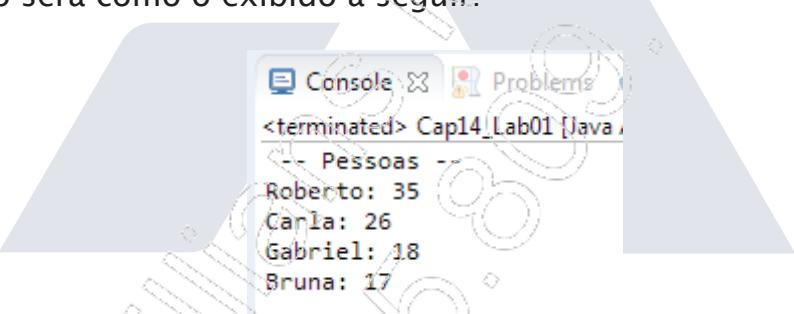


Laboratório 1

A – Utilizando um objeto **HashMap** parametrizado para guardar notas de alunos

1. Crie um projeto Java chamado **Cap15_Lab01**;
2. Dentro dele, crie uma classe com o nome **Cap15_Lab1** e insira a estrutura básica de um programa Java;
3. Importe a classe **HashMap** do pacote **java.util**;
4. Dentro do método **main**, crie um objeto **HashMap <String, Integer>** com o nome **pessoaMap**;
5. Utilizando o método **put**, insira quatro nomes de pessoas e suas respectivas idades no **HashMap pessoaMap**;
6. Utilizando o método **get**, imprima o nome e idade de todas as pessoas;
7. Compile e execute o programa;

O resultado será como o exibido a seguir:



```
Console Problems
<terminated> Cap14_Lab01 [Java]
-- Pessoas --
Roberto: 35
Carla: 26
Gabriel: 18
Bruna: 17
```

 O passo adiante é considerado um desafio!

8. Imprima o nome e a idade de todas as pessoas, utilizando uma expressão lambda.

Laboratório 2

A – Criando a estrutura do programa e um tipo construído denominado Estudante

1. Crie um projeto Java chamado **Cap15_Lab02**;
2. Dentro dele, crie uma classe com o nome **Cap15_Lab2** e insira a estrutura básica de um programa Java;
3. Crie uma segunda classe, chamada **Estudante**, com quatro atributos privados:
 - **nome**: String;
 - **notaMatematica**: double;
 - **notaPortugues**: double;
 - **media**: double.

4. Nessa classe, crie os métodos **getters** e **setters** apropriados e um construtor que receba somente os atributos **nome**, **notaMatematica** e **notaPortugues** como parâmetros e configure-os, atribuindo seus valores às suas variáveis de instância.

B – Utilizando uma coleção do tipo **ArrayList** parametrizada por um tipo **Estudante** para armazenar dados de diversos alunos

1. Dentro do método **main** da primeira classe criada, **Cap15_Lab2**, crie um novo **ArrayList** parametrizado para o tipo **Estudante**, da seguinte forma:

```
ArrayList<Estudante> estudanteList = new ArrayList<>();
```

2. Usando o método **add(E e)** do **ArrayList** criado, insira cinco instâncias da classe **Estudante** contendo os seguintes dados:

Nome	Nota de Matemática	Nota de Português
Joana	8,5	8,5
Antônio	6,0	9,0
Mariana	7,5	9,0
Ricardo	7,0	6,0
Gustavo	9,5	10,0

3. Utilize o método **forEach()** para aplicar uma expressão lambda que, para cada item, calcula a média das notas de matemática e português e assinala com o método **setMedia()**;

4. Use novamente o método **forEach()** para exibir o nome e média de cada aluno;

5. Compile e execute o programa.



15

Coleções e mapas



Projeto Prático – Fase 3

Willymson Martins
305.009.118-45



Editora
IMPACTA



equals, hashCode, ordenação, listas e exceções

Nesta fase, a classe de negócio **Filme** receberá os critérios de igualdade e ordenação através da reescrita dos seguintes métodos:

- **equals**;
- **hashCode**;
- **Filme** deve implementar a interface **Comparable** (método **compareTo**).

Atividades

1. Atualizar todos os métodos que possuem coleções para uma estrutura de dados apropriada (**List** com **ArrayList**):

- Os arrays nos parâmetros e retornos de métodos devem ser substituídos por coleções (**List** / **ArrayList**).

2. Levantar e tratar todas as exceções de negócio que podem acontecer nos métodos já criados, emitindo uma mensagem de erro ao usuário final se for o caso:

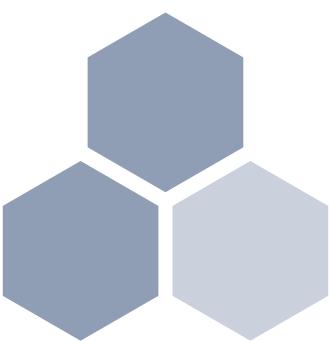
- Onde houver uma exceção em potencial, lançar a exceção e passar na pilha de execução;
- Repassar as exceções lançadas até chegar à camada de apresentação;
- Na camada de apresentação, tratar via **try/catch** e, no tratamento, emitir uma mensagem ao usuário final a respeito do problema.



16

Arquivos - I/O e NIO

- I/O;
- try-with-resources;
- Leitura de arquivos de texto;
- NIO - Arquivos e diretórios.



16.1.I/O

A linguagem Java trabalha com um fluxo controlável de entrada e saída de informações, chamado **stream**, que também é definido como fluxo **Input/Output** (entrada/saída), ou simplesmente fluxo **I/O**. Além disso, o stream é manipulado por objetos e classes da linguagem Java. Nos tópicos a seguir, serão apresentadas as classes que representam os streams de saída e entrada, assim como seus respectivos métodos. Você também conhecerá as classes usadas para a leitura de arquivos binários e de texto e o uso de paths.

16.1.1.Classe OutputStream

Um stream de saída tem a capacidade de enviar bytes para um coletor (**sink**). Esse stream é representado por diversas classes, que, por sua vez, possuem uma superclasse chamada **OutputStream**.

A **OutputStream** é uma classe abstrata com a seguinte representação:

The screenshot shows a Java Integrated Development Environment (IDE) with two open code editors. The top editor contains the abstract class `OutputStream.java`:

```
OutputStream.java
1 public abstract class OutputStream extends Object{  
2  
3 }  
4
```

The bottom editor contains the example class `ExemploCriacaoArquivo.java`:

```
ExemploCriacaoArquivo.java
1 // Vejamos o exemplo a seguir:  
2  
3 import java.io.*;  
4  
5 public class ExemploCriacaoArquivo {  
6  
7     static String texto = new String("JAVA");  
8  
9     public static void main(String args[]){  
10         try{  
11             FileOutputStream arquivo = new FileOutputStream("GerarArquivo.txt");  
12             DataOutputStream dados = new DataOutputStream(arquivo);  
13             dados.writeChars(texto);  
14         }catch(IOException e){  
15             System.out.print(e.getMessage());  
16         }  
17     }  
18 }  
19 }
```

Depois de compilado e executado o código anterior, um arquivo do tipo **.txt** será gerado com o nome **GerarArquivo**, como mostra a imagem a seguir:

Nome	Data de modificaç...	Tipo	Tamanho
.settings	03/01/2013 11:17	Pasta de arquivos	
bin	03/01/2013 11:17	Pasta de arquivos	
src	03/01/2013 11:17	Pasta de arquivos	
.classpath	03/01/2013 11:17	Arquivo CLASSPA...	1 KB
.project	03/01/2013 11:17	Arquivo PROJECT	1 KB
GerarArquivo.txt	03/01/2013 11:17	Documento de Te...	1 KB

A classe em questão trabalha com os seguintes métodos:

- **write(...)**: Grava bytes de saída para o destino. Possui algumas versões sobrecarregadas;
- **flush()**: Gera a saída de um byte de gravação pendente para o stream de destino. O byte de gravação pendente pode estar em um buffer interno;
- **close()**: Fecha o stream de saída. Em vista disso, os recursos de sistema que estavam sendo usados por esse stream são liberados para outros fins.

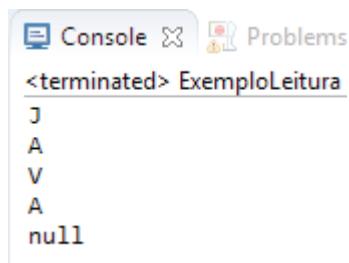
Uma condição para que as aplicações sejam capazes de definir uma subclasse de **OutputStream** é oferecer pelo menos um método de gravação de byte de saída.

Veja o exemplo a seguir:

```

ExemploLeitura.java
1 import java.io.*;
2
3 public class ExemploLeitura {
4     public static void main(String args[]){
5         try{
6             FileInputStream arquivo = new FileInputStream("GerarArquivo.txt");
7             DataInputStream dados = new DataInputStream(arquivo);
8             while(true){
9                 char c = dados.readChar();
10                System.out.println(c);
11            }
12        }catch(IOException e){
13            System.out.print(e.getMessage());
14        }
15    }
16 }
```

Depois de compilado e executado o código anterior, o resultado será como o exibido na imagem a seguir:



```
Console Problems
<terminated> ExemploLeitura
J
A
V
A
null
```

16.1.1.1. Métodos

Neste tópico, serão apresentados cada um dos métodos da classe **OutputStream**:

- **write (int b)**

Tendo como parâmetro o argumento **b** (o byte), o método **write (int b)**, cuja implementação se faz necessária nas subclasses de **OutputStream**, grava o byte especificado para o stream de saída.

- **write(byte[] b)**

Este método da classe **OutputStream** grava bytes de tamanho **b.length** para o stream de saída, a partir do array do byte especificado.

- **write(byte[] b, int off, int len)**

Este método possui os parâmetros **b** (dados), **off** (offset de início nos dados) e **len** (quantidade de bytes a ser gravada). Além disso, a partir do array de byte especificado, grava bytes **len**, com início no offset **off** do stream de saída.

- **flush()**

As funções principais deste método são:

- Esvaziar o stream de saída;
- Estimular a gravação de bytes de saída contidos em buffers.
- **close()**

O método **close()** fecha o stream de saída, como determina o contrato geral. Com o stream fechado, os recursos de sistema por ele utilizados são liberados para outros fins.

16.1.2. Classe InputStream

As classes que representam um stream de entrada de bytes têm como superclasse a **InputStream**, cuja sintaxe é a seguinte:

```
public abstract class InputStream  
extends Object
```

Se sua intenção é fazer com que uma aplicação defina uma subclasse de **InputStream**, é necessária a utilização de um método que retorne o próximo byte de entrada.

16.1.2.1. Métodos

Veja, a seguir, os métodos da classe **InputStream**:

- **read()**

O método **read()**, cuja implementação deve ser oferecida por uma subclasse, faz a leitura do próximo byte de dados que chega do stream de entrada, sendo que o byte de valor retornado apresentará um valor **int** entre 0 e 255.

Se não houver um único byte de dados em razão de o stream ter chegado ao seu final, o método **read()** retorna o valor **-1**.

O método em questão permanece bloqueado até o momento em que:

- Os dados de entrada estejam disponíveis;
- Uma exceção seja lançada;
- O fim do stream seja detectado.
- **read(byte[] b)**

A partir do stream de entrada, este método lê alguns números de bytes (inteiros) que são, em seguida, depositados no array **b** do buffer.

- **read(byte[] b, int off, int len)**

Este método basicamente lê bytes de **len**, do stream de entrada para um array de bytes.

- **skip(long n)**

O método **skip(n)** tem como função pular e ignorar **n** bytes de dados do stream de entrada. Ele retorna o número atual de bytes pulados.

- **available()**

Este método é responsável por retornar o número de bytes que podem ser lidos ou pulados a partir desse stream.

- **close()**

Tal qual no stream de saída, o método **close()** tem a finalidade de fechar o stream de entrada.

Com o stream fechado, os recursos de sistema que estavam sendo usados por ele são disponibilizados para outras tarefas.

- **mark(int readlimit)**

Este método estabelece a posição atual no stream de entrada. Esta posição definida será considerada por uma nova chamada ao método de reinicialização (**reset**), que ajustará o stream de entrada na mesma posição estabelecida anteriormente. O resultado será a releitura dos mesmos bytes.

- **reset()**

Cabe ao método **reset()**, anteriormente mencionado na descrição do método **mark(readlimit)**, posicionar o stream exatamente no mesmo instante em que o método **mark** foi chamado pela última vez.

- **markSupported()**

O método **markSupported()** verifica se o stream de entrada está apto a aceitar o uso dos métodos **mark** e **reset**.

16.1.3. Leitura de arquivos binários

Para fazer a leitura de arquivos binários, é recomendável o emprego da classe **FileInputStream**, que é comumente usada para ler streams de bytes não processados.

Em um sistema de arquivos, a classe **FileInputStream** obtém bytes de entrada a partir de um arquivo pertencente a este sistema. A seguir, veja a sintaxe de **FileInputStream**:

```
public class FileInputStream  
extends InputStream
```

A disponibilidade dos arquivos que fornecerão os bytes de entrada está ligada diretamente ao ambiente host em que se está trabalhando.

16.1.4. I/O - Arquivos e diretórios (classe File)

Na linguagem Java, a classe **File** é usada para indicar um arquivo ou diretório, ou seja, um local específico no sistema operacional. Você pode criá-los independentemente da existência dos diretórios ou arquivos especificados nos caminhos.

Observe o seguinte código:

```
File file = new File("/br/pasta/arquivo.txt");
```

No exemplo, um objeto da classe **java.io.File** é criado. Neste caso, o caminho especificado **/br/pasta/arquivo.txt** aponta para o arquivo **arquivo.txt**.

Agora, observe este outro código:

```
File DiretorioAtual = new File("nome_do_arquivo.txt");  
File DiretorioRaiz = new File("../arquivo.txt");
```

Você pode utilizar ponteiros para indicar locais relativos ao diretório inicial, no sistema. Este procedimento é possível, pois não há como modificar o diretório atual.

A primeira linha do código anterior aponta o arquivo **nome_do_arquivo.txt**, que estará localizado no diretório atual do sistema. Já a segunda linha do código aponta para **../arquivo.txt**, que estará localizado no diretório raiz do diretório atual.

A linguagem Java interpreta o sinal \ (barra invertida) como caractere de escape quando usado em Strings. Portanto, para indicar caminhos no sistema Windows, baseado no uso de barras invertidas, utilize \\ para efetivamente obter a impressão de uma barra invertida.

16.2. try-with-resources

O bloco de instrução **try-with-resources** pode ser considerado como um simples **try** que declara um ou mais recursos com o propósito de fechá-lo(s) automaticamente com o término do bloco.

Veja um exemplo de um trecho de código declarado com a técnica em questão:

```
public static void abrirArquivo() throws IOException{
    File file= new File("arquivo.txt");
    try (InputStream is = new FileInputStream(file)){
        //Código decorrente da manipulação de InputStream
    }
}
```

As classes que podem ser fechadas automaticamente neste processo devem implementar duas interfaces: **AutoCloseable** e **Closeable**.

Uma lista de classes que implementam estas interfaces pode ser verificada nos docs oficiais da linguagem Java (dentre elas, as apresentadas neste capítulo).

Com relação a essas interfaces, é importante salientar que, no processo de fechamento da classe (chamada ao método **close()** do recurso), pode ocorrer uma exceção: **IOException**, no caso de **Closeable**, e **Exception**, no caso das classes que implementem **AutoCloseable**.

16.2.1. Exceções suprimidas

A preocupação com as denominadas exceções suprimidas (**suppressed exceptions**) surge com o advento do bloco **try-with-resources** para os efeitos gerados em objetos que implementam a interface **AutoCloseable**.

As exceções suprimidas surgem quando:

- Uma exceção ocorre no bloco **try** e, dessa forma, o controle passa para um bloco **catch** devido a;
- Tendo em vista que, nesse momento, o bloco **try-with-resources** tenta fechar todos os recursos abertos sob sua responsabilidade, caso ocorra uma exceção nessa tentativa com os recursos **AutoCloseable**, exceções serão lançadas, porém, suprimidas.

Nessa situação, as exceções lançadas pela tentativa de fechamento dos recursos são suprimidas e podem ser acessadas no bloco **catch**, onde se encontra o fluxo por meio de uma construção como a apresentada adiante:

```
} catch (Exception e) {
    System.out.println(e.getMessage());
    for (Throwable t : e.getSuppressed()) {
        System.out.println(t.getMessage());
    }
}
```

16.3. Leitura de arquivos de texto

Veja, a seguir, duas classes utilizadas para a leitura de arquivos de texto, **FileReader** e **BufferedReader**.

16.3.1. Classe **FileReader**

O recurso mais acessível para ler arquivos de texto é a classe **FileReader**, representada da seguinte forma:

```
public class FileReader
extends InputStreamReader
```

A **FileReader** considera como apropriados o tamanho do buffer de bytes e a codificação de caracteres padrão.

É possível estabelecer os valores da codificação de caracteres padrão e do tamanho do buffer de bytes. Para isso, é necessário criar um **InputStreamReader** dentro de um **FileInputStream**.

Como recurso adicional à **FileReader**, você pode usar a classe **BufferedReader** para desempenhar a tarefa de ler cada uma das linhas do stream.

16.3.2. Classe BufferedReader

A classe **BufferedReader** proporciona uma leitura mais eficaz dos caracteres, já que possibilita ler não apenas estes caracteres, mas também as linhas e os arrays de caracteres. Sua sintaxe é a seguinte:

```
public class BufferedReader  
extends Reader
```

A leitura de caracteres por meio da **BufferedReader** é feita a partir de um stream de entrada de caracteres, sendo estes armazenados em um buffer, o qual possui um tamanho padrão satisfatório para atender a propósitos variados. Além disso, é possível ajustar o tamanho do buffer conforme as necessidades.

O armazenamento de uma entrada de arquivo em um buffer evita que os bytes do arquivo sejam lidos, transformados em caracteres e retornados de maneira ineficaz, toda vez que os métodos **read()** ou **readLine()** são chamados.

Veja o exemplo seguinte, no qual a entrada do arquivo especificado está sendo armazenada em buffer:

```
BufferedReader in = new BufferedReader(new FileReader("teste.in"));
```

16.4. NIO – Arquivos e diretórios

A seguir, uma visão geral da API NIO e suas principais classes: **Path**, **Paths** e **Files**.

16.4.1. Visão Geral de NIO

Na versão 1.4, foram introduzidas novas APIs de I/O (NIO), as quais fornecem novos recursos e melhor desempenho nas áreas de gerenciamento de arquivos, de buffer, rede escalável e arquivos I/O, suporte para conjunto de caracteres e correspondência de expressões regulares. Além disso, as APIs NIO complementam as especificações de I/O no pacote **java.io**.

As classes centrais das APIs NIO são:

- **Path**: Um arquivo ou diretório do sistema de arquivos;
- **Paths**: Classe utilitária responsável pela geração de um **Path** a partir do caminho especificado;
- **Files**: Classe utilitária responsável pela manipulação de um **Path** como busca de arquivos;
- **Buffers**: Recipientes para os dados;

- Charsets e seus decodificadores e codificadores associados:** Responsáveis pela tradução entre bytes e caracteres Unicode;
- Canais de vários tipos:** Representam conexões com as entidades que realizam operações I/O;
- Seletores e chaves de seleção:** Juntamente com os canais selecionáveis, definem uma especificação I/O multiplexada sem bloqueio.

16.4.2. Path, Paths e Files

A nova API NIO traz o pacote **java.nio.file** e suas principais classes (**Path**, **Paths** e **Files**), em que temos uma reformulação das rotinas antes executadas pela classe **java.io.File**.

Assim como a classe **File**, um **Path** representa um arquivo ou diretório do sistema operacional. Para criar um **Path**, utilizamos uma das sintaxes a seguir:

```
Path path1 = Paths.get("C:\\Meus Documentos\\docs");
Path path2 = Paths.get("Arquivo_Local.xml");
Path path3 = Paths.get("../\\Arquivo_Nivel_acima.mp3");
```

Veja, a seguir, alguns de seus métodos:

Método	Descrição
toAbsolutePath()	Retorna uma String contendo o caminho completo do Path .
getFileName()	Retorna uma String contendo o nome do arquivo ou diretório especificado pelo Path .
getParent()	Retorna o Path referente ao diretório que contém este Path .
resolve(String)	Cria um novo Path para um arquivo ou subdiretório contido no diretório especificado por este Path .

Agora, veja um exemplo:

```
J ExemploPath.java X
1 import java.nio.file.Path;
2 import java.nio.file.Paths;
3
4 public class ExemploPath {
5
6     public static void main(String[] args) {
7
8         Path poema = Paths.get("resources\\poema.txt");
9
10        /* Exibe o caminho especificado pelo path. */
11        System.out.println(poema);
12
13        /* Exibe o caminho completo do path a partir da raiz. */
14        System.out.println(poema.toAbsolutePath());
15
16        /* Exibe apenas o nome do arquivo. */
17        System.out.println(poema.getFileName());
18
19        /* Exibe o caminho completo do diretório pai. */
20        System.out.println(poema.getParent().toAbsolutePath());
21    }
22 }
```

O resultado será o seguinte:

```
Console X Problems @ Javadoc Declaration Task List
<terminated> ExemploPath [Java Application] C:\Java\jdk\jdk1.8.0_05\bin\java
resources\poema.txt
C:\Meus Documentos\Estudos\resources\poema.txt
poema.txt
C:\Meus Documentos\Estudos\resources
```

Utilizamos a classe **Files** (pacote `java.nio.file`) na manipulação de um **Path**. Todos os seus métodos são estáticos. Alguns deles são listados na tabela a seguir:

Método	Descrição
<code>Files.exists(path)</code>	Verifica se o Path especificado (arquivo ou diretório) existe, retornando um boolean .
<code>Files.isDirectory(path)</code>	Verifica se o Path especificado existe e é um diretório.
<code>Files.isRegularFile(path)</code>	Verifica se o Path especificado existe e é um arquivo.
<code>Files.size(path)</code>	Retorna um long contendo o tamanho em bytes do arquivo especificado pelo Path .
<code>Files.createDirectory(path)</code>	Cria um diretório referente ao Path especificado.
<code>Files.createDirectories(path)</code>	Cria recursivamente os diretórios referentes ao Path especificado.
<code>Files.move(path1, path2)</code>	Move o arquivo ou diretório especificado por um Path .
<code>Files.copy(path1, path2)</code>	Copia o arquivo ou diretório especificado por um Path .
<code>Files.delete(path)</code>	Apaga o arquivo ou diretório especificado.
<code>Files.createFile(path)</code>	Cria um arquivo de tamanho zero (arquivo vazio), conforme o Path especificado.
<code>Files.list(path)</code>	Retorna um Stream<Path> apontando para os arquivos e subdiretórios contidos no Path especificado, a partir do qual pode-se criar uma expressão lambda para processá-los.

Veja um exemplo:

```
J ExemploFiles.java X
1+ import java.io.IOException;
2
3  public class ExemploFiles {
4
5      public static void main(String[] args) {
6
7          try {
8
9              Path arquivo = Paths.get("resources\\poema.txt");
10             Path novaPasta = Paths.get("resources\\textos\\rascunhos\\vazio");
11             Path pasta = Paths.get("C:\\Meus Documentos\\Mp3");
12
13             if (!Files.exists(arquivo)) {
14                 System.out.println("O item não existe.");
15             } else if (Files.isDirectory(arquivo)) {
16                 System.out.println("O item é um diretório.");
17             } else if (Files.isRegularFile(arquivo)) {
18                 System.out.println("O item é um arquivo.");
19             }
20
21             /* Exibe */
22             System.out.println(Files.size(arquivo));
23
24             /* Cria o diretório rascunhos. */
25             Files.createDirectories(novaPasta);
26
27             /* Exibe o conteúdo do diretório Mp3 */
28             Files.list(pasta).forEach(f -> System.out.println(f.getFileName()));
29
30         } catch (IOException e) {
31             e.printStackTrace();
32         }
33     }
34 }
35 }
36 }
37 }
```

Agora, confira o resultado:

```
Console X Problems @ Javadoc Declaration Task List
<terminated> ExemploFiles [Java Application] C:\Java\jdk\jdk1.8.0_05\bin\ja
O item é um arquivo.
272
BobDylan.mp3
BondeDoTigrao.mp3
BaraoVermelho.mp3
InformationSociety.mp3
ToquinhoEVinicius.mp3
RaulSeixas.mp3
XitaozinhoEXororo.mp3
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

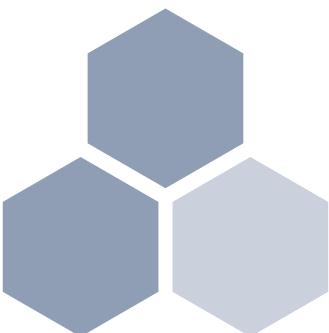
- A linguagem Java trabalha com um fluxo controlável de entrada e saída de informações, chamado **stream**, que também é definido como fluxo **Input/Output** (entrada/saída), ou simplesmente fluxo **I/O**. Além disso, o stream é manipulado por objetos e classes da linguagem Java;
- Na versão 1.4, foram introduzidas novas APIs de I/O (**NIO**), as quais fornecem novos recursos e melhor desempenho nas áreas de gerenciamento de buffer, rede escalável e arquivos I/O, suporte para conjunto de caracteres e correspondência de expressões regulares. Além disso, as APIs **NIO** complementam as especificações de I/O no pacote **java.io**.



16

Arquivos - I/O e NIO

Teste seus conhecimentos



Editora
IMPACTA



1. Qual das alternativas a seguir não é um método da classe OutputStream?

- a) write()
- b) print()
- c) flush()
- d) close()
- e) Todas as alternativas anteriores são métodos da classe OutputStream.

2. Qual é a função do método flush()?

- a) Regravar os dados já enviados.
- b) Estimular a gravação de bytes de saída contidos em buffers.
- c) Fechar o stream de saída, como determina o contrato geral.
- d) Grava o byte especificado para o stream de saída.
- e) Nenhuma das alternativas anteriores está correta.

3. Qual é a função da classe FileInputStream?

- a) Ler arquivos de texto.
- b) Ler streams de bytes não processados.
- c) Ler não apenas caracteres, mas também as linhas e os arrays de caracteres.
- d) Apenas representar os metadados de um arquivo.
- e) Nenhuma das alternativas anteriores está correta.

4. Qual é a função dos paths na linguagem Java?

- a) Indicar arquivos e/ou diretórios.
- b) Criar um caminho de diálogo entre dois objetos.
- c) Criar um caminho que posteriormente será convertido em vetor.
- d) Todas as alternativas anteriores estão corretas.
- e) Nenhuma das alternativas anteriores está correta.



16

Arquivos - I/O e NIO

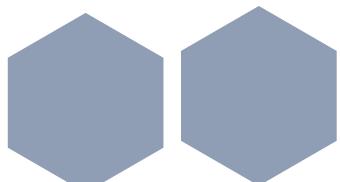


Mãos à obra!

Willians Martins
305.009.178-45



Editora
IMPACTA



Laboratório 1

A – Escrevendo caracteres em um arquivo externo

1. Crie uma classe com o nome **Cap16_Lab1** e insira a estrutura básica de um programa Java;
2. Importe todas as classes do pacote **java.io**;
3. Crie um método estático com o nome **escrever**, que não retorne nada e que receba uma variável do tipo **String** de nome **texto** como parâmetro;
4. Dentro do método, crie um bloco **try/catch** e, nele, crie um objeto **FileOutputStream** para o arquivo **Cap16_Lab1.txt** com o nome **arquivo**;
5. Crie um objeto **DataOutputStream** com o nome **dados** passando o objeto **arquivo** como parâmetro;
6. Chame o método **writeChars** do objeto **dados** passando como parâmetro a variável **texto**;
7. Insira a exceção **IOException** no **catch** e, dentro dele, imprima o erro.

B – Lendo caracteres de um arquivo externo

1. Crie um método estático com o nome **ler**, que não retorne nada e que não receba nenhum parâmetro;
2. Dentro do método, crie um bloco **try/catch** e, nele, crie um objeto **FileInputStream** para o arquivo **Cap16_Lab1.txt** com o nome **arquivo**;
3. Crie um objeto **DataInputStream** com o nome **dados** passando o objeto **arquivo** como parâmetro;
4. Crie um loop **while** que rode enquanto o método **available()** do objeto **dados** retornar um número maior que 0;
5. Dentro do **while**, atribua o método **readChar()** do objeto **dados** a uma variável do tipo **char** com o nome **c**;
6. Imprima o caractere **c** na tela;
7. Insira a exceção **IOException** no **catch** e, dentro dele, imprima o erro;
8. Dentro do método **main**, chame o método **escrever** passando como parâmetro o texto “**Capítulo 16\nLaboratório 1**” e, logo depois, chame o método **ler**;
9. Compile e execute o programa.

Laboratório 2

A – Copiando arquivos entre diretórios

Neste exercício, utilizaremos as classes **Paths**, **Path** e **Files**, do pacote **java.nio.file**, para copiar todos os arquivos de um diretório, gerando um backup.

1. Descompacte o arquivo **documentos.zip** fornecido pelo instrutor para algum diretório de seu sistema de arquivos. Isto deverá gerar um diretório chamado **documentos**;
2. Crie uma classe com o nome **Cap16_Lab2** e, dentro dela, crie:
 - Uma constante estática chamada **PASTA_ORIGEM** do tipo **String** contendo o caminho completo do diretório **documentos**;
 - Outra constante estática chamada **PASTA_BACKUP** contendo o caminho do diretório de backup. Este diretório deverá estar vazio ou inexistente;
 - O método **main()**.
3. No método **main()**, crie duas variáveis, **origem** e **backup**, do tipo **Path**, apontando para os diretórios especificados pelas constantes **PASTA_ORIGEM** e **PASTA_BACKUP**;
4. Crie um bloco **try/catch** contendo o tratamento para **IOException**. No bloco **catch**, utilize o método **printStackTrace()** sobre a exceção. Todo o código a seguir deverá ser colocado dentro do bloco **try**;
5. Verifique se o diretório **backup** existe. Caso não exista, crie-o. Para isso, utilize os métodos **Files.exists()** e **Files.createDirectory()**;
6. Crie uma variável chamada **streamOrigem** do tipo **Stream<Path>** e, nela, coloque o stream gerado pelo método **Files.list(origem)**;
7. Execute o método **forEach()** sobre **streamOrigem**, passando uma expressão lambda que irá, para cada path **p** (que representa um arquivo dentro do diretório de origem), criar uma cópia dentro do diretório **backup**. Para isso, utilize a instrução **Files.copy(p, backup.resolve(p.getFileName()))**;
8. Compile e execute.



16

Arquivos - I/O e NIO



Projeto Prático – Fase 4

Willians Martins
305.009.178-45



Editora
IMPACTA



Importação de filmes

Um arquivo importado do Excel (.CSV) contendo vários filmes do repositório de dados da base do IMDb será utilizado para que seja implementada a funcionalidade de importação dos filmes do arquivo para a base de dados da aplicação.

Para isso, a aplicação irá processar o arquivo, gerando os objetos do tipo **Filme**. Em seguida, cada filme processado será adicionado a uma lista de filmes que, ao final do processamento, estará disponível para inserção dos filmes “em lote” na base de dados da aplicação.

Esta inserção “em lote” será apenas processada na fase de implantação da base de dados, portanto, a aplicação deve simular a chamada ao DAO que realizará o mapeamento necessário no futuro.

Atividades

1. Ler o arquivo informado no parâmetro do método do “Controller”:

- Utilize a classe **Paths** para leitura do arquivo;
- Crie uma classe utilitária para processar o arquivo:
 - Cada linha do arquivo representa uma instância de filme;
 - Utilize a classe **Scanner**;
 - Utilize o método **split** da classe **String** para ler os atributos;
 - Crie uma coleção e adicione os objetos a ela.
- Faça uma chamada ao DAO, passando a coleção criada como parâmetro e simulando que o processo de importação foi realizado.

 O armazenamento na base de dados será implementado em uma próxima fase.

17

Threads

- Programação multithreaded;
- Implementando multithreading;
- Construtores;
- Estados da thread;
- Scheduler;
- Prioridades das threads;
- Sincronização;
- Bloqueios;
- Deadlock;
- Interação entre threads.

17.1. Introdução

Thread é uma classe em Java que permite a execução de diferentes tarefas simultaneamente. O termo thread diz respeito a linhas que compartilham um mesmo contexto e cuja execução ocorre de forma concomitante.

Uma thread representa um fluxo de controle em um processo, ou seja, cada thread em execução dentro de um programa apresenta:

- Um início;
- Uma sequência;
- Um ponto de execução;
- Um final.

O ponto de execução ocorre em qualquer momento no decorrer da execução da thread. Há, também, os objetos **thread**. Estes formam a base para a programação multithreaded, que é a responsável por permitir que diferentes threads sejam executadas simultaneamente em um único programa.

A linguagem Java possui suporte nativo às threads, contudo, em programas Unix, as threads são implementadas por determinadas bibliotecas.

Embora diferentes threads sejam executadas simultaneamente em uma única aplicação, a execução de cada uma delas ocorre de forma independente e, praticamente, de forma paralela. Quanto às instruções presentes em uma thread, estas são processadas de maneira sequencial, gerando uma fila de instruções.

Em suma, thread é um recurso que possibilita a utilização mais eficaz do processamento de uma máquina na medida em que permite a realização de diversas tarefas simultaneamente.

17.2. Programação multithreaded

O conceito de multiprocessamento é bastante importante para a compreensão do conceito de multithreaded. Multiprocessamento refere-se à utilização compartilhada da CPU por vários processos, os quais levam um determinado período de tempo para serem executados. Essa execução simultânea de vários processos ocorre em grande parte dos sistemas atuais, que possuem diversos processadores ou núcleos de processamento, capazes de realizar tarefas de forma concomitante. A programação multithreaded procura tirar vantagem dos múltiplos processadores disponíveis hoje em dia nos computadores modernos.

Multithreaded se refere a um programa que permite a execução simultânea e em paralelo de várias linhas, ou seja, multithreaded significa programação a partir de diversas linhas de execução. Um programa multithread é composto por duas ou mais threads, cada qual definindo um caminho de execução.

O multithreading possibilita o desenvolvimento de programas que utilizam o máximo da capacidade oferecida pela CPU. Com isso, há uma redução do tempo ocioso dessa CPU, o que é importante para os ambientes aos quais a Java é destinada, uma vez que é normal que apresentem tal ociosidade. Isso significa que o multithreading permite ao programador tirar proveito do tempo ocioso. Para que você comprehenda melhor esta questão, veja a seguinte situação presente em uma determinada rede:

- A taxa de transmissão de dados é realizada de forma mais lenta do que a capacidade do micro em processá-los;
- A velocidade em que os arquivos são lidos e escritos é mais baixa do que a capacidade da CPU em processá-los;
- A entrada fornecida pelos usuários é realizada com uma velocidade menor do que a do computador.

O programa precisa aguardar a conclusão de cada uma dessas tarefas para poder iniciar a outra, quando se encontra em um ambiente no qual há uma única thread, fazendo com que haja ociosidade da CPU na maior parte do tempo. Isso é resolvido pelo multithreading.

O multithreading, portanto, é uma forma específica de multitarefas, um recurso bastante conhecido por estar presente em grande parte dos sistemas operacionais mais atuais. São dois os tipos de multitarefas: baseadas em processo e baseadas em threads.

17.2.1. Multitarefa baseada em processo

Tendo em vista que é possível definir um processo como sendo um programa em execução, multitarefa baseada em processo é definida como um recurso capaz de permitir que um micro execute simultaneamente dois ou mais programas. Este é o tipo de multitarefa mais conhecido.

Um programa, neste tipo de multitarefa, é definido como a menor unidade de código que o scheduler é capaz de despachar. O scheduler é o agendador de tarefas. Sendo assim, a multitarefa baseada em processos trabalha com a execução de uma forma mais ampla.

17.2.2. Multitarefa baseada em threads

Na multitarefa baseada em threads, uma thread é definida como a menor unidade de código que o scheduler é capaz de executar, ou seja, ele permite que um programa execute duas ou mais tarefas de forma simultânea.

A multitarefa baseada em threads exige uma quantidade menor de recursos do que a exigida pela multitarefa baseada em processos, uma vez que as threads, ao contrário dos processos, são tarefas leves. Além disso, um único espaço de endereço e um único processo são compartilhados pelas threads.

A comunicação inter-threads é estabelecida de forma rápida, bem como o chaveamento de contexto que ocorre entre threads, o qual, além de ser rápido, também é fácil. A multitarefa baseada em threads é oferecida pela linguagem de programação Java e, também, é mantida sob controle por ela.

 Embora Java ofereça e controle a multitarefa baseada em threads, é a multitarefa baseada em processos a utilizada pelos programas Java. Contudo, não são esses programas que controlam esse tipo de multitarefa, mas sim o sistema operacional.

A programação multithreaded também é realizada para os sistemas operacionais Windows, mas o fato de Java gerenciar a multithreading faz com que a programação multithreaded com a linguagem Java apresente menos complexidade, na medida em que os detalhes deixam de ser uma preocupação para o programador.

17.3. Implementando multithreading

Quando você trabalha com a linguagem Java, a implementação do recurso multithreaded pode ser realizada de duas formas distintas: por meio da implementação da interface **Runnable** ou por meio da extensão da classe **Thread**.

Quanto à criação de threads, esta toma como base o fato de cada thread ser capaz de executar o método **run()**, seja este o seu próprio método ou o método de outros objetos. Sendo assim, a criação de threads pode ser realizada:

- Por meio de uma classe que realize a implementação da interface **java.lang.Runnable**;
- Por meio de uma subclasse referente à classe **java.lang.Thread**.

A implementação da interface **Runnable** ocorre com mais frequência do que a extensão da classe **Thread**. A extensão da classe **Thread** é um procedimento menos complexo, contudo, em modelos OO (Orientados a Objetos), este não é o procedimento mais adequado, tendo em vista ser o objetivo da herança prover uma versão mais específica da superclasse de caráter genérico.

Considerando o modelo OO, estender apenas a classe **Thread** é um procedimento adequado nas situações em que se tem uma versão mais específica dessa classe.

Quanto à implementação da interface **Runnable**, esta se torna mais adequada por conta de uma única thread ser capaz de executar as tarefas necessárias, pois, dessa forma, a classe poderá estender outra classe.

É a partir de uma instância de **java.lang.Thread** que uma thread é iniciada, cujo gerenciamento – isto é, criação, inicialização e paralisação – é realizado por métodos presentes na classe **Thread**. Neste caso, os principais métodos são: **run()**, **start()**, **sleep()**, **yield()**.

O método **run()**, é utilizado para iniciar uma tarefa. Nas situações em que você deseja executar uma tarefa em segundo plano, ou seja, executar uma tarefa na própria thread, essa thread será a executora do código. Vale destacar que o código sempre deve ser escrito em uma thread separada do método **run()**.

Embora o método **run()** realize a chamada de diversos métodos, o primeiro método a ser chamado pela thread de execução sempre será o **run()**. Isso significa que o método **run()** entrará em uma das classes utilizadas com a finalidade de determinar a tarefa da thread.

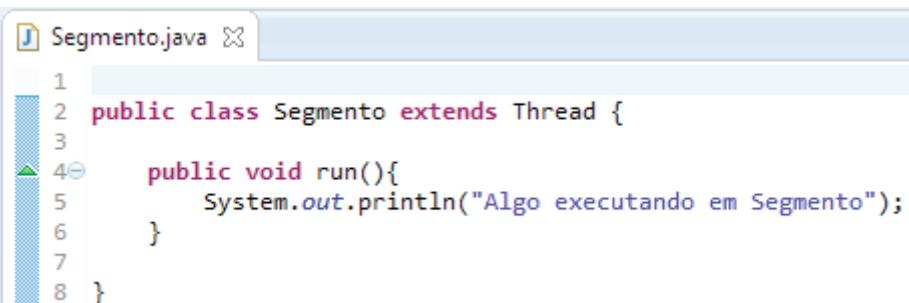
17.3.1.java.lang.Thread

Estender a classe **java.lang.Thread** significa criar uma classe a partir dela, ou seja, criar uma classe que obtenha a herança de **java.lang.Thread**. A partir desta criação, realize o seguinte procedimento:

1. Crie um objeto da classe criada;
2. Chame o método **start()**, o qual é responsável por registrar a thread no escalonador.

A subclasse de **java.lang.Thread** pode sobrescrever o método **run()**, embora isso não seja obrigatório, uma vez que a classe em questão fornece uma implementação para esse método. Tal implementação, entretanto, não tem quaisquer funções, o que faz com que a classe **Thread** sobrescreva o método **run()** com a finalidade de colocar em seu lugar o código a ser executado.

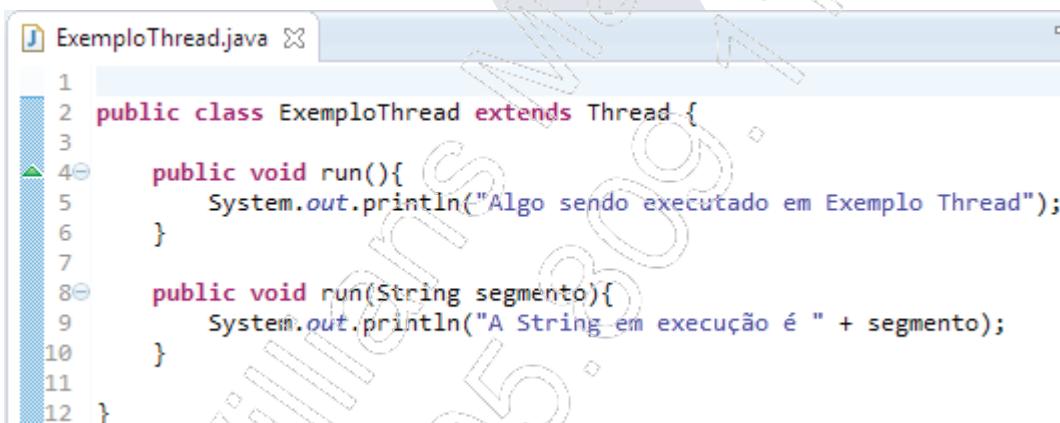
Estender a classe **Thread** e sobrescrever o método **run()** são duas tarefas realizadas para definir que um código seja executado em uma thread à parte, como demonstrado no exemplo seguinte:



```
1  public class Segmento extends Thread {  
2  
3  
4  public void run(){  
5      System.out.println("Algo executando em Segmento");  
6  }  
7  
8 }
```

O exemplo apresentado, porém, tem algumas limitações, pois, ao estender a classe **Thread**, não será possível estender outras classes. Ainda, destacamos que a herança da classe **Thread** não traz um comportamento estritamente necessário, uma vez que a utilização de uma thread depende de sua instanciação.

O exemplo descrito a seguir demonstra como sobrecarregar o método **run()** em uma classe estendida de **Thread**. Observe:



```
1  public class ExemploThread extends Thread {  
2  
3  
4  public void run(){  
5      System.out.println("Algo sendo executado em Exemplo Thread");  
6  }  
7  
8  public void run(String segmento){  
9      System.out.println("A String em execução é " + segmento);  
10 }  
11  
12 }
```

Nesse exemplo, **String segmento** refere-se ao parâmetro do método **run()** sobreposto, o qual será invocado apenas se uma chamada explícita for efetuada. Sendo assim, o método em questão não será utilizado com a base para nova pilha de chamadas.

Apenas o método **public void run() deve ser sobreposto. Se o método **public void start()** é sobreposto, a thread não é criada na Java Virtual Machine nem registrada no escalonador.**

Observe o exemplo a seguir:

```
ExemploThread2.java
1 public class ExemploThread2 extends Thread {
2
3     private String mensagem;
4
5     public ExemploThread2(String mensagem){
6         this.mensagem = mensagem;
7     }
8
9     public void run(){
10        while(true){
11            System.out.println(mensagem);
12        }
13    }
14
15    public static void main(String args[]){
16        ExemploThread2 segmento = new ExemploThread2("Executando a nova thread");
17        segmento.start();
18        while(true){
19            System.out.println("Executando a thread principal");
20        }
21    }
22
23 }
24 }
```

17.3.2. java.lang.Runnable

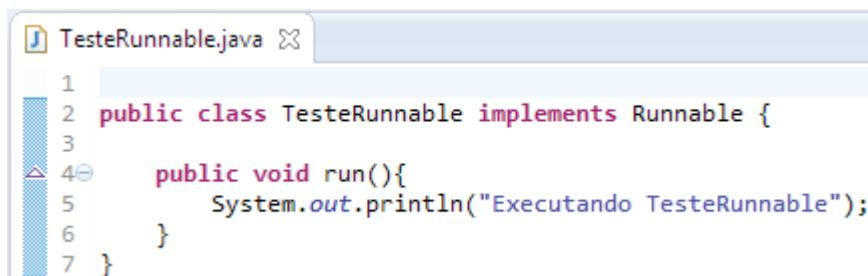
Uma thread também pode ser criada por meio da execução do método **public void run()** pertencente a algum objeto que esteja fora da árvore de herança de **java.lang.Thread**. Porém, tendo em vista que a thread é sempre representada por uma subclasse de **java.lang.Thread**, a implementação deve ser realizada por meio do construtor **Thread(Runnable r)** da classe **Thread**. Isto para que essa thread seja capaz de executar o código do objeto de outra classe.

Esse construtor **Thread** recebe uma referência a um objeto responsável por implementar a interface **java.lang.Runnable**. Tal referência é o seu parâmetro. Assim que esse construtor é chamado, ocorre o seguinte:

1. Armazena-se a referência que é passada como argumento;
2. A thread é registrada no escalonador assim que o método **start()** dessa thread é invocado;
3. O método **run()** responsável por implementar a interface **Runnable** é executado assim que o processador é obtido pela thread.

Runnable é uma interface que não apresenta complexidade e que realiza a declaração do método **public void run()** somente. A implementação de tal interface permite a extensão de qualquer classe, bem como permite definir a execução de uma thread à parte.

O formato apresentado pela interface **Runnable** é o seguinte:



```
1  public class TesteRunnable implements Runnable {  
2  
3  
4  public void run(){  
5      System.out.println("Executando TesteRunnable");  
6  }  
7 }
```

Este formato demonstra que há um código cujo processamento pode ser realizado por uma thread de execução, o que ocorre independente do mecanismo selecionado.

17.4. Construtores

Instanciar uma classe **Thread** é o primeiro passo para criar uma thread de execução, embora ainda seja necessário um objeto **Thread** nas situações em que o método **run()** se encontra na classe de implementação da interface **Runnable** e nas ocasiões em que esse método se encontra em uma classe estendida de **Thread**.

A instanciação da classe **Thread** é bastante simples, se realizada a partir da extensão dessa própria classe, como demonstrado a seguir:

```
Segmento s = new Segmento();
```

No entanto, a instanciação da classe **Thread** é mais complexa quando realizada a partir da implementação da interface **Runnable**, uma vez que ainda será necessária uma instância de **Thread**, a fim de que o código seja executado pela thread. Neste caso, o código do método **run()** deverá ser dividido em duas classes, que são:

- A classe da implementação da interface **Runnable** para o código da tarefa realizada pela thread;
- A classe **Thread** para o código da thread.

Para realizar a instanciação da classe **Runnable**, você deve utilizar o seguinte código:

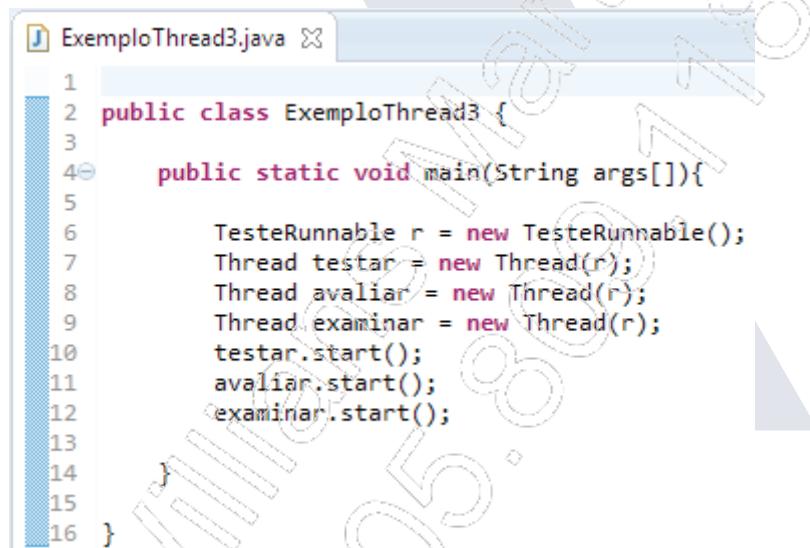
```
TesteRunnable r = new TesteRunnable();
```

A fim de criar uma instância de `java.lang.Thread` para a qual a tarefa deve ser passada, utilize o código seguinte:

```
TesteRunnable r = new TesteRunnable();
Thread s = new Thread(r); // passa Runnable para Thread
```

Você pode utilizar um construtor sem argumentos a fim de criar uma thread. Ao fazer isso, essa thread iniciará o trabalho chamando seu próprio método `run()`. Nas situações em que estender a classe `Thread`, a intenção é que a thread chame seu próprio método. Contudo, quando você implementa a interface `Runnable`, faz-se necessário alertar essa thread quanto à execução de seu método.

Destacamos que somente uma instância de `Runnable` pode ser passada para diversos objetos `Thread`. Como resultado, diversas threads terão como destino uma única instância, o que indica que uma mesma tarefa será realizada por diversas threads de execução. O código descrito a seguir demonstra tal situação:



```
ExemploThread3.java
1 public class ExemploThread3 {
2     public static void main(String args[]){
3         4
4         TesteRunnable r = new TesteRunnable();
5         Thread testar = new Thread(r);
6         Thread avaliar = new Thread(r);
7         Thread examinar = new Thread(r);
8         testar.start();
9         avaliar.start();
10        examinar.start();
11    }
12 }
```

Você viu, até este momento, como instanciar uma thread, mas não como iniciá-la. Uma thread não iniciada ainda não é uma thread de execução. Isso significa que ela está no estado `new`, ou seja, ela não é considerada como uma thread ativa.

A fim de verificar se uma determinada thread está ativa, você pode chamar o método `isAlive()` na instância da classe `Thread`. Este método é capaz de definir se uma thread foi iniciada sem que o seu método `run()` tenha sido concluído. Para que a thread seja considerada ativa, a JVM necessita de algum tempo para configurá-la após a chamada do método `start()`. Uma vez chamado este método, a thread será considerada inativa somente após ter sido descartada.

17.5. Estados da thread

As threads, que são utilizadas na linguagem Java com a finalidade de manter a sincronia entre as execuções realizadas em um determinado ambiente, apresentam-se em um determinado estado. Os estados em que elas podem estar serão descritos na tabela a seguir:

Estado da thread	Descrição
New	Uma thread encontra-se no estado new quando já há uma instância de Thread criada, porém, seu método start() ainda não foi chamado. Assim, uma thread neste estado ainda está inativa, ainda não é uma thread de execução.
Ready to run	Uma thread neste estado está pronta para ser executada, mas ainda não foi selecionada pelo scheduler como sendo uma thread pronta para o processamento. Quando está no estado ready to run , a thread é considerada ativa. Ela entra neste estado pela primeira vez assim que seu método start() é chamado, mas pode entrar neste estado em outras situações, como no momento em que acaba de ser processada, ou quando retorna de outros estados, como resume , blocked e suspended .
Running	Uma thread neste estado está em execução, o que ocorre no momento em que o scheduler a seleciona, a partir do pool executável, como sendo um processo cuja execução deve ocorrer prontamente. Vale destacar que uma thread pode sair do estado running por algumas razões, desde uma decisão do scheduler para que ela saia, até o fato de a thread estar em um dos estados que serão descritos posteriormente: resume , blocked e suspended ; os quais indicam que a thread está no estado ready to run , mas não está no estado running .
Resume	A thread está neste estado quando retorna às suas atividades, no momento em que um determinado evento ocorre, após ter ficado no estado suspended ou blocked .
Blocked	A thread está neste estado quando se encontra aguardando por um recurso. Porém, ela retornará ao estado ready to run no momento em que este recurso for disponibilizado.
Suspended	A thread está no estado suspended quando seu código de execução a informou para que ficasse inativa durante um certo período de tempo. Nesta situação, a thread retorna ao estado ready to run assim que expira seu prazo de suspensão.
Terminated	A thread está neste estado nas situações em que seu método run() é finalizado, ou seja, nas situações em que a thread está inativa. Threads inativas não podem ser reativadas. Sendo assim, caso você chame o método start() de uma instância de Thread que esteja inativa, receberá uma exceção de tempo de execução.

A classe **Thread** contém os métodos **stop()** e **suspend()**, os quais permitem que uma thread informe a outra thread a respeito de sua suspensão. Contudo, esses métodos foram considerados depreciados e, dessa forma, não devem ser utilizados. Nesse rol de depreciados, também devem ser incluídos os métodos **destroy()** e **resume()**.

17.6. Scheduler

O scheduler é o responsável por definir quais threads serão executadas em um dado momento, além de fazer com que a thread saia do estado **ready to run**. Nas situações em que somente uma máquina realiza o processamento, é possível que apenas uma thread seja executada por vez, sendo que o scheduler é o responsável por determinar qual thread será processada.

É importante saber que, para ser selecionada pelo scheduler, uma thread deve estar qualificada, ou seja, deve estar no estado **ready to run**. Embora haja uma fila de threads a serem executadas, não é possível assegurar que a ordem dessa fila será obedecida. A ordem normal seria a seguinte:

- Assim que a execução da thread for concluída, esta passa para o final da fila;
- Esta thread, então, aguardará até que chegue em primeiro lugar da fila novamente para que possa ser, mais uma vez, executada.

A fila mencionada é, na verdade, chamada de pool executável, o que demonstra ainda mais o fato de não haver uma ordem certa de execução. Tal ordem de execução não é garantida porque o scheduler não pode ser controlado, mas apenas influenciado por meio de alguns métodos, os quais estão contidos tanto na classe **java.lang.Thread**, quanto na classe **java.lang.Object** (serão detalhadas posteriormente neste capítulo).

17.7. Prioridades das threads

Na linguagem Java, são atribuídas prioridades às threads representadas por números inteiros que, em geral, ficam em um intervalo de 1 a 10, sendo que a prioridade padrão é 5. Essa prioridade determina como a thread será executada em relação às outras. É essencial ter em mente que esse número não determina se a execução de uma thread será mais rápida do que a execução de outra, mas sim o momento em que será realizado o chaveamento de contexto, o chamado **context switch**.

O momento em que esse chaveamento ocorrerá é determinado a partir de algumas regras, as quais serão descritas a seguir:

- **Cessão do controle de uma thread**

Uma thread pode desistir do controle de maneira espontânea. Isso pode ser feito de três formas distintas: a thread pode ceder explicitamente esse controle; ela pode entrar em estado **blocked**; ou pode entrar em modo **sleep**. Quando uma thread cede o controle dessa maneira, todas as outras threads são analisadas e, dependendo da política de escalonamento (algoritmo do **scheduler** do sistema operacional corrente), pode ser que aquela thread, cuja prioridade é mais elevada, e a qual se encontra no estado **ready to run**, seja considerada para obter o controle da CPU.

- **Deslocamento de threads**

É possível que uma thread seja deslocada por outra cuja prioridade é mais alta. Caso a thread cuja prioridade é mais baixa não ceda o controle, ela é deslocada por uma de prioridade mais alta independentemente das tarefas que estiver realizando, uma vez que, teoricamente, uma thread de prioridade mais elevada pode ser executada no momento necessário. O mecanismo que permite a execução de thread com prioridade mais alta no momento desejado é chamado de multitarefa preemptiva.

- **Competição entre threads**

Há situações nas quais threads que possuem a mesma prioridade disputam os ciclos da CPU.

Grande parte das JVMs possui um scheduler que utiliza um agendamento preemptivo com base em prioridades. Isso quer dizer que as JVMs utilizam a divisão de tempo, embora sua especificação não exija a implementação de um scheduler com essa divisão de tempo.

Um scheduler com divisão de tempo permite que um determinado período de tempo seja alocado a cada uma das threads. Após o término desse período, a thread retorna ao estado **ready to run** e, dessa maneira, possibilita a execução de uma outra thread.

Além das JVMs que utilizam um scheduler com divisão de tempo, há aquelas que utilizam um scheduler capaz de permitir que uma thread seja executada até o momento em que seu método **run()** seja finalizado. Entretanto, o scheduler que trabalha com prioridades para as threads ainda é o mais utilizado pelas Java Virtual Machines. Com este scheduler, é comum que as threads com prioridade mais elevada sejam executadas em primeiro lugar.

A prioridade é como uma garantia de ordem de execução de threads. Apesar disso, não é possível confiar que, por conta dessas prioridades, o comportamento de um determinado programa será o mais adequado. Ainda, você não pode confiar nas prioridades quando projetar um programa com diversas threads, porque, em última instância, o scheduler da JVM está atrelado ao tipo de algoritmo executado pelo sistema operacional.

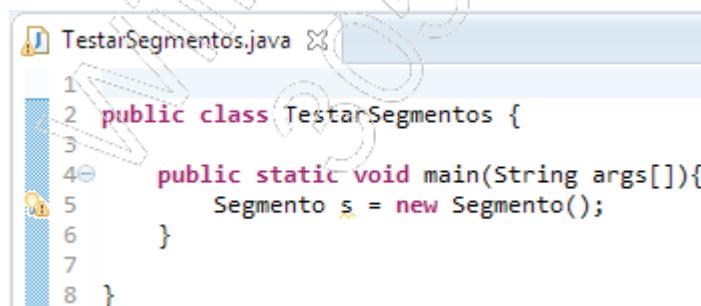
A melhor utilização das prioridades é para o aumento da eficiência apresentada por um programa, uma vez que essas prioridades para as threads são uma garantia de comportamento adequado do programa.

Nas situações em que as threads têm prioridades iguais, é papel do scheduler determinar se o tempo das threads de pool será dividido para que todos tenham oportunidades de execução equivalentes, ou se uma determinada thread será executada até que entre no estado **blocked** ou que seu método **run()** seja concluído.

Como já apresentado, a configuração das prioridades é realizada com números inteiros, que geralmente estão no intervalo de 1 a 10. Entretanto, esses valores também não são garantidos, pois caso existam dez threads cujas prioridades são distintas, e a execução do programa estiver sendo realizada em uma JVM cujo intervalo alocado é de apenas cinco prioridades, é possível que para uma mesma prioridade sejam mapeadas duas ou mais threads.

O intervalo das prioridades é determinado por três constantes da classe **Thread**: **Thread.MIN_PRIORITY**, **Thread.NORM_PRIORITY** e **Thread.MAX_PRIORITY**.

Veja, a seguir, as maneiras pelas quais é possível configurar as prioridades das threads:



```
TestarSegmentos.java
1
2 public class TestarSegmentos {
3
4     public static void main(String args[]){
5         Segmento s = new Segmento();
6     }
7
8 }
```

Nesse código, **s** está referenciando uma thread, a qual terá a mesma prioridade da thread principal. Isso ocorre porque essa thread está executando o código que realiza a criação da instância de **Segmento**.

Há outra maneira de configurar essa prioridade. Veja:

```
TesteRunnable r = new TesteRunnable();
Segmento s = new Segmento(r);
s.setPriority(7);
s.start();
```

Nesse código, a prioridade da thread é configurada por meio da chamada do método **setPriority()** em uma instância de **Thread**.

17.7.1. Método **yield()**

A função do método **yield()** é permitir que uma thread em estado **running** retorne ao estado **ready to run** para que outras threads com a mesma prioridade também possam ser processadas. Este método, porém, não garante que a thread em execução passe para o estado **ready to run** a fim de dar oportunidade de execução à outra thread.

Caso o processador esteja ocupado, o método **yield()** informa que uma determinada thread não está em execução e, por isso, a preferência pode ser passada a outra thread. No entanto, caso o processador não esteja ocupado, essa thread cuja execução estava paralisada é reativada de forma instantânea. Com isso, o método em questão é capaz de reduzir o tempo de espera para a execução de threads.

O método **yield()**, portanto, deve ser utilizado nas situações em que se deseja executar uma thread que tem prioridade igual à thread que está em execução no momento.

A sintaxe do método **yield()** é a seguinte:

```
public static void yield()
```

17.7.2. Método **join()**

Este método permite que uma thread seja adicionada ao final de outra thread. Isso significa que, se uma determinada thread não puder entrar em execução até que outra thread tenha concluído a sua, é mais adequado adicionar esta primeira thread àquela que está em execução. Assim, uma thread não será executada até que a outra tenha sido concluída. Este método aguarda até que a execução da thread para a qual foi chamado seja finalizada.

```
public final void join()
    throws InterruptedException
```

Algumas outras formas do método **join()** permitem determinar o período de tempo para que a thread especificada seja finalizada. O nome atribuído a esse método (**join** = juntar) deve-se ao fato de que a thread que o chama aguarda até que a thread especificada junte-se a ela.

Como suas principais características, destacamos que o método em questão é estático e pertence à classe **Thread**.

17.7.3. Método **isAlive()**

A função do método **isAlive()** é determinar se uma thread ainda está em execução. Para isso, é preciso chamá-lo para a thread desejada. Caso seja chamado para uma thread que ainda esteja em execução, o valor retornado por **isAlive()** é **true**, caso contrário, o valor retornado é **false**.

O método **isAlive()**, que também está localizado na classe **Thread**, possui a seguinte forma:

```
public final boolean isAlive()
```

17.7.4. Método **sleep()**

A função deste método é permitir que a thread dentro da qual foi chamado fique no estado **suspended** durante um determinado período de tempo, que é definido em milissegundos. Este método é utilizado com a finalidade de tornar o percurso da thread pelo código mais lento, bem como para forçar essas threads a dar oportunidade a outras.

O método **sleep()**, pertencente à classe **Thread**, força a thread a entrar no estado **suspended** antes que ela retorne ao estado **ready to run**. A fim de que volte a este estado, é preciso que a thread seja despertada. É importante ter em mente, porém, que o fato de a thread ter sido despertada não significa que ela será executada, uma vez que ela retorna ao estado **ready to run**, tendo que aguardar até que possa entrar no estado **running**.

O tempo que é determinado por meio do método **sleep()** refere-se somente ao período mínimo em que a thread não será executada, o que faz deste método um timer suficiente, porém, impreciso, uma vez que não é possível garantir que a execução da thread será iniciada no momento em que o período determinado finalizar ou, ainda, quando essa thread for despertada.

O **sleep()** é um método estático que pode ser colocado em qualquer local dentro do código, pois todo esse código será executado por uma thread e, assim que a execução do código atingir a chamada deste método, a thread atual entrará em estado **suspended**.

A forma geral de **sleep()** é a seguinte:

```
public static void sleep(long millis)
    throws InterruptedException
```

Em que:

- **millis**: É o parâmetro que determina o período de suspensão da thread em milissegundos;
- **InterruptedException**: É uma exceção que pode ser lançada pelo método **sleep()**.

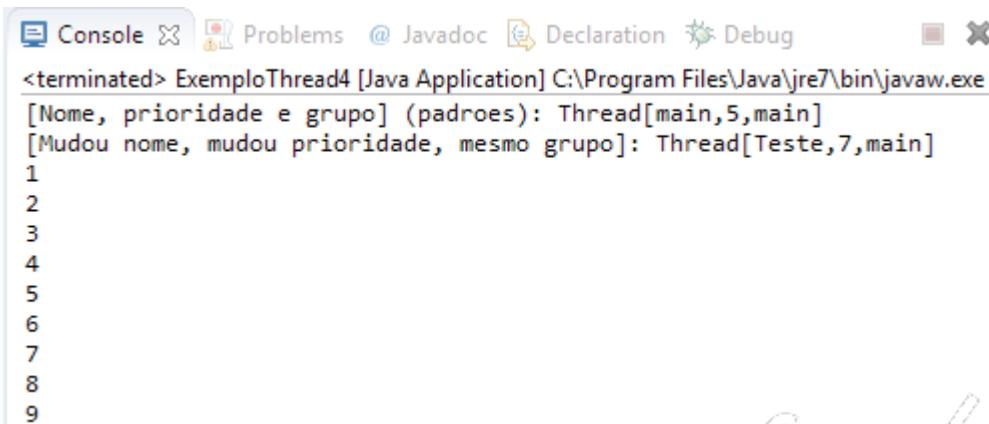
O método **sleep()** também possui uma outra forma, mas esta apenas é útil nos ambientes em que a determinação do período de suspensão pode ser realizada em milissegundos e em nanossegundos. A forma de utilização em questão é:

```
public static void sleep(long millis, int nanos)
    throws InterruptedException
```

Veja o exemplo a seguir:

```
ExemploThread4.java
1
2 public class ExemploThread4 {
3
4     public static void main(String args[]){
5
6         Thread segmento = Thread.currentThread();
7         System.out.println("[Nome, prioridade e grupo] (padroes): " + segmento);
8         segmento.setName("Teste");
9         segmento.setPriority(7);
10        System.out.println("[Mudou nome, mudou prioridade, mesmo grupo]: "
11                           + segmento);
12
13        try{
14            for(int x = 1; x < 10; x++){
15                System.out.println(x);
16                Thread.sleep(500);
17            }
18        }catch(InterruptedException e){
19
20        }
21    }
22 }
```

Depois de compilado e executado o código anterior, o resultado será como o exibido na figura seguir:



The screenshot shows a Java application window titled "ExemploThread4 [Java Application]". The console tab is active, displaying the following text:
<terminated> ExemploThread4 [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe
[Nome, prioridade e grupo] (padroes): Thread[main,5,main]
[Mudou nome, mudou prioridade, mesmo grupo]: Thread[Teste,7,main]
1
2
3
4
5
6
7
8
9

17.8. Sincronização

Os recursos compartilhados apenas podem ser utilizados por uma thread de cada vez, porém, no decorrer da execução de alguns programas, é possível que haja mais de uma thread necessitando de acesso a esses recursos. Para resolver essa situação, utilize a sincronização.

A sincronização é importante porque, caso duas threads chamem um método em um mesmo objeto, o estado do objeto pode ser corrompido, podendo afetar outras partes do programa, se estas mantiverem dados compartilhados com esse estado. Quando o estado de um objeto é corrompido, os valores de sua variável de instância são inconsistentemente alterados.

O monitor de bloqueio de acesso, definido como um objeto cuja função é ser uma trava mútua, é um conceito bastante importante para o processo de sincronização de threads. Apenas uma thread por vez pode estar com esse monitor.

Assim que uma thread consegue acessar o recurso desejado, o acesso a ele é bloqueado às outras threads. Quando isso acontece, dizemos que a thread entrou no monitor, e qualquer outra thread que tentar entrar nele ficará em estado **suspended** até que a thread inicial deixe o monitor. A thread encontrada no monitor pode entrar novamente nele se desejar.

17.8.1. Palavra-chave `synchronized`

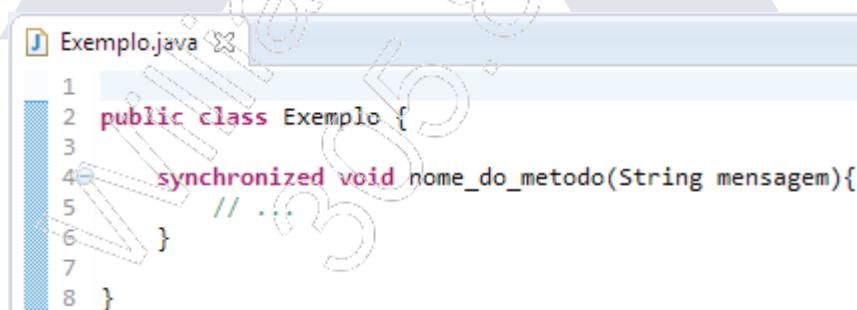
Quando **synchronized** é utilizada na declaração de um método, somente uma thread por vez pode acessar esse método, ou seja, antes da execução desse método, um monitor de bloqueio de acesso é adquirido. Este monitor de bloqueio é o próprio objeto ao qual o método pertence, ou a classe, caso seja estático.

17.8.1.1. Race condition

Race condition é o nome dado a uma situação na qual um mesmo conjunto de dados é manipulado simultaneamente por diversos processos, e em que temos um resultado determinado pela ordem em que tais processos realizaram seus acessos aos dados.

Mais especificamente, temos uma **race condition** nas situações em que mais de uma thread acessa o mesmo método de forma concomitante, causando uma competição por esse método. As consequências de uma **race condition** podem ser desastrosas, mas também há momentos em que um programa funciona de forma adequada apesar desse problema. Isso acontece porque uma **race condition** pode ocorrer de forma discreta, bem como não pode ser prevista, uma vez que o momento em que um chaveamento de contexto ocorrerá não é garantido.

O código a seguir mostra como solucionar os problemas causados por uma **race condition**. Essa solução envolve a restrição do acesso ao método a apenas uma thread por vez:



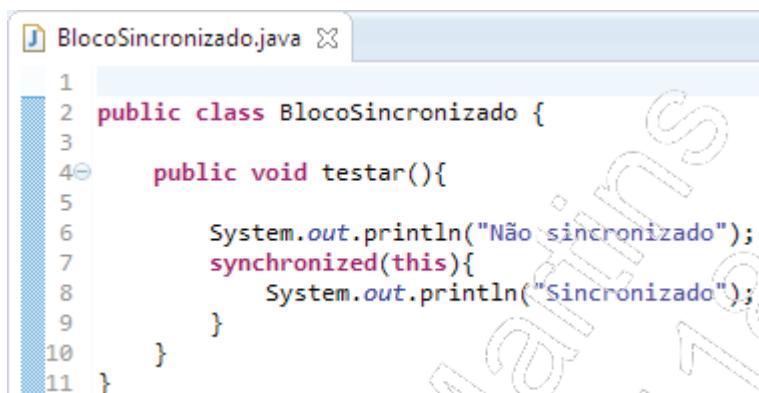
```
J Exemplo.java
1
2 public class Exemplo {
3
4     synchronized void nome_do_metodo(String mensagem){
5         // ...
6     }
7
8 }
```

O exemplo apresenta a definição do método com **synchronized**, o que evita que outras threads chamem um método enquanto este estiver sendo utilizado.

17.8.2. Bloco sincronizado

O processo de sincronização deve ser realizado somente sobre o trecho de código necessário para a proteção dos dados, uma vez que tal processo exclui a possibilidade de haver concorrência. Sendo assim, caso um método apresente um escopo maior do que o necessário para sincronização, você pode reduzir tal escopo de forma a sincronizar apenas uma parte desse método, ou seja, apenas um bloqueio, o qual é chamado de bloco sincronizado.

Um bloco sincronizado apresenta o formato descrito a seguir:



```
1  public class BlocoSincronizado {  
2      public void testar(){  
3          System.out.println("Não sincronizado");  
4          synchronized(this){  
5              System.out.println("Sincronizado");  
6          }  
7      }  
8  }
```

Um código é executado em contexto sincronizado quando a thread processa não somente esse código a partir de um bloco sincronizado, mas também o código dos métodos presentes nesse bloco.

É preciso determinar o bloqueio de objeto que será utilizado quando você sincroniza um bloco de código, o que permite ter, em um único objeto, mais de um bloqueio para o processo de sincronização de códigos.

Quanto aos métodos estáticos, estes podem ser sincronizados, porém, essa sincronização precisa de um único bloqueio para toda a classe. Esse bloqueio é o da instância de `java.lang.Class`, a qual representa cada classe carregada em Java. A sincronização de métodos estáticos é feita da seguinte maneira:

```
public static synchronized void getCount(){ }
```

A criação de blocos sincronizados é a solução encontrada nas situações em que classes não podem ser alteradas a partir da declaração de seus métodos como sincronizados. Os blocos sincronizados permitem a sincronização do acesso aos objetos de uma classe cuja criação não foi realizada a partir dos conceitos de execução multithreaded.

Vale destacar que um bloco sincronizado garante que um método membro de um objeto apenas será chamado no momento em que a thread atual entrar no monitor desse objeto. Ao criar um bloco sincronizado, é possível inserir nele as chamadas ao método da classe que não é sincronizada, tal como demonstrado a seguir:

```
synchronized(objeto){  
    // comandos a serem sincronizados  
}
```

Os blocos não precisam ser colocados entre chaves {} quando for realizada a sincronização de comandos. Ressaltamos que o código descrito anteriormente (objeto) refere-se ao objeto a ser sincronizado.

17.9. Bloqueios

Bloqueios e sincronização são dois conceitos relacionados entre si, uma vez que a sincronização funciona com os bloqueios. Há algumas questões importantes a respeito de ambos que devem ser levadas em consideração:

- Os bloqueios de uma thread que entra em estado **suspended** também entram nesse estado;
- Uma classe não precisa que todos os seus métodos sejam sincronizados. Sendo assim, pode haver tanto métodos sincronizados quanto métodos não sincronizados em uma classe;
- Nas classes que possuem tanto métodos sincronizados quanto não sincronizados, estes poderão ser acessados por diversas threads;
- A sincronização é realizada apenas sobre os métodos, e não sobre as variáveis;
- Um objeto pode possuir somente um bloqueio;
- Diversos bloqueios podem ser utilizados por uma thread;
- Quando em uma classe são sincronizados dois métodos, somente um poderá ser acessado por uma única thread.

Os objetos da linguagem Java contêm um único bloqueio interno. Assim que o código de método desse objeto é sincronizado, esse bloqueio surge. Sendo assim, quando esse bloqueio é utilizado por uma determinada thread, os métodos sincronizados desse objeto não poderão ser acessados por qualquer outra thread. Esse bloqueio será liberado quando a thread sair do método sincronizado e outra entrar nele.

Em suma, quando o bloqueio de um objeto está sendo utilizado por uma thread, outra thread não poderá utilizar um método sincronizado desse objeto.

É preciso destacar que a sincronização deve ser realizada somente sobre os métodos que acessam dados a serem protegidos, uma vez que ela tem grande impacto sobre a performance.

Embora o bloqueio de um objeto possa ser acessado por uma thread de cada vez, cada uma dessas threads pode acessar mais de um objeto por vez. Esses bloqueios, então, são liberados gradativamente com o desenrolar da pilha.

Dessa forma, é possível que a thread obtenha o objeto e, ainda, utilize-o para chamar um método sincronizado presente nele, pois a JVM sabe que a thread em questão já possui o bloqueio.

Existem métodos que mantêm o bloqueio e existem métodos e classes que o liberam. Veja quais são esses métodos e classes:

- **Métodos e classes que liberam o bloqueio**
 - Classe `java.lang.Thread`;
 - Método `wait()`.
- **Métodos que mantêm o bloqueio**
 - Método `join()`;
 - Método `notify()`;
 - Método `sleep()`;
 - Método `yield()`.

O método `notify()` pode tanto manter quanto liberar o bloqueio. Ele o libera quando a thread sai do código sincronizado após o método ter sido chamado.



A classe `java.lang.Object` é a responsável por definir o método.

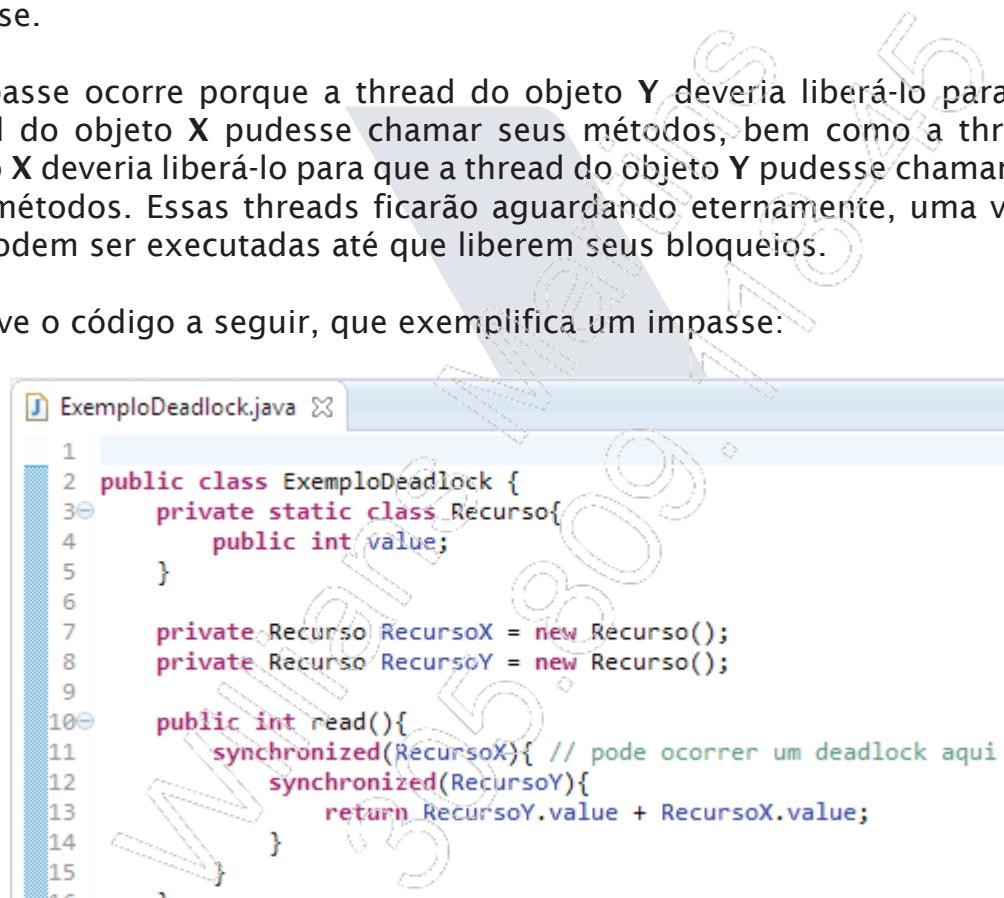
17.10. Deadlock

Um deadlock ocorre nas situações em que duas threads dependem uma da outra em um par de objetos que estão sincronizados. Em situações como esta, temos uma ou mais threads cuja execução está paralisada, esperando pelo acesso a um recurso que já foi alocado por outra thread.

Para uma melhor compreensão a respeito de um deadlock, considere os objetos X e Y. Uma determinada thread entrou no monitor do objeto X e outra thread entrou no monitor do objeto Y. Essas threads ficarão bloqueadas caso tentem chamar um método sincronizado do outro objeto, ou seja, caso a thread do objeto X tente chamar um método do objeto Y e vice-versa. Surge, assim, um impasse.

O impasse ocorre porque a thread do objeto Y deveria liberá-lo para que a thread do objeto X pudesse chamar seus métodos, bem como a thread do objeto X deveria liberá-lo para que a thread do objeto Y pudesse chamar um de seus métodos. Essas threads ficarão aguardando eternamente, uma vez que não podem ser executadas até que liberem seus bloqueios.

Observe o código a seguir, que exemplifica um impasse:



```
J ExemploDeadlock.java x
1
2 public class ExemploDeadlock {
3     private static class Recurso{
4         public int value;
5     }
6
7     private Recurso RecursoX = new Recurso();
8     private Recurso RecursoY = new Recurso();
9
10    public int read(){
11        synchronized(RecursoX){ // pode ocorrer um deadlock aqui
12            synchronized(RecursoY){
13                return RecursoY.value + RecursoX.value;
14            }
15        }
16    }
17
18    public void write(int x, int y){
19        synchronized(RecursoY){ // pode ocorrer um deadlock aqui
20            synchronized(RecursoX){
21                RecursoX.value = x;
22                RecursoY.value = y;
23            }
24        }
25    }
26 }
```

No código apresentado, há possibilidade de ocorrer um deadlock, o que é demonstrado nas linhas 11 e 19. Ele é justificado de acordo com a seguinte situação: **read()** foi iniciado por uma thread e **write()** foi iniciado por uma outra thread. O risco de impasse existe porque as duas threads são capazes de ler e gravar de forma independente. Já que a thread de leitura (**read**) possui o recurso X, e a thread de gravação (**write**) possui o recurso Y, ambas permanecerão paralisadas, aguardando a desistência uma da outra.

Entretanto, é bastante provável que o deadlock não aconteça em razão de a CPU passar da thread de leitura para a thread de gravação em um ponto determinado. A fim de eliminar qualquer possibilidade de impasse, basta inverter a ordem do bloqueio da thread de leitura e da thread de gravação, neste caso, linhas 11 e 12 ou linhas 19 e 20.

É importante atentar para o fato de que um deadlock dificilmente ocorre, pois, para que ele aconteça, é preciso que duas threads estejam compartilhando o tempo da CPU da mesma maneira. Além disso, o deadlock pode envolver mais de duas threads e mais de dois objetos sincronizados, uma situação ainda mais complexa.

17.11. Interação entre threads

A interação entre threads garante que elas estabeleçam comunicação entre si. Há três métodos que auxiliam nessa interação, são eles: **notify()**, **notifyAll()** e **wait()**, todos contidos na classe **java.lang.Object**. Esses métodos auxiliam as threads a conseguir informações a respeito de determinados eventos relevantes para as mesmas. Os métodos **wait()** e **notify()** fazem com que uma thread aguarde até que outra thread a notifique de que ela precisa retornar à execução.

É importante destacar que para uma thread poder chamar os métodos **wait()**, **notify()** e **notifyAll()** de um determinado objeto, é necessário que, primeiramente, ela tenha o bloqueio desse objeto.

Tendo em vista que **wait()** e **notify()** são métodos de instância da classe **Object**, é possível que haja uma lista de threads aguardando o recebimento de uma notificação de um objeto. Para que uma thread entre nessa lista, o método **wait()** do objeto de destino deve ser executado e, quando isso acontece, é preciso que o método **notify()** do objeto de destino seja chamado para que essa thread possa executar alguma outra instrução. Somente uma thread poderá continuar sua execução caso haja várias threads esperando em um mesmo objeto. Caso contrário, nenhuma ação ocorrerá.

Observe o código a seguir. Ele demonstra um objeto aguardando pela notificação de outro:

```
SegmentoY.java
1 public class SegmentoY extends Thread {
2
3     int montante;
4
5     public void run(){
6         synchronized(this){
7             for(int a = 0; a <= 10; a++){
8                 montante += a;
9             }
10            notify();
11        }
12    }
13 }
14

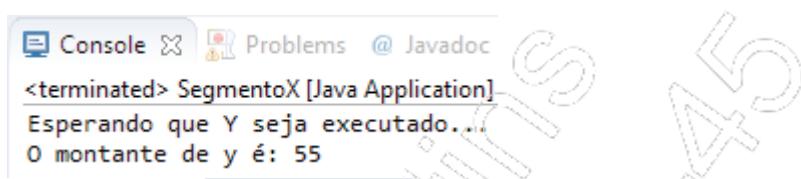
SegmentoX.java
1 public class SegmentoX {
2     public static void main(String args[]){
3
4         SegmentoY y = new SegmentoY();
5         y.start();
6
7         synchronized(y){
8             try{
9                 System.out.println("Esperando que Y seja executado...");
10                y.wait();
11            }catch(InterruptedException e){
12            }
13        }
14        System.out.println("O montante de y é: " + y.montante);
15    }
16 }
17 }
18 }
```

No código apresentado, encontramos a seguinte situação:

- O objeto **SegmentoX** possui a thread principal;
- O objeto **SegmentoY** possui a thread responsável por somar os números de 0 a 10;
- Na linha 6, o método `y.start()` será chamado, o que fará com que **SegmentoX** passe para a linha de código seguinte de sua classe;

- Na linha 11, é utilizado o método `y.wait()` para que **SegmentoX** não chegue até a linha 16 antes que **SegmentoY** possa finalizar seus cálculos;
- Na linha 8, o código e o objeto `y` são sincronizados. **SegmentoX** necessita do bloqueio desse objeto `y` para que o método `wait()` possa ser chamado no objeto;
- Nas linhas de 9 a 14, o método `wait()` está sendo encapsulado por um bloco `try-catch`.

Depois de compilado e executado o código anterior, o resultado será como o da imagem a seguir:



```
Console Problems @ Javadoc
<terminated> SegmentoX [Java Application]
Esperando que Y seja executado...
O montante de y é: 55
```

Outro código bastante comum é o seguinte:

```
synchronized(ObjetoA){ // tem o bloqueio em ObjetoA
    try{
        ObjetoA.wait(); // a thread libera o bloqueio e aguarda
                      // para continuar, a thread necessita do bloqueio
                      // então, ela deve ser paralisada até que consiga esse
                      // bloqueio
    }catch(InterruptedException e) { }
}
synchronized(this){
    notify();
}
```

Este código esperará até que o método `notify()` seja chamado no objeto `ObjetoA`. O código anteriormente descrito mostra que, enquanto a thread estiver esperando, ela libera o bloqueio para outras threads, mas, para que retorne à sua execução, ela necessita novamente desse bloqueio. O método `notify()` fará a notificação a qualquer thread que esteja aguardando no objeto `this`.

É importante ressaltar que somente uma thread que possui o bloqueio do objeto pode chamar os métodos **wait()** e **notify()**. Caso essa thread não possua o bloqueio, será lançada uma exceção **IllegalMonitorStateException**, a qual não precisa ser capturada de forma explícita, uma vez que não é verificada. Essa exceção, porém, deve ser manipulada, pois uma thread em espera, assim como uma thread no estado **suspended**, pode ser interrompida a qualquer momento. O código descrito a seguir mostra como fazer essa manipulação:

```
try {  
    wait();  
}catch(InterruptedException e){  
    // manipular a exceção  
}
```

Caso a thread não seja interrompida, ela pode continuar sua execução até que receba uma notificação, ou até a expiração do prazo de espera. Esse prazo máximo de espera, determinado pelo método **wait()**, também pode ser estabelecido em milissegundos, o que é feito da seguinte maneira:

```
synchronized(x){ // a thread obtém o bloqueio em x  
  
    x.wait(1000); // a thread libera o bloqueio e aguarda a notificação  
    // mas somente por um tempo máximo de um segundo e,  
    // então, volta para Runnable  
}
```

Uma thread libera prontamente seu bloqueio assim que o método **wait()** é chamado em um objeto, porém, isso não acontece necessariamente quando o método **notify()** é chamado. Neste caso, o bloqueio apenas será liberado no momento em que a thread tiver concluído o código sincronizado.

Já o método **notifyAll()** permite notificar todas as threads que aguardam em um determinado objeto. Ao utilizar esse método em um objeto, você permite que as threads saiam do estado de espera e retornem ao estado **ready to run**:

```
notifyAll(); // todas as threads em espera serão notificadas
```

Assim que todas as threads são notificadas, elas iniciam uma competição em busca do bloqueio. Dessa forma, as threads entram em ação sem que haja a necessidade de uma notificação adicional, uma vez que o bloqueio será utilizado e liberado por cada uma dessas threads.

Utilizar o método **notifyAll()** é importante principalmente nas situações em que você tem diversas threads aguardando em um objeto ou, ainda, quando deseja que a thread certa seja notificada. Com o método **notify()**, somente uma thread será notificada, impossibilitando assegurar que esta será a thread certa.

O quadro a seguir apresenta uma relação dos principais métodos utilizados na manipulação de threads:

Métodos definidos na classe Object	Método definido na interface Runnable	Métodos definidos na classe Thread	Métodos estáticos	Métodos não estáticos
notify() notifyAll() wait()	run()	join() sleep() start() yield()	sleep() yield()	join() start()

Os métodos **notify()**, **notifyAll()** e **wait()** são definidos na classe **Object**, e não na classe **Thread**, porque eles atuam sobre os monitores, e não sobre as threads. Os monitores, como você viu, referem-se a objetos que estão relacionados às instâncias de uma classe.

Em suma, quando o método **wait()** é chamado, a thread é suspensa e inserida na fila do monitor até que receba uma notificação. Para que essa notificação seja recebida por uma thread, é preciso que o método **notify()** seja chamado.

Para que essa notificação seja recebida por todas as threads, é preciso que o método **notifyAll()** seja chamado. Sendo assim, é preciso que esses métodos sejam chamados em uma thread que possua o monitor do objeto ao qual ela pertence. Vale ressaltar que, em razão disso, esses métodos devem ser invocados em blocos sincronizados, ou em outros métodos.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- **Thread** é uma classe em Java que permite a execução de diferentes tarefas simultaneamente. O termo **thread** diz respeito a linhas que compartilham um mesmo contexto e cuja execução ocorre de forma concomitante;
- O conceito de multiprocessamento é bastante importante para a compreensão do conceito de multithreaded. Multiprocessamento refere-se à utilização compartilhada da CPU por vários processos, os quais levam um determinado período de tempo para serem executados. Essa execução simultânea de vários processos ocorre em grande parte dos sistemas atuais, os quais se baseiam em micros contendo um único processador que realiza tarefas de forma concomitante;
- Quando você trabalha com a linguagem Java, a implementação do recurso multithreaded pode ser realizada de duas formas distintas: por meio da implementação da interface **Runnable** ou por meio da extensão da classe **Thread**;
- Instanciar uma classe **Thread** é o primeiro passo para criar uma thread de execução, embora ainda seja necessário um objeto **Thread** nas situações em que o método **run()** se encontra na classe de implementação da interface **Runnable** e nas ocasiões em que esse método se encontra em uma classe estendida de **Thread**;
- As threads, que são utilizadas na linguagem Java com a finalidade de manter a sincronia entre as execuções realizadas em um determinado ambiente, apresentam-se em um determinado estado;
- O scheduler é o responsável por definir quais threads serão executadas em um dado momento, além de fazer com que a thread saia do estado **ready to run**. Nas situações em que somente uma máquina realiza o processamento, é possível que apenas uma thread seja executada por vez, sendo que o scheduler é o responsável por determinar qual thread será processada;

- Na linguagem Java, são atribuídas prioridades às threads representadas por números inteiros que, em geral, ficam em um intervalo de 1 a 10, sendo que a prioridade padrão é 5. Essa prioridade determina como a thread será executada em relação às outras. É essencial ter em mente que esse número não determina se a execução de uma thread será mais rápida do que a execução de outra, mas sim o momento em que será realizado o chaveamento de contexto, o chamado **context switch**. Não há qualquer garantia de que as prioridades atribuídas sejam acatadas pelo Sistema Operacional da forma desejada;
- Os recursos compartilhados apenas podem ser utilizados por uma thread de cada vez, porém, no decorrer da execução de alguns programas, é possível que haja mais de uma thread necessitando de acesso a esses recursos. Para resolver essa situação, utilize a sincronização;
- Bloqueios e sincronização são dois conceitos relacionados entre si, uma vez que a sincronização funciona com os bloqueios;
- Um **deadlock** ocorre nas situações em que duas threads dependem uma da outra em um par de objetos que estão sincronizados. Em situações como esta, temos uma ou mais threads cuja execução está paralisada, esperando pelo acesso a um recurso que já foi alocado por outra thread;
- A interação entre threads garante que elas estabeleçam comunicação entre si. Há três métodos que auxiliam nessa interação, são eles: **notify()**, **notifyAll()** e **wait()**, todos contidos na classe **java.lang.Object**. Esses métodos auxiliam as threads a conseguir informações a respeito de determinados eventos relevantes para elas. Os métodos **wait()** e **notify()** fazem com que uma thread aguarde até que outra thread a notifique de que ela precisa retornar à execução.

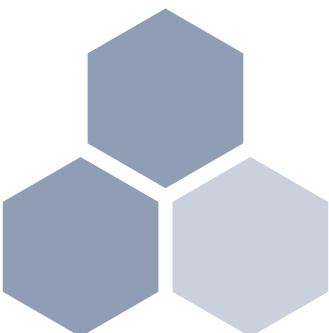


17

Threads

Teste seus conhecimentos

Willians Martins
305.009.778-45



1. Sobre as threads, qual alternativa está correta?

- a) A linguagem Java não possui suporte nativo às threads, contudo, elas podem ser implementadas por determinadas bibliotecas.
- b) É uma classe em Java que bloqueia a execução de diferentes tarefas simultaneamente.
- c) Uma thread representa um fluxo de controle em um processo.
- d) Todas as alternativas anteriores estão corretas.
- e) Nenhuma das alternativas anteriores está correta.

2. Qual das alternativas a seguir descreve o estado blocked de uma thread?

- a) A thread está nesse estado quando se encontra aguardando por um recurso.
- b) A thread está nesse estado quando seu código de execução a informou para que ficasse inativa durante um certo período de tempo.
- c) A thread está nesse estado nas situações em que seu método run() é finalizado, ou seja, nas situações em que a thread está inativa.
- d) Uma thread nesse estado está pronta para ser executada, mas ainda não foi selecionada pelo scheduler como sendo uma thread pronta para o processamento.
- e) Nenhuma das alternativas anteriores está correta.

3. Qual das alternativas a seguir descreve o estado ready to run de uma thread?

- a) A thread está nesse estado quando se encontra aguardando por um recurso.
- b) A thread está nesse estado quando seu código de execução a informou para que ficasse inativa durante certo período de tempo.
- c) A thread está nesse estado nas situações em que seu método run() é finalizado, ou seja, nas situações em que a thread está inativa.
- d) Uma thread nesse estado está pronta para ser executada, mas ainda não foi selecionada pelo scheduler como sendo uma thread pronta para o processamento.
- e) Nenhuma das alternativas anteriores está correta.

4. Qual das alternativas a seguir está INCORRETA em relação à prioridade de execução de threads?

- a) Uma thread pode desistir do controle de maneira espontânea.
- b) É possível que uma thread seja deslocada por outra cuja prioridade é mais alta.
- c) Há situações nas quais threads que possuem a mesma prioridade disputam os ciclos da CPU.
- d) A ordem de prioridade das threads sempre serão respeitadas pelo scheduler.
- e) Todas as alternativas anteriores estão corretas.

5. Qual é a função do método sleep()?

- a) Determinar se uma thread ainda está em execução.
- b) Permitir que a thread dentro da qual foi chamado fique no estado suspended durante um determinado período de tempo.
- c) Permitir que uma thread seja adicionada ao final de outra thread.
- d) Permitir que uma thread em estado running retorne ao estado ready to run para que outras threads com a mesma prioridade também possam ser processadas.
- e) Nenhuma das alternativas anteriores está correta.



17

Threads

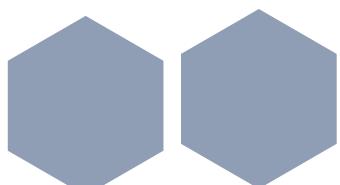


Mãos à obra!

Williams Martins
305.009.778-45



Editora
IMPACTA



Laboratório 1

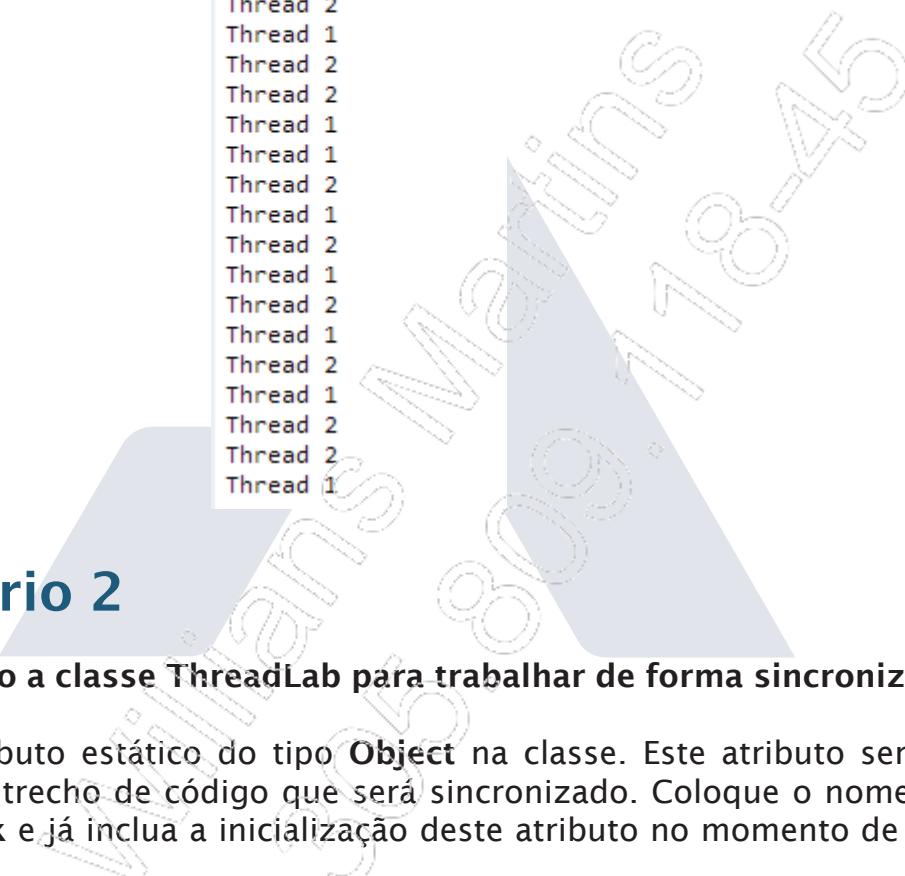
A – Criando a classe ThreadLab

1. Crie uma classe com o nome **ThreadLab** que estenda a classe **Thread**;
2. Crie um construtor que receba uma String como parâmetro e passe a String para o construtor da classe pai;
3. Implemente o método **run()** de forma que não retorne nada;
4. Dentro do método **run**, insira um **for** que rode 10 vezes;
5. Dentro do **for**, imprima na tela o método **getName()** da classe **Thread** e insira um bloco **try/catch**;
6. Dentro do **try**, chame o método **sleep** da classe **Thread** passando como parâmetro o valor **500**;
7. Insira a exceção **InterruptedException** no **catch** e imprima o erro na tela;
8. Salve e compile a classe.

B – Testando a classe ThreadLab

1. Crie uma classe com o nome **Cap17_Lab1** e insira a estrutura básica de um programa Java;
2. Dentro do método **main**, crie dois objetos de **ThreadLab** passando como parâmetro para o construtor **Thread 1** e **Thread 2**;
3. Chame o método **start()** dos dois objetos;
4. Compile e execute o programa.

O resultado deve ser como o exibido a seguir:



```
Console X Probl
<terminated> Cap17_Lab1
Thread 2
Thread 1
Thread 1
Thread 2
Thread 1
Thread 2
Thread 2
Thread 1
Thread 1
Thread 2
Thread 1
Thread 1
```

Laboratório 2

A – Modificando a classe ThreadLab para trabalhar de forma sincronizada

1. Crie um atributo estático do tipo **Object** na classe. Este atributo servirá de bloqueio ao trecho de código que será sincronizado. Coloque o nome do atributo de **lock** e já inclua a inicialização deste atributo no momento de sua declaração;
2. Envolva a estrutura do **for** com um bloco **synchronized(lock)**;

3. Execute o programa:

```
Console X Prob
<terminated> Cap17_Lab1
Thread 1
```

Dessa vez o acesso foi sincronizado entre as threads. Enquanto uma thread não terminou sua execução, a outra não pode obter o **lock** do objeto para executar. Isso só ocorreu após o término da execução de uma das threads.

Observação 1: A execução pode iniciar por qualquer uma das threads: 1 ou 2;

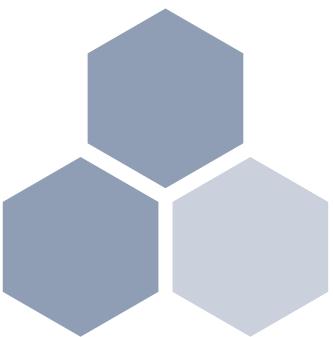
Observação 2: É importante observar que, para a sincronização da execução das threads, é necessário que o objeto de bloqueio tenha o mesmo valor para as duas ou mais threads envolvidas na execução. Por isso que foi utilizada uma instância estática da classe, o que garante que o mesmo endereço de memória é utilizado como bloqueio tanto para a thread 1 quanto pra a thread 2.



18

Geração de Pacotes - Instalação de Aplicações Java (JAR)

- ◆ Conceito de aplicações e bibliotecas;
- ◆ Geração de bibliotecas e executáveis.



18.1. Conceito de aplicações e bibliotecas

Quando necessitamos de uma biblioteca externa de apoio a uma aplicação Java ou precisamos gerar um arquivo dito “executável” da nossa aplicação para instalarmos em algum cliente, utilizamos arquivos no formato **JAR** (Java Archive).

O formato JAR permite que sejam agrupados múltiplos arquivos em um único arquivo. Normalmente, o arquivo JAR contém arquivos que representam classes e recursos auxiliares às aplicações (ex.: arquivos de configuração ou imagens).

Arquivos JAR são gerados no formato ZIP, portanto os arquivos podem ser empacotados e desempacotados utilizando os softwares de compressão tradicionais.

De qualquer forma, para realizar tarefas básicas relacionadas ao arquivo JAR, utilize o **Java Archive Tool** disponibilizado no JDK, que também é integrado às principais IDEs Java do mercado.

A seguir, serão apresentadas a geração de um arquivo Java executável e sua posterior utilização como aplicativo Java (e, num segundo momento, como uma biblioteca Java).

18.2. Geração de bibliotecas e executáveis

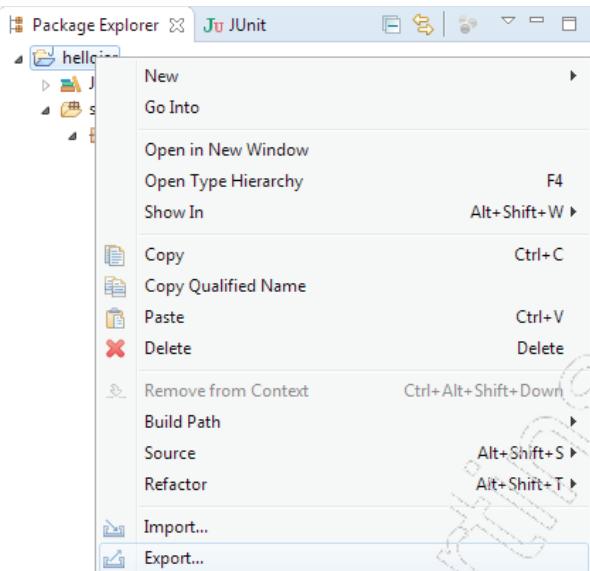
Para esta demonstração, considere o projeto **hellojar**, contendo uma classe principal que apenas exibe uma janela quando é executada:



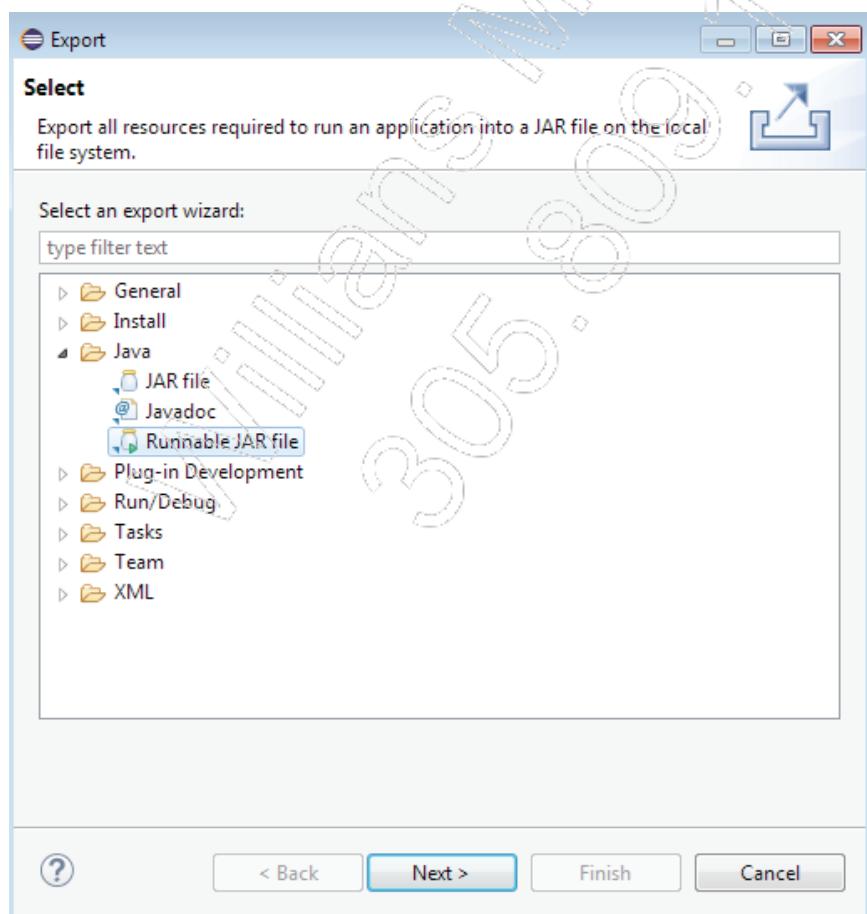
```
1 package hellojar;
2
3 import java.awt.Dimension;
4 import javax.swing.JFrame;
5
6
7 public class HelloFrame extends JFrame{
8
9     public static void main(String[] args) {
10         JFrame frame = new JFrame("Hello JAR !");
11         frame.setSize(new Dimension(400,200));
12         frame.setDefaultCloseOperation(EXIT_ON_CLOSE); //encerra a aplicação ao fechar a janela
13         frame.setLocationRelativeTo(null); //centraliza o frame
14         frame.setVisible(true);
15     }
16 }
17
```

18.2.1. Geração de um pacote executável

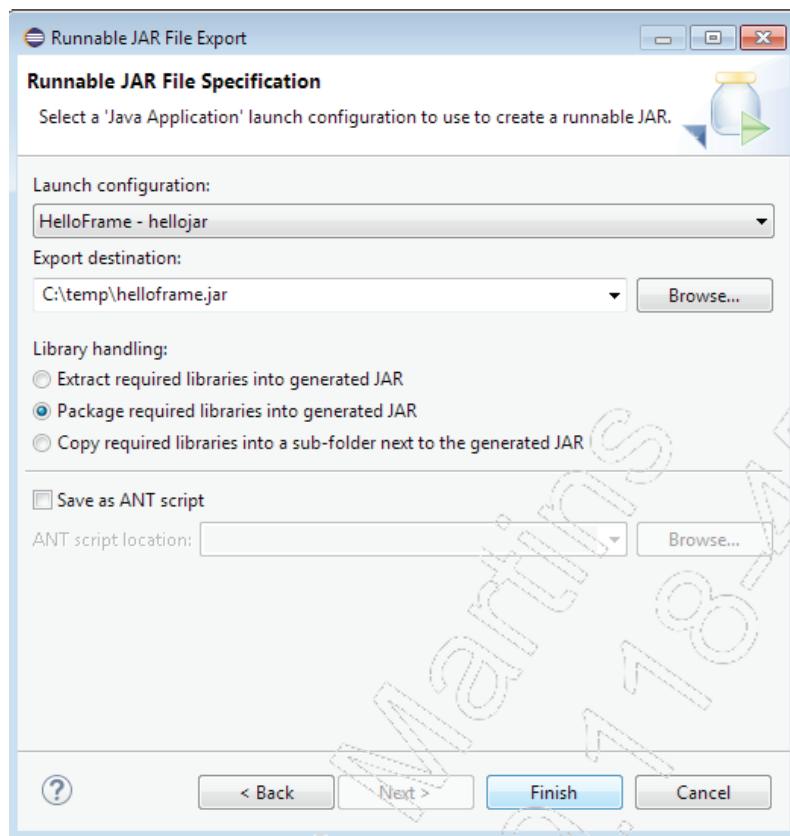
A partir do Eclipse, selecione a opção de exportação de projeto:



Escolha a opção **Runnable JAR file**:

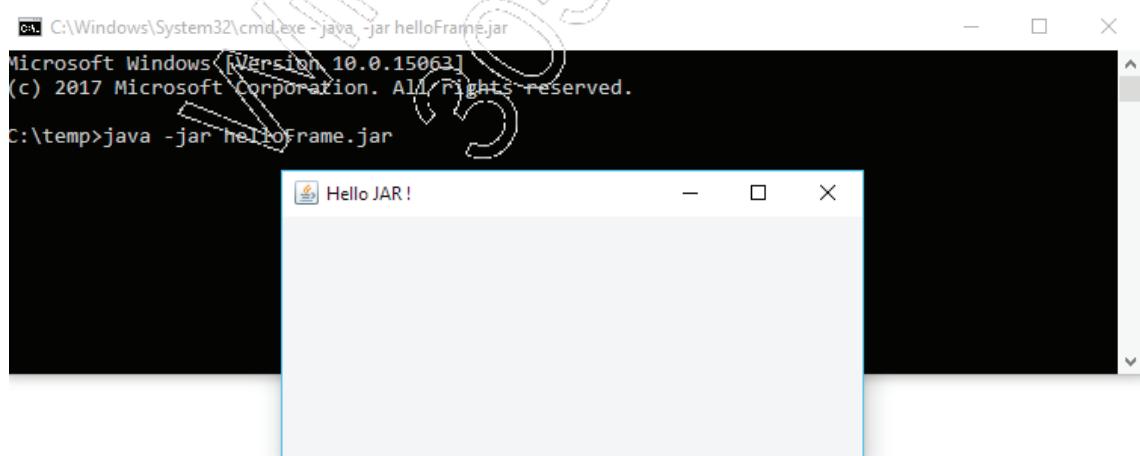


Selecione, dentre as opções, a classe principal do pacote que está sendo gerado:



Para execução da aplicação, dê um duplo clique no arquivo gerado (dependendo da configuração do sistema operacional) ou acesse o prompt de comando e execute o seguinte comando:

```
java -jar [nome-arquivo].jar
```

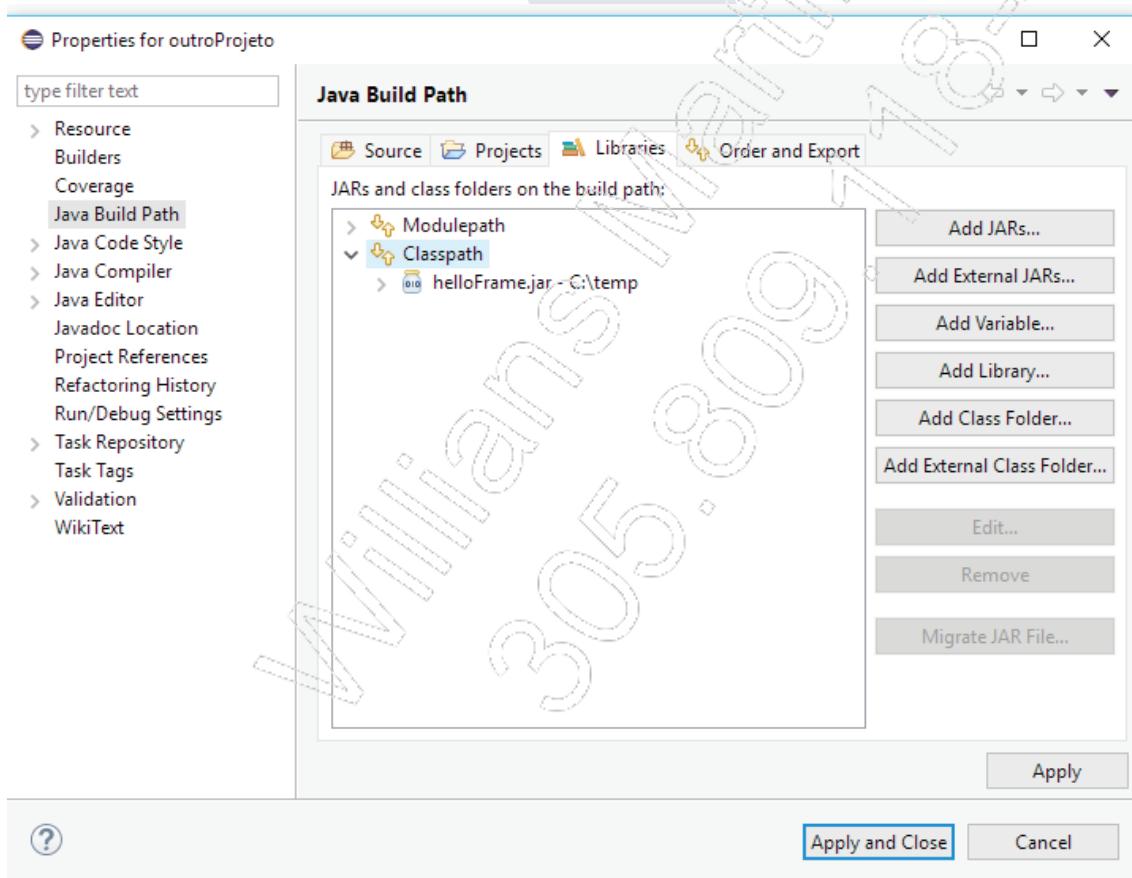


18.2.2. Utilização de uma biblioteca em projetos

Para utilizar este mesmo pacote gerado como biblioteca de um outro projeto, o procedimento também é bem simples, utilizando a IDE.

A partir do projeto onde se pretende utilizar uma biblioteca Java externa, basta colocar sua referência no classpath do projeto, que é onde são buscadas dinamicamente pela JVM classes que não pertençam ao projeto e que são referenciadas em classes Java. Para isso, realize os passos a seguir:

1. Clique com o botão direito no projeto e selecione a opção **Properties / Java Build Path / Libraries / Classpath**;
2. Selecione a opção **Add External JARs...**;
3. Selecione a biblioteca desejada conforme a figura adiante:



A partir daí, o projeto atual pode referenciar e importar qualquer classe contida na biblioteca.

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

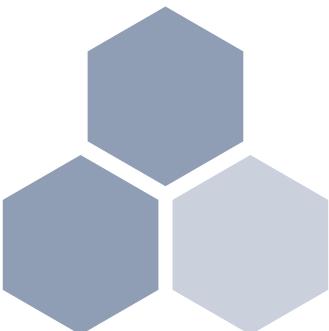
- Quando necessitamos de uma biblioteca externa de apoio a uma aplicação Java ou precisamos gerar um arquivo dito “executável” da nossa aplicação para instalarmos em algum cliente, utilizamos arquivos no formato **JAR** (Java Archive);
- Para realizar tarefas básicas relacionadas ao arquivo JAR, utilize o **Java Archive Tool** disponibilizado no JDK, que também é integrado às principais IDEs;
- Um JAR pode ser utilizado como um executável Java ou uma biblioteca para outros projetos.



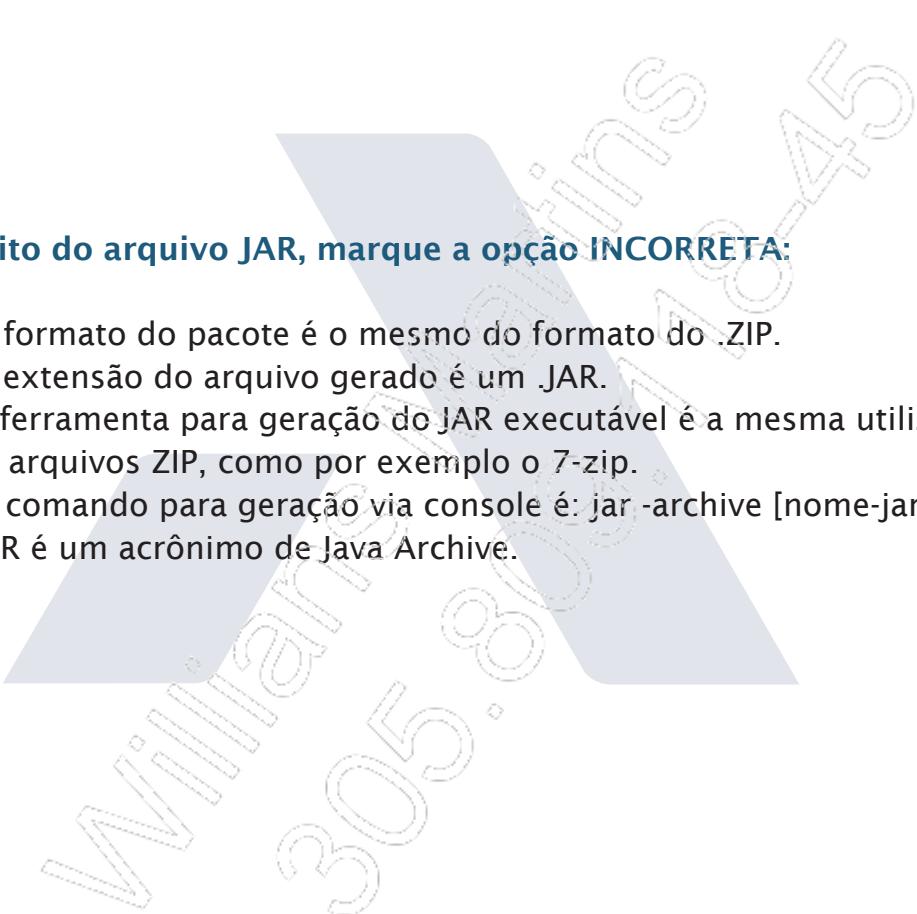
18

Geração de Pacotes - Instalação de Aplicações Java (JAR)

Teste seus conhecimentos



1. A respeito do arquivo JAR, marque a opção INCORRETA:

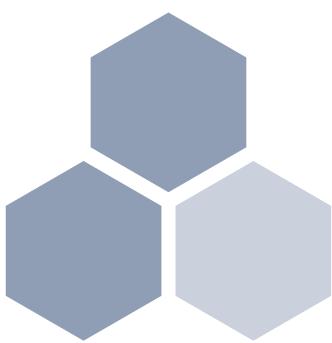
- a) O formato do pacote é o mesmo do formato do .ZIP.
 - b) A extensão do arquivo gerado é um .JAR.
 - c) A ferramenta para geração do JAR executável é a mesma utilizada para arquivos ZIP, como por exemplo o 7-zip.
 - d) O comando para geração via console é: jar -archive [nome-jar].
 - e) JAR é um acrônimo de Java Archive.
- 



19

Banco de dados com Java - JDBC

- ◆ Pacote `java.sql`;
- ◆ Conexões com banco de dados;
- ◆ Operações na base de dados;
- ◆ Operações parametrizadas;
- ◆ Transações;
- ◆ Consultas.



19.1. Introdução

O **JDBC** (Java Data Base Connectivity) é uma especificação elaborada no Java para prover a acessibilidade a aplicações com bancos de dados por esta linguagem, cuja primeira versão foi criada em 1997.

Esta API proporciona um acesso aos bancos de dados de forma fácil e simples, especialmente acesso a dados armazenados em um banco de dados relacional. Trata-se de um padrão de acesso a dados obedecido pela indústria de bancos de dados.

A fim de seguir este padrão, os fabricantes devem distribuir drivers JDBC aos desenvolvedores Java.

Com JDBC, você pode escrever aplicativos Java para gerenciar estas três atividades de programação:

- Conexão a uma fonte de dados, como um banco de dados;
- Envio de consultas e instruções de atualização para esse banco de dados;
- Recuperação e processamento de resultados recebidos do banco de dados em resposta à consulta.

A API JDBC inclui:

- O pacote **java.sql**, referido como o núcleo JDBC;
- O pacote **javax.sql**, referido como o pacote opcional da API JDBC, o qual estende a funcionalidade da API JDBC a partir de uma API do lado do cliente (client-side) para uma API do lado do servidor (server-side), e é uma parte essencial da tecnologia de Java Enterprise Edition (Java EE).

Chamamos de driver um componente de software, geralmente desenvolvido pelo próprio fabricante de seu banco de dados, que permite a comunicação com este banco de dados através de instruções Java/JDBC.

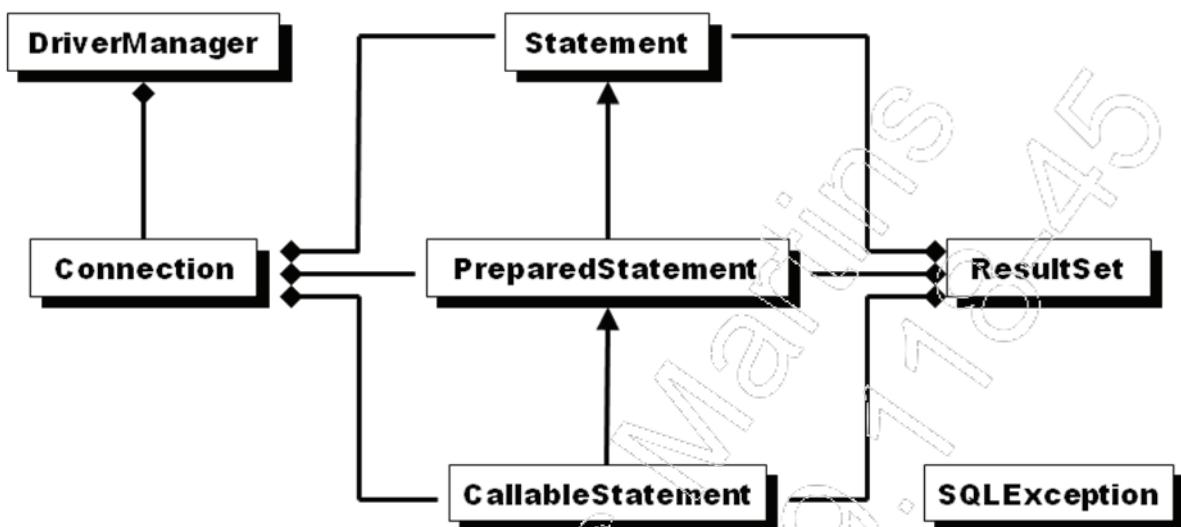
Em geral, um driver é composto por um ou mais arquivos .JAR, que devem ser incorporados à sua aplicação (ao classpath do Java) para garantir o acesso àquela base de dados.

Uma vez que muitos dos novos recursos são opcionais e, consequentemente, há alguma variação em drivers e as características que eles suportam, você deve sempre verificar a documentação do driver para ver se ele suporta um recurso antes de tentar usá-lo.

19.2. Pacote java.sql

A API JDBC foi configurada para que você possa executar instruções SQL em um banco de dados, bem como acessar e executar stored procedures e outras rotinas programáticas.

Para isso, utilize um pacote chamado **java.sql**. Com ele, você pode acessar e, também, processar os dados que estão armazenados em uma fonte de dados, bem como abrir tabelas, além de realizar diversas operações, como inclusão, alteração, exclusão de dados e consultas aos dados de uma tabela.



O acesso às diferentes fontes de dados é realizado por meio de drivers distintos, instalados de forma dinâmica, e que estão incluídos na framework da API.

19.3. Conexões com banco de dados

Toda e qualquer operação a ser realizada na base de dados se dá através de uma conexão. A conexão com o banco de dados é representada pela interface **Connection**. Quando as instruções SQL são executadas, o resultado é retornado no contexto de uma conexão.

19.3.1. Estabelecendo uma conexão

A comunicação com o banco de dados é realizada a partir de um driver, que é um conjunto de classes Java. Por meio dos drivers, você pode realizar tarefas como a criação de conexões e tabelas, a chamada de **Stored Procedures** e a execução de comandos SQL.

Antes de efetuar a conexão com o banco de dados, é necessário que o driver JDBC do banco de dados utilizado seja colocado no **classpath**.

Com o framework Java SQL, é possível ter diversos drivers de banco de dados. Todos os drivers possíveis são carregados por **DriverManager**, o qual, na sequência, solicita que cada driver tente estabelecer a conexão com a URL de destino.

Para cada banco de dados existe um driver correspondente, o qual é fornecido pelo próprio fabricante do banco de dados.

Para estabelecer uma conexão com o banco de dados de sua preferência, utilize a seguinte sintaxe para obter um objeto de conexão com o banco de dados:

```
Connection con = DriverManager.getConnection(  
    "url", "myLogin", "myPassword");
```

Nessa sintaxe, a classe **DriverManager**, que será estudada mais adiante, é a responsável pela conexão com o banco de dados.

Assim como a maioria dos métodos do JDBC, o método **getConnection()** da classe **DriverManager** requer o tratamento da exceção **SQLException** que pode ocorrer na aplicação em caso de falha de conexão.

De uma forma geral, colocamos todas as instruções de acesso dentro do bloco **try**, e o seu respectivo **catch** possui tratamento para esta exceção:

```
try {  
    Connection con = DriverManager.getConnection(  
        "url", "myLogin", "myPassword");  
  
    // realiza as operações com a conexão  
  
} catch (SQLException e) {  
    // realiza o tratamento em caso de falha  
}
```

19.3.2. Interface Connection

A conexão com o banco de dados é representada pela interface **Connection**. Quando as instruções SQL são executadas, o resultado é retornado no contexto de uma conexão. Ao utilizar o método **getMetaData**, diversas informações são fornecidas a partir do objeto **Connection** do banco de dados. Dentre essas informações, estão aquelas que descrevem a tabela do banco de dados, além das que citam as capacidades da conexão e as stored procedures utilizadas.

A sintaxe de **Connection** é a seguinte:

```
public interface Connection  
extends Wrapper, AutoCloseable
```

Após cada instrução ser executada em um banco de dados, as alterações são confirmadas. Isso ocorre porque o objeto **Connection** está configurado com o modo de confirmação automática, o qual pode ser desabilitado. Nesse caso, uma vez desabilitado o modo automático, torna-se necessário chamar o método **commit** explicitamente para que as alterações realizadas no banco de dados sejam confirmadas.

Para criar um novo objeto **Connection**, a forma mais comum é utilizar o método **getConnection()**, estático, na classe **DriverManager**.

19.3.3. Classe DriverManager

Para gerenciar um conjunto de drivers JDBC, utilize o **DriverManager**, que é uma classe de Java. A conexão com a fonte de dados é realizada a partir da interface **DataSource**, sendo que o principal meio de conexão utilizado é com o objeto **DataSource**.

A sintaxe desta classe é a seguinte:

```
public class DriverManager  
extends Object
```

As classes de driver indicadas na propriedade **jdbc.drivers** do sistema podem ser carregadas na inicialização do **DriverManager** ou explicitamente. Dessa forma, os drivers JDBC que serão utilizados nas aplicações podem ser configurados de acordo com a preferência do usuário.

Quando o método **getConnection** for chamado, o **DriverManager** fará uma pesquisa entre todos os drivers carregados tanto na sua inicialização quanto explicitamente, em busca do driver adequado. Para tanto, o mesmo classloader da aplicação ou do applet atual será utilizado.

O exemplo a seguir ilustra a utilização de **DriverManager**:

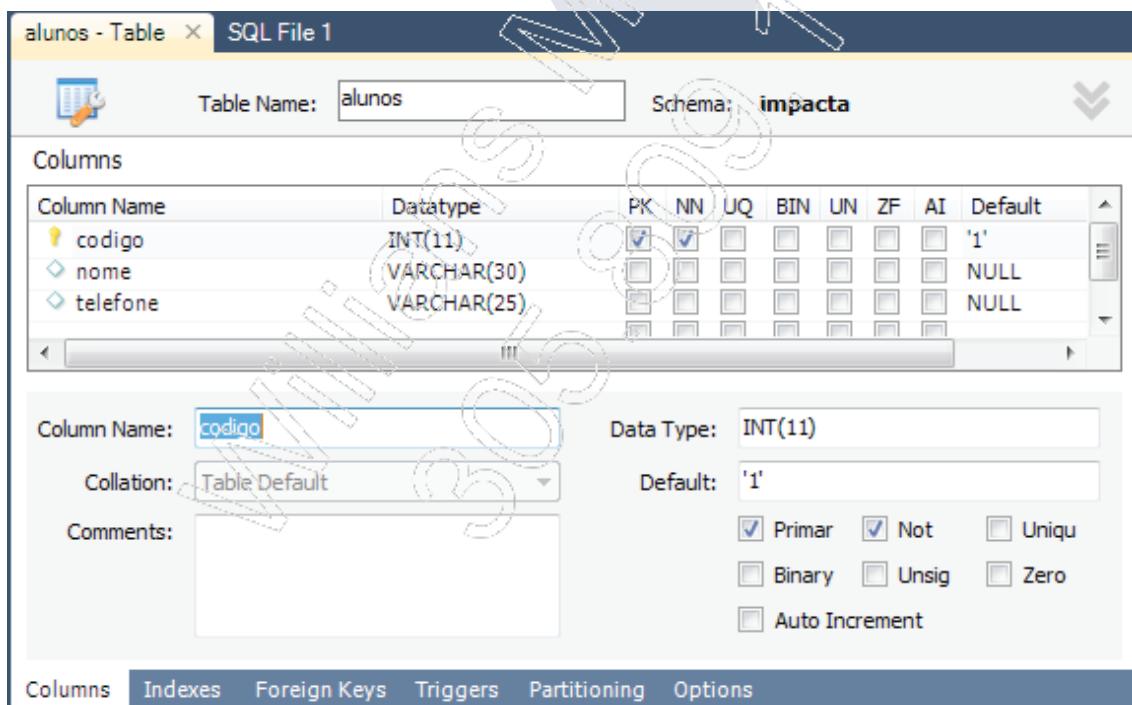
```
Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/impacta", "root", "impacta");
```

19.3.4. Estabelecendo a conexão com o banco de dados

No exemplo adiante, mostramos como estabelecer uma conexão com o banco de dados.

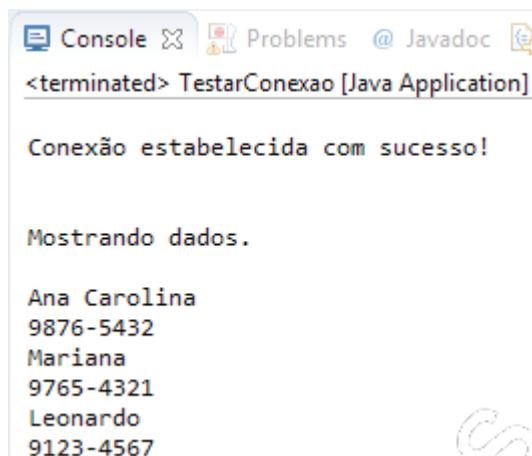
Antes de executar o exemplo, é necessário criar um banco de dados e uma tabela. Para isso, execute o seguinte procedimento:

1. No **MySQL Workbench**, crie o banco de dados **impacta**;
2. Em seguida, crie uma tabela com as seguintes características:
 - Nome da tabela: **alunos**;
 - Campo: **codigo**, tipo **INT**, Primary Key (PK), Not Null, default 1;
 - Campo: **nome**, tipo **VARCHAR(30)**;
 - Campo: **telefone**, tipo **VARCHAR(25)**.



```
J TestarConexao.java X
1 import java.sql.*;
2
3 public class TestarConexao {
4
5     private final static String url = "jdbc:mysql://localhost:3306/impacta";
6
7     private final static String username = "root";
8     private final static String password = "impacta";
9
10    private Connection con;
11    private Statement stmt;
12    private ResultSet rs;
13
14    private String nome = null;
15    private String telefone = null;
16
17    public static void main(String args[]){
18        TestarConexao b = new TestarConexao();
19        b.openDB();
20        b.mostra();
21        b.closeDB();
22    }
23
24    public void openDB(){
25        try{
26            con = DriverManager.getConnection(url, username, password);
27            stmt = con.createStatement();
28            System.out.println("\nConexão estabelecida com sucesso!\n");
29        }catch(SQLException e){
30            System.out.println("\nNão foi possível estabelecer conexão " + e + "\n");
31            System.exit(1);
32        }
33    }
34
35    public void closeDB(){
36        try{
37            con.close();
38        }catch(SQLException e){
39            System.out.println("\nNão foi possível fechar conexão " + e + "\n");
40            System.exit(1);
41        }
42    }
43
44    public void setNome(String nome){
45        if(nome.trim().length() == 0)
46            this.nome = null;
47        else
48            this.nome = nome;
49    }
50
51    public void setTelefone(String telefone){
52        if(telefone.trim().length() == 0)
53            this.telefone = null;
54        else
55            this.telefone = telefone;
56    }
57
58    public String getName(){
59        return nome;
60    }
61
62    public String getTelefone(){
63        return telefone;
64    }
65
66    public void mostra(){
67        String query;
68
69        try{
70            query = "SELECT * FROM alunos";
71            System.out.println("\nMostrando dados.\n");
72
73            rs = stmt.executeQuery(query);
74            while(rs.next()){
75                System.out.println(rs.getString("nome"));
76                System.out.println(rs.getString("telefone"));
77            }
78        }catch(SQLException e){
79            setNome(null);
80            setTelefone(null);
81        }
82    }
83 }
```

Depois de compilado e executado o código anterior, o resultado será similar ao da figura a seguir:



The screenshot shows a Java application window titled "Console". The title bar also includes "Problems", "@ Javadoc", and a help icon. Below the title bar, it says "<terminated> TestarConexao [Java Application]". The main content area displays the following text:
Conexão estabelecida com sucesso!

Mostrando dados.

Ana Carolina
9876-5432
Mariana
9765-4321
Leonardo
9123-4567

19.3.5. Método Close

Sempre que abrir uma conexão com o banco de dados, você deve fechá-la, pois uma conexão não pode permanecer aberta por um longo período. Para fechar essa conexão, utilize um método **close()** da classe **Connection**.

O método **close()** pode lançar erros e, por esse motivo, o bloco **try/catch** deve ser utilizado para fechar a conexão com o banco de dados. Com relação à chamada a esse método, recomendamos que ela seja colocada dentro do bloco **finally**.

Veja o trecho de código a seguir:

```
public void closeDB(){
    try{
        con.close();
    }catch(SQLException e){
        System.out.println("\nNão foi possível fechar conexão " + e + "\n");
        System.exit(1);
    }
}
```

A partir do Java 7, é possível utilizar o **try-with-resources** (try com recursos) para inicializar os objetos **Connection**, **Statement** e **ResultSet**. Para tanto, basta declará-los dentro do comando **try**, que se encarrega de fechar os recursos automaticamente ao final do bloco. Veja um exemplo de como esse recurso pode ser usado com conexões de banco de dados:

```
public void openDB() {
    try (Connection con = DriverManager.getConnection(url, user, password);
        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery("Select * from alunos")) {
        System.out.println("\nConexão estabelecida com sucesso!\n");

    } catch (SQLException e) {
        System.out.println("\nNão foi possível estabelecer conexão" + e
            + "\n");
        System.exit(1);
    }
}
```

19.4. Operações na base de dados

Com a conexão aberta, podemos realizar operações na base de dados, como inclusão, exclusão, alteração de dados, entre outras.

Cada operação na base de dados é definida por uma instrução na linguagem SQL, chamadas de **statements**. São exemplos de statements:

```
UPDATE tab_funcionario SET salario = 5000 WHERE matr = 34

INSERT INTO tab_setor (codigo, nome) VALUES (34, 'RH')

DELETE FROM tab_produto WHERE codigo = 4983
```

Para realizar tais operações na base de dados, contamos com três diferentes interfaces. Cada qual com sua característica:

Interface	Descrição
Statement	Utilizada na realização de operações mais simples que possuam valores fixos.
PreparedStatement	Utilizada na realização de operações parametrizadas, em que, a cada execução, novos valores são operados pelo statement.
CallableStatement	Utilizado na chamada/execução de stored procedures.

Observe um exemplo de utilização da interface **Statement**:

```
J ExemploStatement.java X
1 import java.sql.*;
2
3 public class ExemploStatement {
4
5     public static void main(String[] args) {
6
7         try {
8
9             Connection cn = DriverManager.getConnection(
10                 "jdbc:mysql://localhost:3306/impacta", "root", "impacta");
11
12             Statement st = cn.createStatement();
13
14             st.executeUpdate("UPDATE tab_func SET salario = salario + 200");
15             System.out.println("Salários aumentados com sucesso.");
16
17             st.close();
18             cn.close();
19             System.out.println("Conexão encerrada.");
20
21         } catch (SQLException e) {
22             System.out.println("Falha ao realizar a operação.");
23             e.printStackTrace();
24         }
25     }
26 }
27
```

Observe que, a partir da conexão que já está aberta, executamos o método **createStatement()** para obter um statement, a partir do qual realizamos a operação desejada.

! Utilizamos o método **executeUpdate()** da interface **Statement** para realizar operações de manipulação (que não retornam dados), como INSERT, DELETE, CREATE TABLE, UPDATE etc. Operações de consulta (como SELECT) serão vistas adiante.

19.5. Operações parametrizadas

Uma instrução SQL parametrizada é um statement em que alguns valores são dinamicamente assinalados pela aplicação.

Para criar statements parametrizados, utilizamos a interface **PreparedStatement**.

A principal vantagem deste tipo de statement é que sua instrução pode ser executada diversas vezes, cada hora com valores diferentes.

Um **PreparedStatement** é criado a partir do método **prepareStatement()**, executado sobre a conexão aberta, onde passamos a instrução SQL contendo interrogações (?) para cada valor dinâmico a ser preenchido:

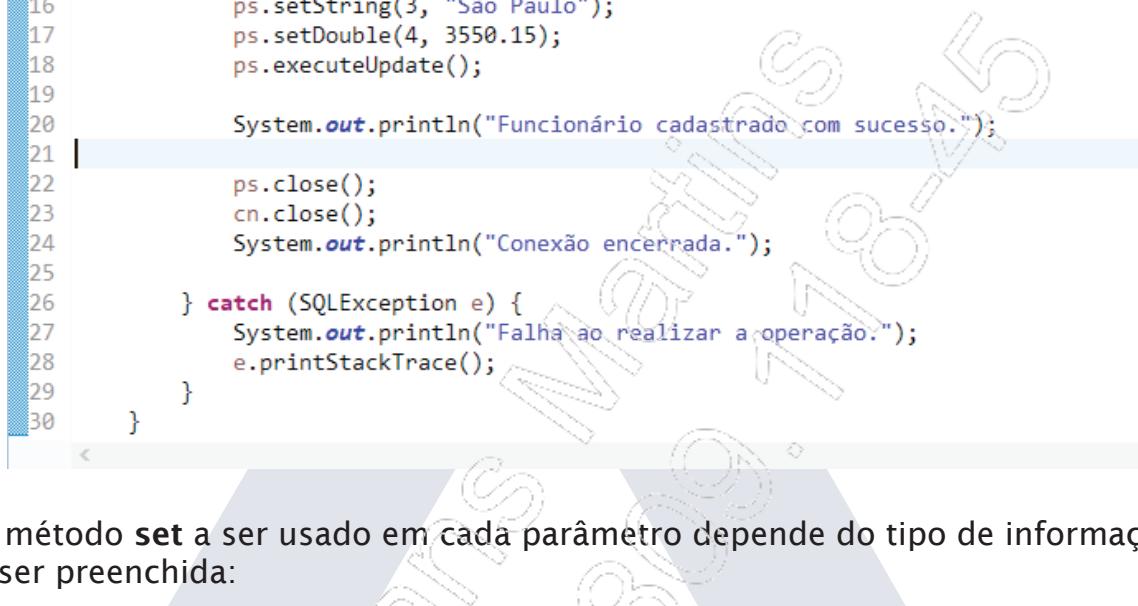
```
PreparedStatement ps = cn.prepareStatement(  
    "INSERT INTO tab VALUES (?, ?, ?, ?, ?, ?, ?)");
```

Cada interrogação representa um parâmetro. Antes de ser efetivamente executado, o **PreparedStatement** precisa ter os seus parâmetros preenchidos. Isto deve ser realizado através de seus métodos **set**, conforme o tipo de cada campo:

```
ps.setXXX(< INDICE DO PARAMETRO >, < VALOR >);
```

O índice do parâmetro (índice da interrogação) é de base **1**. No exemplo a seguir, estamos considerando um **INSERT** com quatro parâmetros:

- **Matrícula**: Numérico inteiro;
- **Nome**: Texto;
- **Cargo**: Texto;
- **Salário**: Numérico com parte fracionária.



```
ExemploPreparedStatement.java
3 public class ExemploPreparedStatement {
4
5     public static void main(String[] args) {
6
7         try {
8             Connection cn = DriverManager.getConnection(
9                 "jdbc:mysql://localhost:3306/impacta", "root", "impacta");
10
11            PreparedStatement ps = cn.prepareStatement(
12                "INSERT INTO tab_func VALUES (?, ?, ?, ?, ?)");
13
14            ps.setInt(1, 108);
15            ps.setString(2, "Manuel da Silva");
16            ps.setString(3, "São Paulo");
17            ps.setDouble(4, 3550.15);
18            ps.executeUpdate();
19
20            System.out.println("Funcionário cadastrado com sucesso.");
21        } catch (SQLException e) {
22            System.out.println("Falha ao realizar a operação.");
23            e.printStackTrace();
24        }
25    }
26}
```

O método **set** a ser usado em cada parâmetro depende do tipo de informação a ser preenchida:

Tipo SQL	Método set	Tipo Java
CHAR, VARCHAR	setString()	java.lang.String
INT, DECIMAL, NUMERIC (sem parte fracionária)	setInt()	int
DOUBLE, DECIMAL, NUMERIC (podendo haver parte fracionária)	setDouble()	double
BOOLEAN	setBoolean()	boolean
DATE	setDate()	java.sql.Date
TIME	setTime()	java.sql.Time
DATETIME, TIMESTAMP	setTimestamp()	java.sql.Timestamp
BLOB, MEDIUMBLOB, LONGBLOB	setBinaryStream()	java.io.InputStream
NULL	setNull()	

19.6. Transações

Chamamos de transação um conjunto de operações atômicas realizadas na base de dados e que podem ser desfeitas em situações de falha ou outro problema.

Quando corretamente utilizada, uma transação garante a integridade dos dados contidos na base.

Chamamos de **rollback** o comando utilizado para desfazer as operações retidas pela transação e de **commit** o comando utilizado para efetivá-las na base de dados.

O JDBC permite a criação de aplicações Java que manipulam transações com bancos de dados que oferecem suporte a este tipo de recurso. Para isso, contamos com os seguintes métodos da interface `java.sql.Connection`:

- `setAutoCommit(boolean)`;
- `commit()`;
- `rollback()`.

```
ExemploTransacao.java
1 import java.sql.*;
2
3 public class ExemploTransacao {
4
5     public static void main(String[] args) {
6         try {
7             realizaOperacao();
8         } catch (Exception e) {
9             e.printStackTrace();
10        }
11    }
12
13    private static void realizaOperacao() throws Exception {
14        Connection cn = null;
15        Statement st = null;
16
17        try {
18            cn = DriverManager.getConnection(
19                "jdbc:mysql://localhost:3306/impacta", "root", "impacta");
20            cn.setAutoCommit(false);
21
22            st = cn.createStatement();
23            st.executeUpdate("INSERT INTO tab (...) VALUES (...)");
24            st.executeUpdate("DELETE FROM tab WHERE ...");
25            st.executeUpdate("UPDATE tab SET ....");
26
27            cn.commit();
28        } catch (SQLException e) {
29            cn.rollback();
30            throw new Exception("Falha ao acessar base de dados.", e);
31        } finally {
32            st.close();
33        }
34    }
35}
```

19.7. Consultas

As consultas na base de dados são realizadas mediante o comando **SELECT**. Através dele, podemos obter os dados contidos em uma ou mais tabelas, seguindo critérios, agrupamentos e/ou ordenações, conforme necessidade da aplicação.

Para capturarmos os dados provenientes do comando **SELECT**, o JDBC conta com a interface **java.sql.ResultSet**.

Um **ResultSet** representa um cursor proveniente da base de dados. Trata-se de um conjunto de dados em memória de forma tabular que possui um ponteiro apontando para uma de suas linhas, a qual é chamada de registro atual.

Para obter um **ResultSet**, executamos o método **executeUpdate()** sobre um **Statement** (consulta fixa) ou um **PreparedStatement** (consulta dinâmica):

```
ResultSet rs = st.executeQuery("SELECT ...");
```

Após executar a consulta através deste comando, deve-se mover o cursor para a primeira linha do **ResultSet** e, se esta linha existir, podemos coletar os seus dados e avançar para a próxima linha.

Para avançar o cursor para a próxima linha, utilize o método **next()**. Além de forçar o cursor a avançar, este método retorna um booleano informando se a próxima linha existe ou não:

```
boolean existe = rs.next();
```

Tendo avançado para uma linha válida, podemos então coletar os dados de algum campo daquele registro através de um método **get**, conforme o tipo do campo:

```
<TIPO DA INFORMAÇÃO> variavel = rs.getXXX(<NOME DO CAMPO>);
```

No exemplo a seguir, considere que a tabela **TAB_FUNC** possua os campos **NOME** e **SALARIO**, respectivamente dos tipos **varchar** e **number** (com parte fracionária):

```
J ExemploResultSet.java X
1 import java.sql.*;
2
3 public class ExemploResultSet {
4
5     public static void main(String[] args) {
6
7         try {
8
9             Connection cn = DriverManager.getConnection(
10                 "jdbc:mysql://localhost:3306/impacta", "root", "impacta");
11             Statement st = cn.createStatement();
12             ResultSet rs = st.executeQuery("SELECT nome, salario FROM tab_func");
13
14             while (rs.next()) {
15                 String nome = rs.getString("nome");
16                 double sal = rs.getDouble("salario");
17                 System.out.println(nome + " - " + sal);
18             }
19
20             rs.close();
21             st.close();
22             cn.close();
23         } catch (SQLException e) {
24             System.out.println("Falha ao realizar a operação.");
25             e.printStackTrace();
26         }
27     }
28 }
```

Pontos principais

Atente para os tópicos a seguir. Eles devem ser estudados com muita atenção, pois representam os pontos mais importantes do capítulo.

- Para executar instruções SQL, é necessário ter a API **JDBC**, cuja primeira versão foi criada em 1997. Esta API proporciona um acesso a banco de dados fácil e simples. Isto quer dizer que ela permite acessar qualquer tipo de dados tabulares, especialmente dados armazenados em um banco de dados relacional;
- Para executar instruções SQL em um banco de dados, utilize um pacote chamado **java.sql**. Com ele, você pode acessar e, também, processar os dados que estão armazenados em uma fonte de dados, bem como abrir tabelas, além de realizar diversas operações, como inclusão, alteração, exclusão de dados e consultas aos dados de uma tabela;
- Toda e qualquer operação a ser realizada na base de dados se dá através de uma conexão. A conexão com o banco de dados é representada pela interface **Connection** e a comunicação com o banco de dados é realizada a partir de um driver, que é um conjunto de classes Java;
- Com a conexão aberta, podemos realizar operações na base de dados, como inclusão, exclusão, alteração de dados, entre outras. Cada operação na base de dados é definida por uma instrução na linguagem SQL, chamada de **statements**. Para realizar tais operações na base de dados, contamos com três diferentes interfaces: **Statement**, **PreparedStatement** e **CallableStatement**;
- Uma instrução SQL parametrizada é um statement em que alguns valores são dinamicamente assinalados pela aplicação. Para criar statements parametrizados, utilizamos a interface **PreparedStatement**. A principal vantagem deste tipo de statement é que sua instrução pode ser executada diversas vezes, cada hora com valores diferentes;
- Chamamos de transação um conjunto de operações atômicas realizadas na base de dados e que podem ser desfeitas em situações de falha ou outro problema. Chamamos de **rollback** o comando utilizado para desfazer as operações retidas pela transação e de **commit** o comando utilizado para efetivá-las na base de dados;
- As consultas na base de dados são realizadas mediante o comando **SELECT**. Para capturarmos os dados provenientes do comando SELECT, o JDBC conta com a interface **java.sql.ResultSet**.



19

Banco de dados com Java - JDBC

Teste seus conhecimentos



1. Qual método fecha uma conexão com um banco de dados?

- a) close()
- b) stop()
- c) break()
- d) dbend()
- e) Nenhuma das alternativas anteriores está correta.

2. Fazem parte das classes e interfaces que compõem o JDBC, EXCETO:

- a) Path
- b) DriverManager
- c) Connection
- d) ResultSet
- e) CallableStatement

3. Qual das seguintes atividades de programação pode ser gerenciada com o JDBC?

- a) Conexão a uma fonte de dados.
- b) Envio de consultas e instruções de atualização para a fonte de dados.
- c) Recuperação e processamento de resultados recebidos da fonte de dados.
- d) Todas as alternativas anteriores estão corretas.
- e) Nenhuma das alternativas anteriores está correta.

4. Qual das alternativas a seguir está INCORRETA a respeito dos drivers?

- a) A comunicação com o banco de dados é realizada a partir de um driver, que é um conjunto de classes Java.
- b) Por meio dos drivers, você pode realizar tarefas como a criação de conexões e tabelas, a chamada de Stored Procedures e a execução de comandos SQL.
- c) O driver é uma interface privada que deve ser implementada por todas as classes Driver.
- d) Para cada banco de dados existe um driver correspondente, o qual é fornecido pelo próprio fabricante do banco.
- e) Nenhuma das alternativas anteriores está correta.



19

Banco de dados com Java - JDBC



Mãos à obra!

Willians Martins
305.000-178-45



Editora
IMPACTA



Laboratório 1

Neste laboratório, você criará um cadastro de funcionários.

A – Criando a classe para tratamento de exceções

1. Crie um package **br.com.impacta.java.dao** e, dentro dele, crie uma classe chamada **DAOException**, filha da classe **Exception**;

2. Dentro da classe **DAOException**, crie os quatro construtores adiante e, em cada um deles, utilize o comando **super()** para chamar o respectivo construtor da classe mãe:

- **DAOException();**
- **DAOException(String);**
- **DAOException(Throwable);**
- **DAOException(String, Throwable).**

B – Criando a classe modelo

1. Crie um package chamado **br.com.impacta.java.modelo**, dentro dele, crie a classe **Funcionario**;

2. Na classe **Funcionario**, crie os seguintes atributos e seus respectivos métodos **get** e **set**:

- **int id;**
- **String nome;**
- **double salario;**
- **int cargold.**

C – Criando a classe de acesso à base

1. Crie, no pacote **br.com.impacta.java.dao**, uma classe chamada **FuncionarioDAO**, contendo quatro constantes estáticas e privadas:

- **String DRIVER = “com.mysql.jdbc.Driver”;**
- **String URL = “jdbc:mysql://localhost:3306/impacta”;**
- **String USER = “aluno”;**
- **String PASSWORD = “java”.**

2. Crie, na classe **FuncionarioDAO**, os métodos descritos a seguir:

- **private Connection getConnection() throws DAOException**

Este método será responsável por carregar o driver especificado pela constante **DRIVER** e abrir a conexão com a base de dados utilizando as constantes **URL**, **USER** e **PASSWORD**. A conexão aberta deverá ser retornada para quem tiver chamado o método.

! Utilize todas as instruções dentro de um bloco **try**. No bloco **catch**, realize o tratamento despachando uma **DAOException** contendo a exceção ocorrida. Utilize o comando **throw new DAOException(e)**.

- **private void closeResources(Connection cn, Statement st, ResultSet rs)**

Este método será responsável por fechar os itens passados como parâmetros: **cn**, **st** e **rs**. Cada um destes itens é opcional, o que significa que podem possuir o valor **null**. O método deverá fechar cada um destes parâmetros (se não forem nulos) com um tratamento de erro que não faz nada (bloco **catch** vazio).

D – Criando métodos de acesso a dados

1. Na classe **FuncionarioDAO**, crie o método de inserção:

```
public void persist(Funcionario func) throws DAOException
```

Este método deverá receber um objeto **Funcionario** já com os atributos preenchidos pela aplicação principal e deverá realizar um **INSERT** na tabela **TAB_FUNC**, através de um **PreparedStatement** com o comando SQL:

```
INSERT INTO tab_func (func_name, func_rmnt_val, role_code)  
VALUES (?, ?, ?)
```

2. Preencha os parâmetros do **PreparedStatement** com os dados dos atributos do objeto **Funcionario** e execute-o;

! Todo o código criado neste método deverá ser colocado em um bloco **try**, com exceção das declarações de variáveis de conexão e statement.

3. O bloco **catch** deverá tratar a **SQLException** despachando uma **DAOException** com a mensagem de erro: “**Falha ao salvar dados do funcionário**”. Utilize o comando **throw new DAOException(“msg”, e)**;

4. Crie um bloco **finally** e, dentro dele, feche a conexão e o **PreparedStatement** utilizados chamando: **closeResources(cn, ps, null);**

! Não é necessário exibir nenhuma mensagem de sucesso.

5. Na classe **FuncionarioDAO**, crie um método de busca:

```
public List<Funcionario> findByName(String nome) throws DAOException
```

Este método deverá receber um pedaço do nome de algum funcionário e realizar uma busca com o critério **LIKE**:

```
SELECT func_code, func_name, func_rmnt_val, role_code  
FROM tab_func  
WHERE func_name LIKE ?
```

6. Utilize novamente o **PreparedStatement** para a operação anterior e preencha o parâmetro (interrogação) com o valor da variável **nome** recebida pelo método. Em seguida, utilize o método **executeQuery()** para obter o **ResultSet** com os dados encontrados;

7. Instancie um **ArrayList<Funcionario>** ou **LinkedList<Funcionario>** que deverá conter os objetos de retorno;

8. Para cada linha encontrada no **ResultSet**, crie um objeto **Funcionario**, preencha seus atributos (**id**, **nome**, **salario** e **cargoid**) com os dados das colunas (**func_code**, **func_name**, **func_rmnt_val** e **role_code**) e adicione o objeto à lista instanciada no parágrafo anterior. Ao final do loop, retorne a lista;

! Todo o código criado neste método deverá ser colocado em um bloco **try**, com exceção das declarações de variáveis de conexão, statement e resultset.

9. O bloco **catch** deverá tratar a **SQLException** despachando uma **DAOException** com a mensagem de erro: “**Falha ao realizar consulta**”. Utilize o comando **throw new DAOException(“msg”, e);**

10. Crie um bloco **finally** e, dentro dele, feche a conexão, o statement e o resultset utilizados chamando: **closeResources(cn, ps, rs);**

E – Criando a aplicação principal

1. Crie a classe **ExecutarCadastro** dentro do pacote **br.com.impacta.java** e, dentro dela, crie o método **main()**;
2. Utilize a classe **Scanner**, método **nextLine()**, para solicitar que o usuário digite os seguintes dados: nome, salário e código do cargo;
3. Crie uma instância da classe **Funcionario** e preencha os atributos **setNome()**, **setSalario()** e **setCargoId()** com os valores coletados pelo Scanner. Antes de assinalar os valores, o salário deverá ser convertido para **double** e o código do cargo para **int**;
4. Crie uma instância da classe **FuncionarioDAO** e execute o método **persist()**, passando o objeto criado no passo anterior. Ao final, exiba a mensagem “**Funcionário cadastrado com sucesso**”;
5. Todo o código criado no método **main()** deverá ser colocado em um bloco **try** e deverão ser criados dois blocos **catch** tratando as exceções:
 - **NumberFormatException**: Exibir a mensagem de erro “**Dados digitados inválidos**”;
 - **DAOException**: Exibir a mensagem de erro “**Falha ao salvar dados do funcionário**”.
6. Compile e execute a aplicação.

O resultado deve ser como o exibido a seguir:

```
Console Problems Javadoc Declaration T
<terminated> ExecutarCadastro [Java Application] C:\Java\jdk\jdk1.8
Nome : Manuel
Salario : 8250.15
Cod. Cargo: 4
Funcionário cadastrado com sucesso.
```

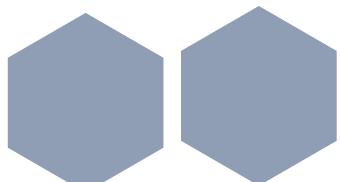



19

Banco de dados com Java - JDBC



Projeto Prático – Fase 5



Conclusão

Agora chegou o momento de construir a camada de persistência da aplicação, assim como implementar os métodos de mapeamento objeto-relacional contidos na classe DAO (Data Access Object).

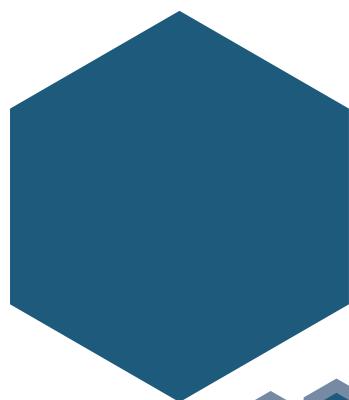
Por simplicidade, e para não sair muito do escopo do treinamento, a classe **Filme** se tornará uma única tabela, cujas colunas “espelham” os atributos da classe. (com a adição de uma chave primária necessária para tabelas de banco de dados relacionais).

Será fornecido o **schema** da base de dados do projeto para importação na ferramenta cliente do banco de dados MySQL, bem como o driver **JDBC** para importação no projeto.

Adiante, veremos as atividades necessárias para conclusão do projeto.

Atividades

1. Importar o **schema** contendo a tabela **Filme** para o MySQL Workbench;
2. Incluir no projeto o driver **JDBC** do MySQL;
3. Criar dois métodos na classe DAO que serão utilizados pelos demais métodos quando necessitarem abrir ou fechar conexões com a base de dados:
 - **getConnection();**
 - **closeConnection();**
4. Implementar na classe DAO os métodos que realizam o mapeamento objeto-relacional a lembrar:
 - Consulta de filmes (via filtro);
 - Consulta de filme (via sorteio);
 - Remoção de filme;
 - Adição de um novo filme.
5. Realizar alguns cadastros manuais e posteriormente importar os dados do arquivo de importação para a tabela de filmes;
6. Testar os fluxos da aplicação, verificando o resultado na base de dados.



Apêndice I



Instalando e configurando
o ambiente Java



Editora
IMPACTA



1.1. Instalando o JDK

O download do JDK (Java Development Kit) pode ser feito a partir do site da Oracle, em <http://www.oracle.com>. No item **Downloads**, basta efetuar o download da versão a ser utilizada. Não é recomendável baixar a versão beta.

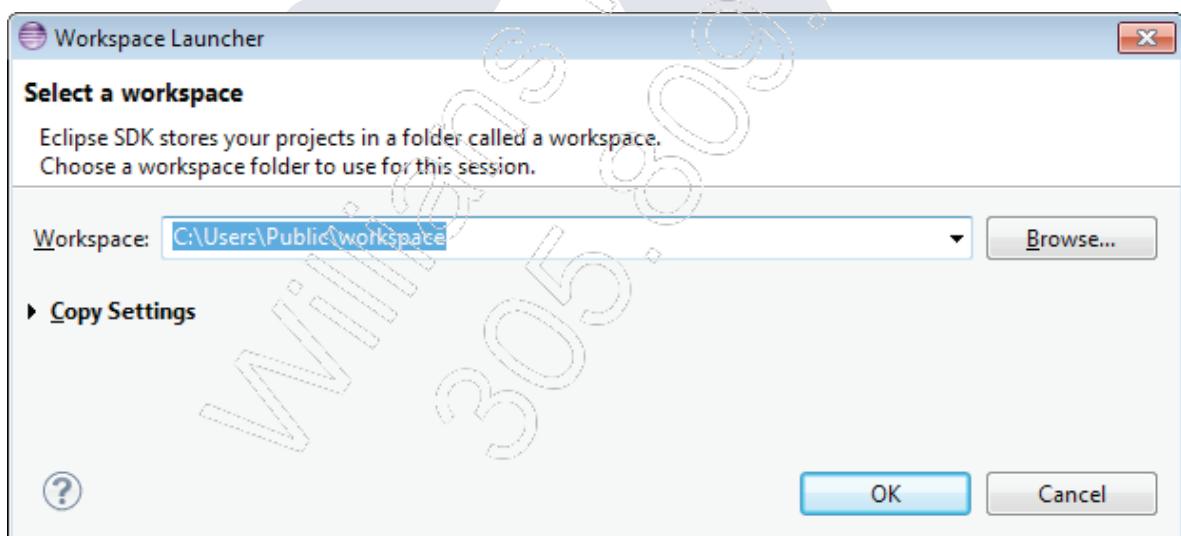
1.2. Instalando o Eclipse

Neste treinamento, vamos utilizar o Eclipse IDE, cujo download pode ser feito a partir do link <http://www.eclipse.org>. A versão que dá suporte ao Java 9 é o **Eclipse Oxygen** ou superior.

Concluído o download, basta descompactar o Eclipse em um diretório. Recomendamos que você utilize os drives C:, E:, F:, entre outros, evitando descompactá-lo na área de trabalho.

Após descompactá-lo em um diretório, será criada uma pasta denominada **eclipse**. Para acessar o Eclipse, então, basta clicar no ícone **eclipse.exe**.

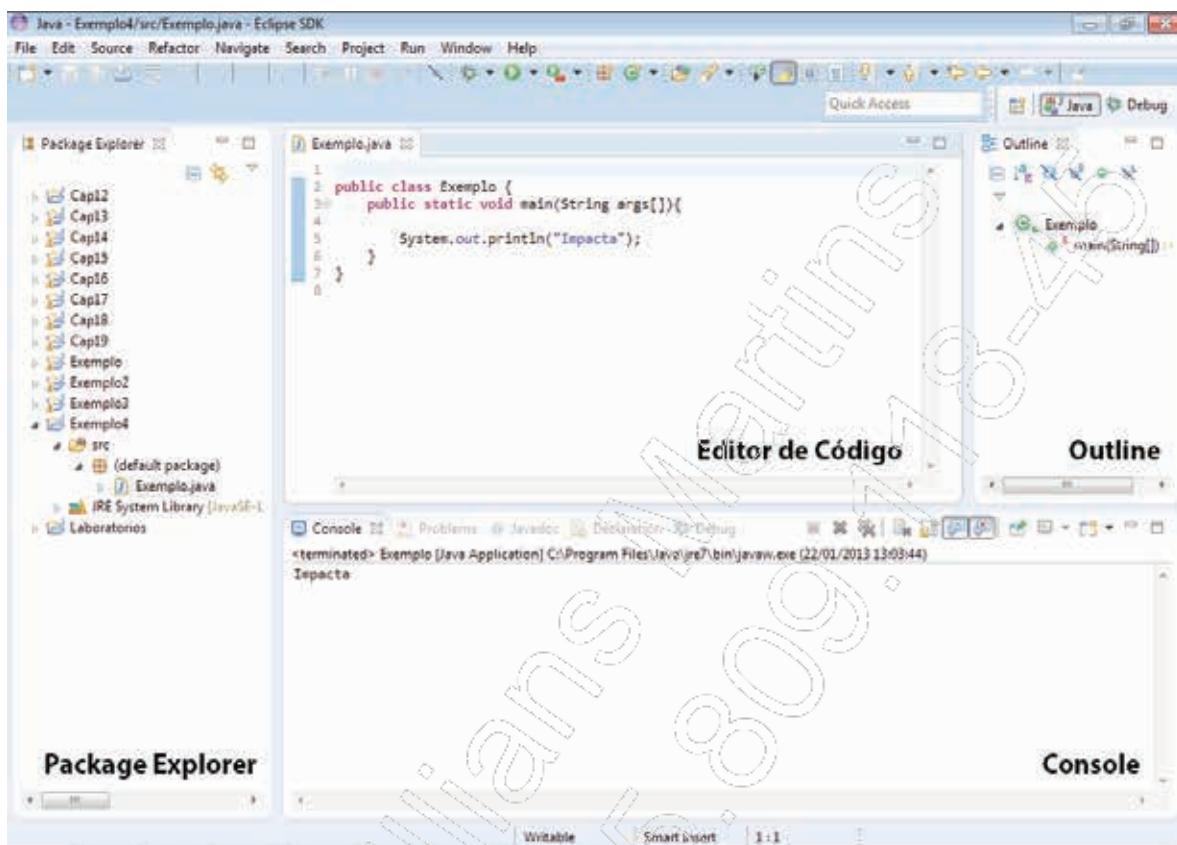
Em sua execução inicial, o Eclipse pede para nomear o workspace, ou seja, a área de trabalho. Para isso, basta alterar o caminho para o diretório desejado, conforme apresentado na imagem adiante:



1.3. Iniciando o Eclipse

Ao clicar no ícone de execução do Eclipse, será exibida uma tela de boas-vindas ao programa. Ao fechá-la, será exibida a tela inicial do ambiente, a partir da qual iremos trabalhar os projetos e códigos Java.

A tela inicial do ambiente Eclipse é dividida em algumas seções, como mostra a imagem a seguir:



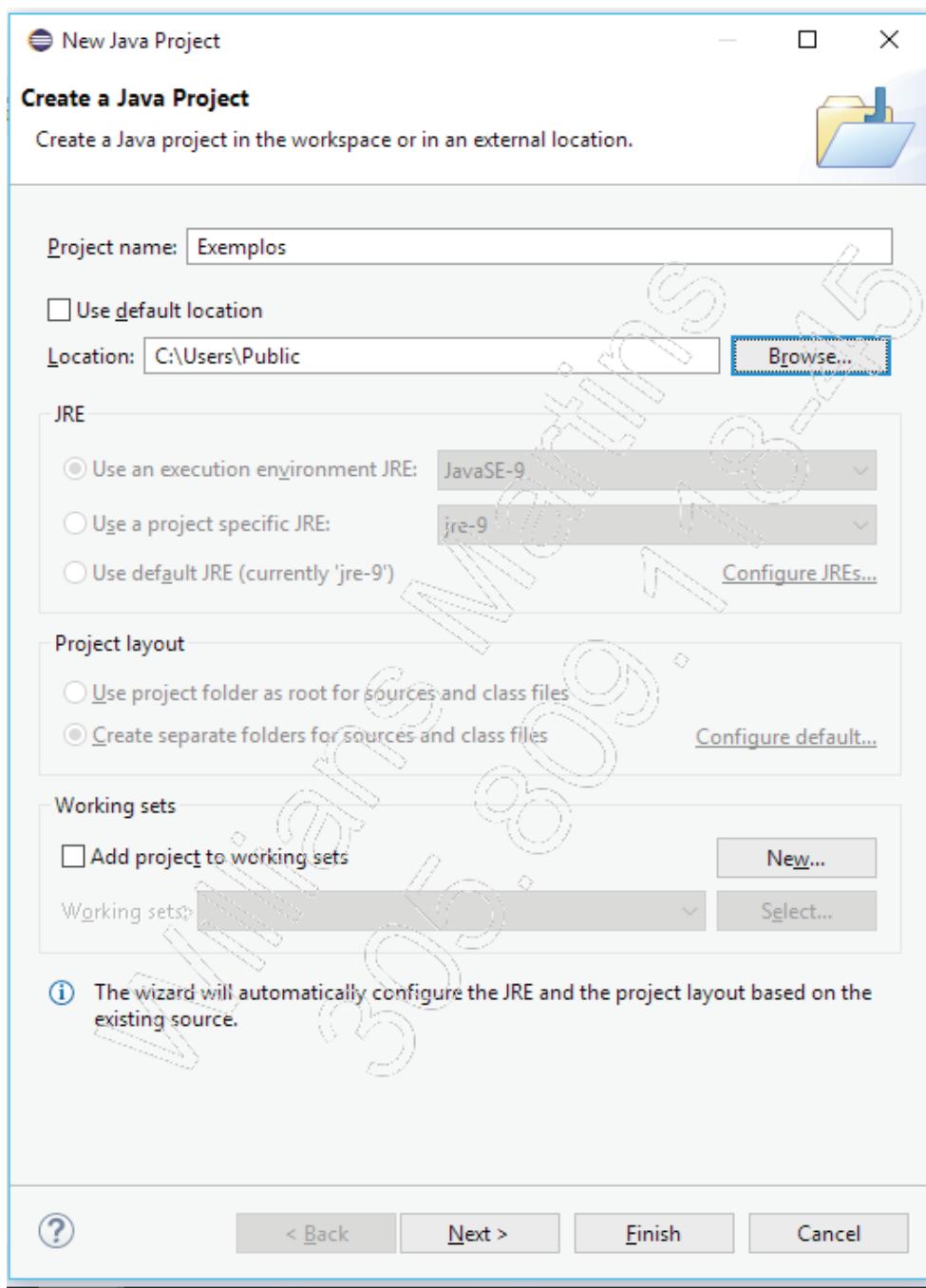
Estas são as principais seções da tela inicial do Eclipse:

- **Package Explorer:** Aqui você pode visualizar o projeto e navegar por toda a sua estrutura;
- **Editor de código:** Nesta seção, você insere e edita os códigos do programa, de acordo com a funcionalidade a ser atingida;
- **Outline:** Esta seção é semelhante ao **Package Explorer**, mas centrada no acesso à estrutura interna do arquivo **.java**. As seções do arquivo são destacadas por ícones;
- **Console:** Nesta área, você visualiza a saída gerada após a compilação e execução do código.

1.4. Criando um projeto

Para criar um projeto, utilize o seguinte procedimento:

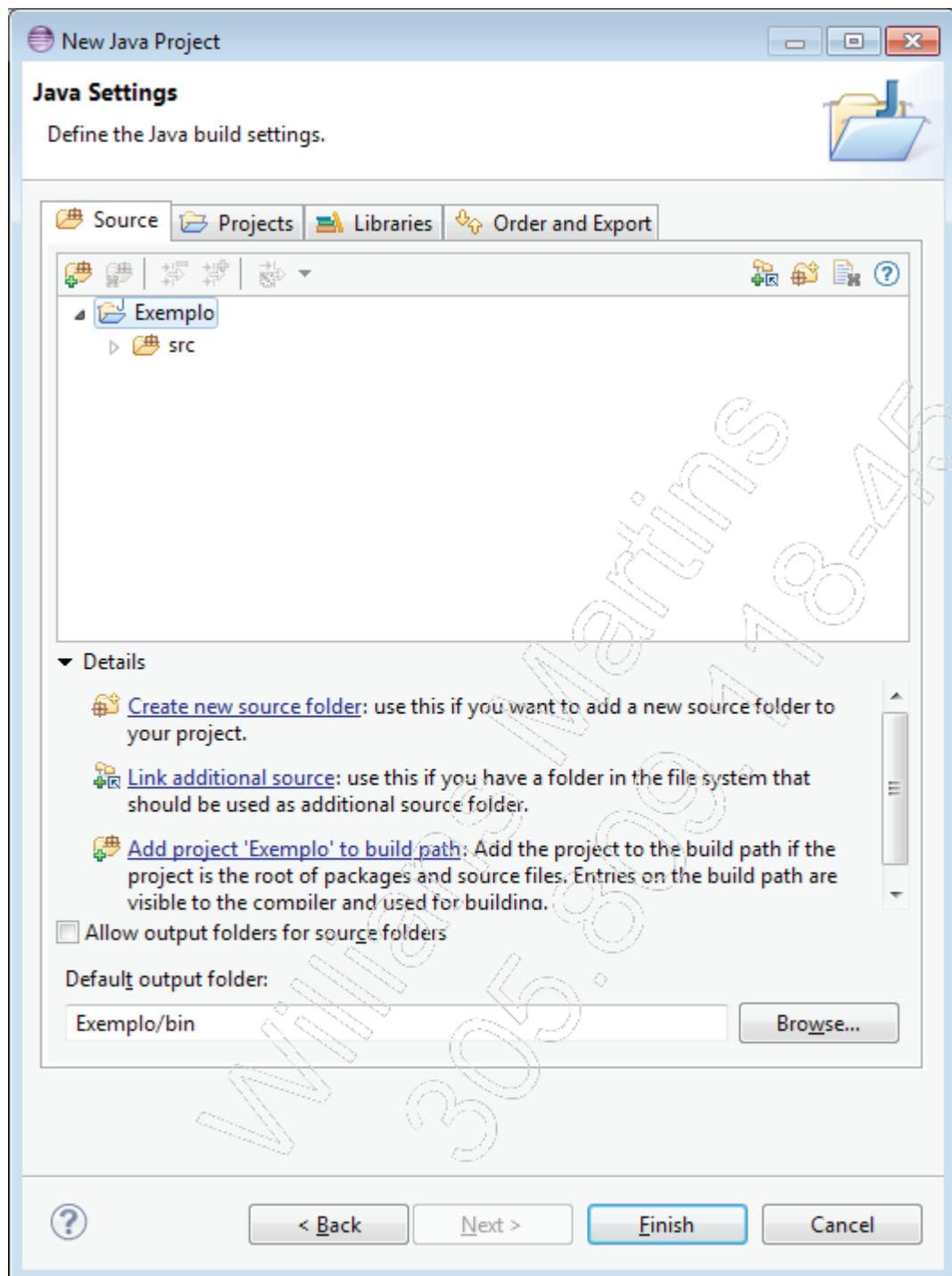
1. Clique em **File / New / Java Project**:



2. Na janela **New Java Project**, defina um nome para o projeto e confirme as opções oferecidas (JRE e layout);

Apêndice I - Instalando e configurando o ambiente Java

3. Clique em **Next**. A seguinte tela será exibida:

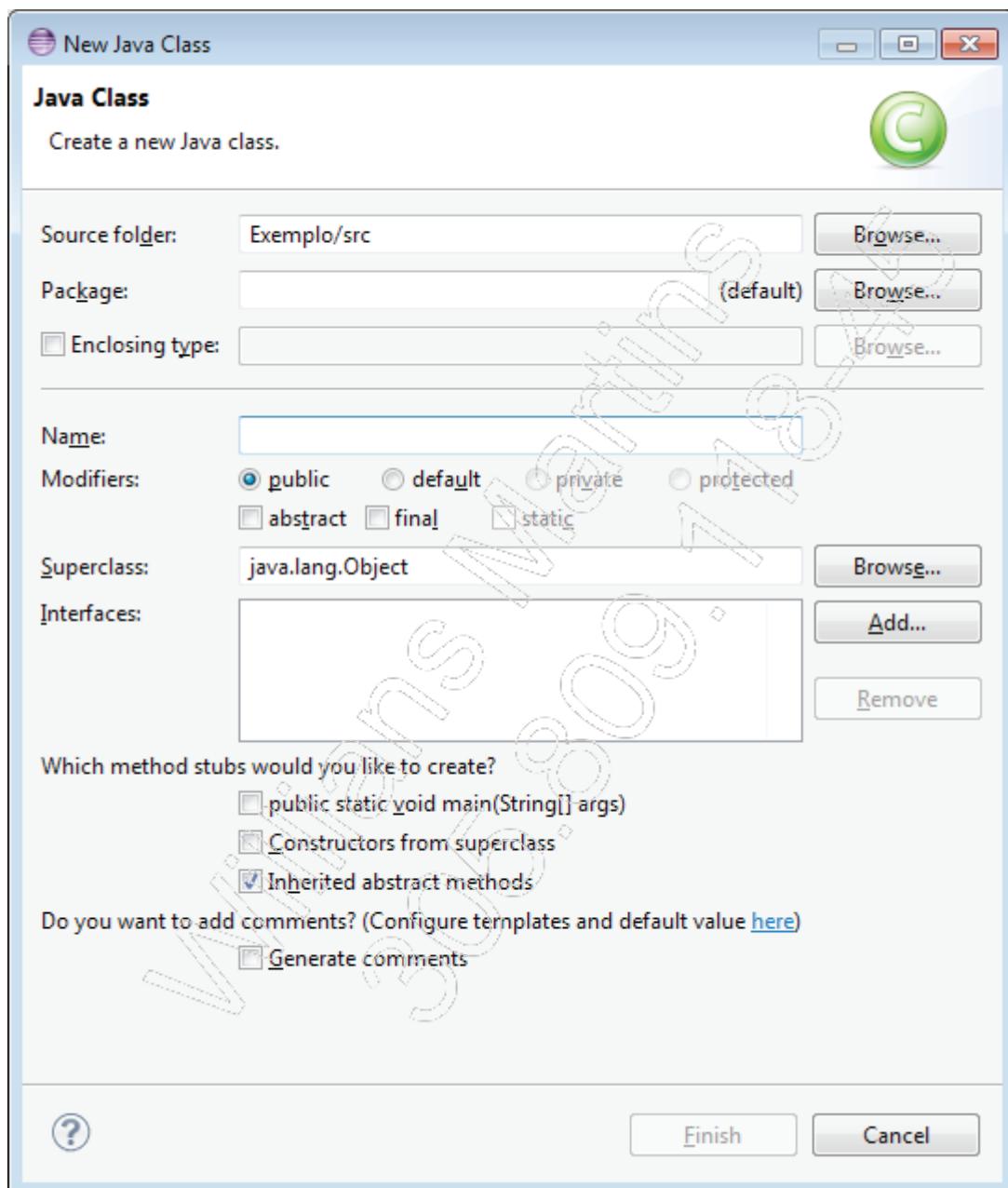


4. Ajuste as configurações de Java e, então, clique em **Finish**. O projeto será criado e exibido no **Package Explorer**.

1.5. Criando uma classe

Veja os procedimentos que você deve adotar para a criação de classes:

1. Clique no menu **File / New / Class**. A mesma opção pode ser acessada ao clicar com o botão direito do mouse sobre o **Package Explorer**:



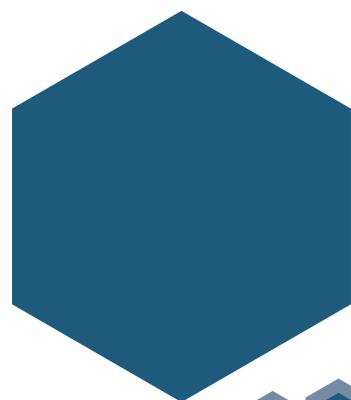
2. Defina um nome para o pacote e a classe;
3. Clique em **Finish**. O arquivo será criado na pasta **src** e aberto para edição.

1.6. Compilando e executando o código

Após você escrever e salvar o código, o Eclipse compilará e exibirá eventuais erros automaticamente. Este tipo de compilação é denominado Incremental.

Para executar o código, basta clicar no botão de execução (botão verde com símbolo de play), disposto na barra de ferramentas do Eclipse. Você também pode utilizar o menu **Run / Run**.

Na execução inicial, selecione a opção **Java Application**. O resultado da execução será exibido no **Console** do Eclipse.



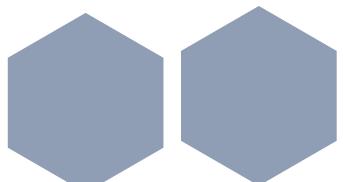
Apêndice II



String, Math e
LocalDate Time



Editora
IMPACTA



2.1. Introdução

Na medida em que é responsável por definir as classes mais importantes, as quais são importadas de forma automática, os pacotes **java.lang** e **java.util** são de suma importância para a linguagem de programação Java. A seguir, abordaremos os conceitos relacionados a três grupos de classes distintos: **String**, **Math** e a classe **LocalDateTime**, do pacote **java.time**.

2.1.1. Classe String

Uma das características do objeto **String** é a inalterabilidade. Uma vez atribuído um valor a ele, este não mais poderá sofrer alterações. Embora não seja possível alterar o objeto **String**, é possível alterar sua variável de referência sem problemas. Veja o exemplo a seguir:

```
nome = nome.concat(" cliente especial");
// o método concat() inclui um valor literal ao final
```

Nesse exemplo, o valor da **String nome**, que é “**João**”, teve “**cliente especial**” adicionado ao seu final, ou seja, o valor resultante é “**João cliente especial**”. Como não é possível alterar uma **String**, a JVM não é capaz de inserir uma nova **String** àquela referenciada por **nome**. Sendo assim, um novo objeto **String**, cujo valor é “**João cliente especial**”, foi criado e, então, referenciado por **nome**.

Perceba que o exemplo tem três objetos **String**: o primeiro com valor “**João**”, o segundo com valor “**João cliente especial**” e o terceiro, que, na verdade, é um argumento literal concatenado, também é um objeto **String** (“**cliente especial**”), embora apenas “**João**” e “**João cliente especial**” sejam referenciados por **nome2** e por **nome**, respectivamente.

Se antes do chamado de **nome = nome.concat(“ cliente especial”)** não fosse criada uma segunda variável de referência para “**João**”, esta seria considerada perdida, uma vez que não haveria um código do programa que fosse capaz de referenciá-la. Lembre-se que, por ser inalterável, a **String** original “**João**” não foi modificada, as alterações foram feitas somente na variável de referência **nome** a fim de que pudesse referenciar outra string. O exemplo a seguir demonstra claramente os detalhes expostos:

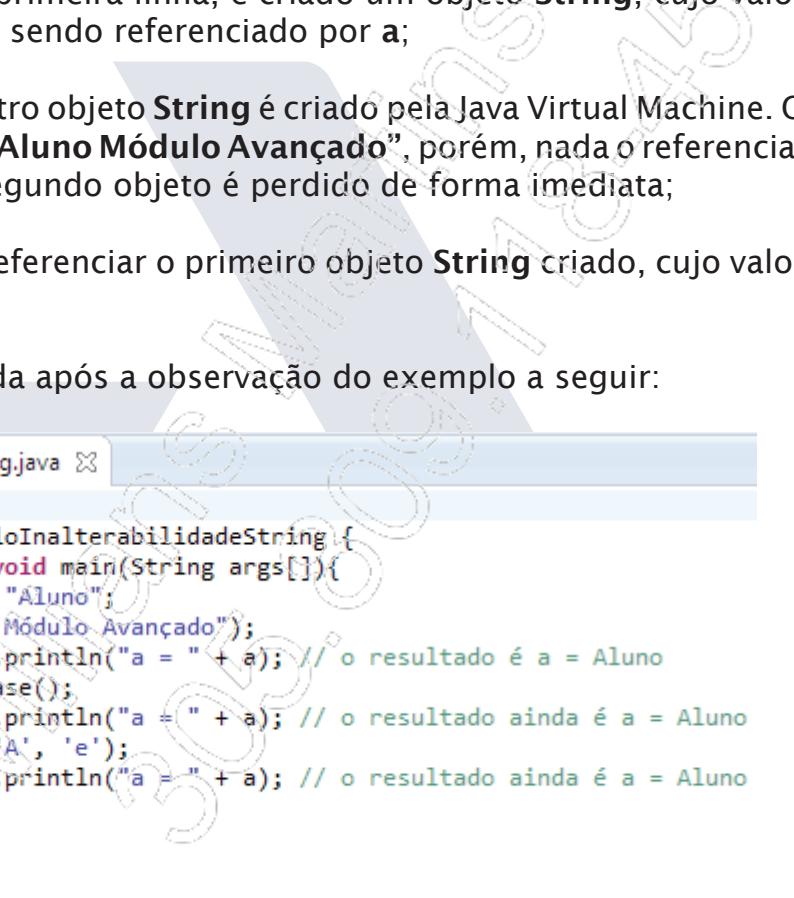
```
String nome = "João";
// cria um novo objeto String com o valor "João",
// nome referencia o objeto

String nome2 = nome;
// cria uma segunda variável de referência que referencia o mesmo objeto String

nome = nome.concat (" cliente especial");
// cria um novo objeto String com valor "João cliente especial"
// nome referencia esse objeto
// altera a referência de nome da String antiga para a nova String
// lembre-se que nome2 ainda faz referência à String original "João".
```

Apêndice II – String, Math e LocalDateTime

A imagem a seguir mostra um exemplo da inalterabilidade da classe **String**:

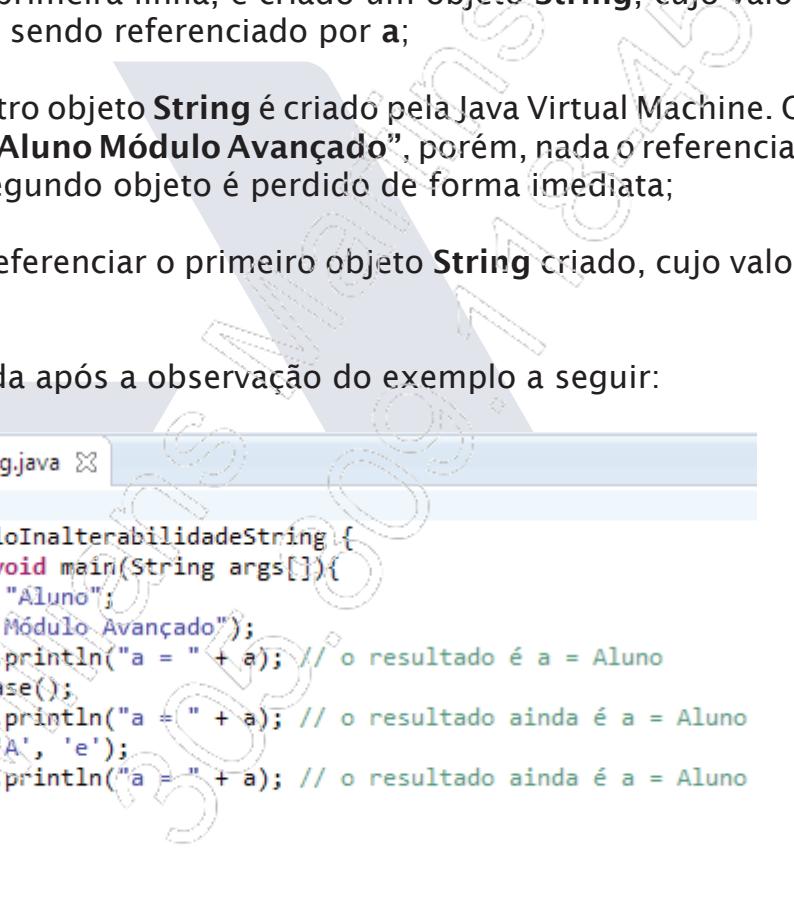


```
J ExemploString.java X
1
2 public class ExemploString {
3     public static void main(String args[]){
4         String a = "Aluno";
5         a.concat(" Módulo Avançado");
6         System.out.println("a = " + a); // o resultado é a = Aluno
7     }
8 }
```

Veja uma explicação a respeito de cada linha deste exemplo:

- Dentro do **main**, na primeira linha, é criado um objeto **String**, cujo valor é “**Aluno**” e que está sendo referenciado por **a**;
- Na segunda linha, outro objeto **String** é criado pela Java Virtual Machine. O valor desse objeto é “**Aluno Módulo Avançado**”, porém, nada o referencia. Sendo assim, esse segundo objeto é perdido de forma imediata;
- Então, **a** continua a referenciar o primeiro objeto **String** criado, cujo valor é **Aluno**.

A explicação será retomada após a observação do exemplo a seguir:



```
J ExemploInalterabilidadeString.java X
1
2 public class ExemploInalterabilidadeString {
3     public static void main(String args[]){
4         String a = "Aluno";
5         a.concat(" Módulo Avançado");
6         System.out.println("a = " + a); // o resultado é a = Aluno
7         a.toUpperCase();
8         System.out.println("a = " + a); // o resultado ainda é a = Aluno
9         a.replace('A', 'e');
10        System.out.println("a = " + a); // o resultado ainda é a = Aluno
11    }
12 }
```

Nesse último exemplo, inicialmente, foi criado um novo objeto **String** cujo valor é “**Aluno**”, porém, em seguida, ele foi perdido. Apesar disso, a string “**Aluno**” original continua sem alterações, bem como a **a** ainda a referencia. Na sequência, um novo objeto **String**, cujo valor é “**eluno**”, foi criado pela VM. Esse objeto, contudo, também é perdido. Dessa forma, o objeto **String** original, cujo valor é “**Aluno**”, permanece sem alterações, e **a** continua referenciando-o.

Embora diversos métodos tenham sido chamados para criar uma nova string que altere a existente, esta não foi alterada em razão de uma variável de referência não ser atribuída à nova string.

O exemplo a seguir demonstra como solucionar esse problema:

```
J Exemplo2ImutabilidadeString.java X
1
2 public class Exemplo2ImutabilidadeString {
3     public static void main(String args[]){
4         String a = "Aluno";
5         a = a.concat(" Módulo Avançado"); // a está sendo atribuído à nova String
6         System.out.println("a = " + a); // o resultado é a = Aluno Módulo Avançado
7     }
8 }
```

De acordo com o exemplo, no momento da execução da segunda linha, é criado o objeto **String “Aluno Módulo Avançado”**, que será referenciado por **a**. O objeto original **“Aluno”** será perdido quando não for referenciado e isso ocorre com a reatribuição de uma nova string à referência **a**. Isso ocorre nas situações em que há dois objetos **String** e uma única variável de referência.

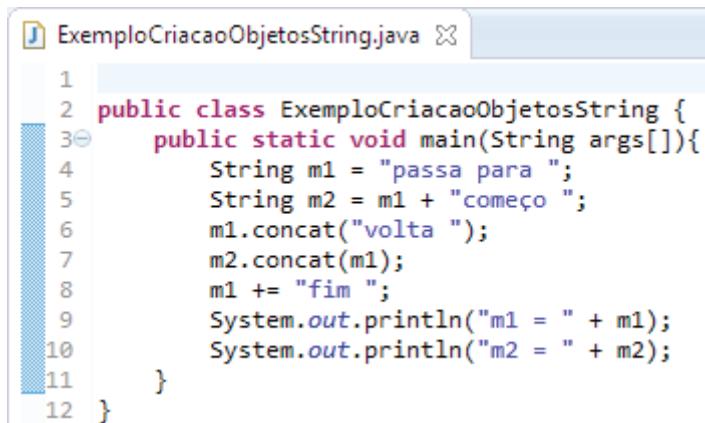
```
J ExemploUsoString.java X
1
2 public class ExemploUsoString {
3     public static void main(String args[]){
4         String a = "Aluno";
5         System.out.println("a = " + a);
6
7         a = a.concat(" Módulo Avançado");
8         System.out.println("a = " + a);
9
10        a = a.toUpperCase();
11        System.out.println("a = " + a);
12
13        a = a.replace('A', 'e');
14        System.out.println("a = " + a);
15    }
16 }
```

Após a compilação e execução do código anterior, o resultado será como o mostrado na imagem a seguir:

```
Console X Problems @
<terminated> ExemploUsoString []
a = Aluno
a = Aluno Módulo Avançado
a = ALLUNO MÓDULO AVANÇADO
a = eLUNO MÓDULO eVeNçeDO
```

Apêndice II – String, Math e LocalDateTime

Antes de finalizarmos, considere o exemplo a seguir:



```
1  public class ExemploCriacaoObjetosString {
2      public static void main(String args[]){
3          String m1 = "passa para ";
4          String m2 = m1 + "começo ";
5          m1.concat("volta ");
6          m2.concat(m1);
7          m1 += "fim ";
8          System.out.println("m1 = " + m1);
9          System.out.println("m2 = " + m2);
10     }
11 }
12 }
```

Neste exemplo, as variáveis de referência são **m1** e **m2**, resultando em “**m1 = passa para fim**” e “**m2 = passa para começo**”. Os objetos **String** criados foram:

- “**passa para**”;
- “**começo**”, que foi perdido;
- “**passa para começo**”;
- “**volta**”, que foi perdido;
- “**passa para volta**”, que foi perdido;
- “**passa para começo passa para**”, que foi perdido;
- “**fim**”, que também foi perdido.

No total, foram criados oito objetos **String**, mas seis foram perdidos, restando apenas dois: “**passa para fim**” e “**passa para começo**”.

Para compreender melhor a inalterabilidade, lembre-se que as linguagens de programação buscam utilizar a memória de maneira eficiente, usando, para isso, a manipulação dos objetos de string na memória. Uma vez que determinadas strings literais tomam um grande espaço na memória e, inclusive, podem ser redundantes dentro de um programa, a Java Virtual Machine mantém uma área reservada chamada de pool constante de strings.

Assim que encontra uma string literal, o compilador verifica o pool constante de strings, com a finalidade de identificar se há uma string literal igual à encontrada. Caso haja essa coincidência, não será criado qualquer novo objeto **String**, e uma referência será direcionada ao novo valor literal para a string já existente.

Para assegurar sua inalterabilidade, uma classe **String** é marcada como final e, assim, os métodos do objeto **String** não são modificados de forma alguma.

2.1.2. Principais métodos

A classe **String** possui diversos métodos. Veja, a seguir, a descrição de alguns dos principais:

- **charAt(Número_do_Índice)**

Este método retorna o caractere que se encontra no índice determinado da classe **String**, que é iniciado em zero.

Veja o seguinte exemplo para melhor compreensão:

```
1
2 public class ExemploCharAt {
3     public static void main(String args[]){
4         String x = "Impacta";
5         System.out.println(x.charAt(2));
6     }
7 }
```

Este exemplo retornará a letra p.

- **concat(Texto_A_Ser_Concatenado)**

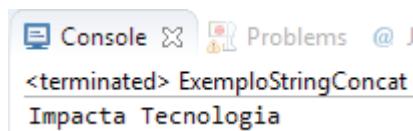
A função deste método é retornar um novo objeto **String**, composto pelo valor da string que foi passada para ele, acrescido ao final da string que foi utilizada a fim de chamar esse objeto.

Veja, a seguir, um exemplo de utilização deste método:

```
1
2 public class ExemploStringConcat {
3     public static void main(String args[]){
4         String x = "Impacta";
5         System.out.println(x.concat(" Tecnologia"));
6     }
7 }
```

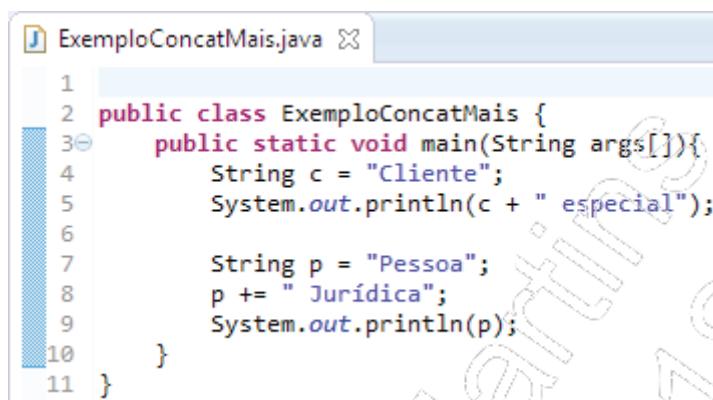
Apêndice II – String, Math e LocalDateTime

Após a compilação e execução do código anterior, o resultado será como ilustrado na imagem a seguir:



The screenshot shows a Java IDE interface with a 'Console' tab active. The output window displays the text: '<terminated> ExemploStringConcat' followed by 'Impacta Tecnologia'. This indicates that the program has completed its execution and printed the string 'Impacta Tecnologia' to the console.

Os operadores de sobreposição + e += têm as mesmas funções do método **concat()**. Veja a seguir um exemplo da utilização desses operadores:



The screenshot shows a Java code editor with a file named 'ExemploConcatMais.java'. The code contains a main method that creates two strings, 'c' and 'p'. It prints 'c' followed by the string 'especial'. Then it concatenates 'p' with the string 'Jurídica' and prints the result. The code is as follows:

```
1
2 public class ExemploConcatMais {
3     public static void main(String args[]){
4         String c = "Cliente";
5         System.out.println(c + " especial");
6
7         String p = "Pessoa";
8         p += " Jurídica";
9         System.out.println(p);
10    }
11 }
```

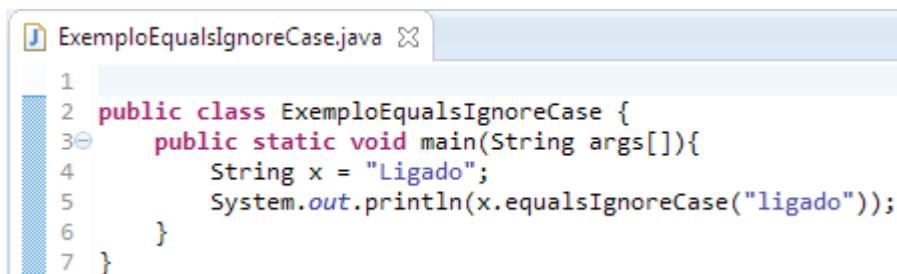
Observe, nesse exemplo, que o valor atribuído a **p** foi alterado de fato. Em razão de **+=** ser um operador de atribuição, na linha de código **p += “Jurídica”;**, um novo objeto **String** é criado e atribuído à variável **p**. Após a execução dessa linha, a string que a variável **p** estava referenciando originalmente é abandonada, neste caso “**Pessoa**”, resultando agora em “**Pessoa Jurídica**”.

- **equalsIgnoreCase(Argumento)**

Este método retorna um valor booleano baseado em uma comparação entre o valor referente à string do argumento e o valor utilizado com a finalidade de chamar o método.

Em razão de os valores booleanos serem **true** ou **false**, o método retornará **true** se houver coincidência entre os valores comparados, independentemente do uso de caixa alta ou baixa em qualquer ponto das strings comparadas.

Veja o exemplo a seguir:

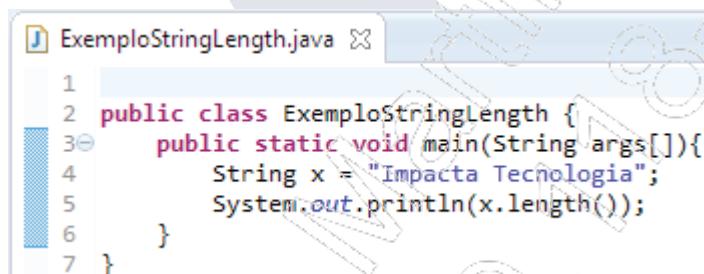


```
ExemploEqualsIgnoreCase.java
1
2 public class ExemploEqualsIgnoreCase {
3     public static void main(String args[]){
4         String x = "Ligado";
5         System.out.println(x.equalsIgnoreCase("ligado"));
6     }
7 }
```

Este exemplo retornará **true**.

- **length()**

Este método tem como função retornar o tamanho apresentado pela string utilizada com a finalidade de chamar o método, como no exemplo a seguir:



```
ExemploStringLength.java
1
2 public class ExemploStringLength {
3     public static void main(String args[]){
4         String x = "Impacta Tecnologia";
5         System.out.println(x.length());
6     }
7 }
```

Este exemplo retornará **18**, pois os espaços também são contados.

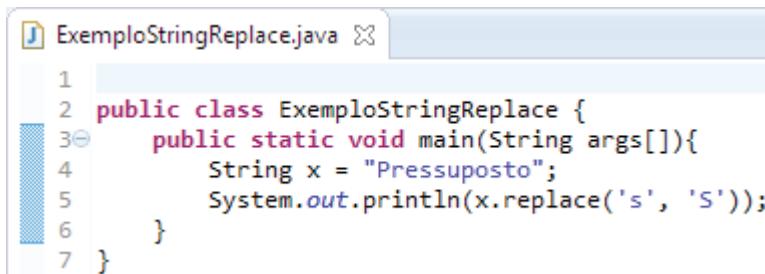
Lembre-se que há um atributo dos arrays que também é chamado de **length**. Assim, a utilização do método **length()** em um array ou, ainda, do atributo **length** em um objeto **String**, gera erros de compilação.

- **replace(Caractere_Antigo, Caractere_Novo)**

Este método retorna um novo objeto **String** que apresenta como valor a **String** utilizada com a finalidade de chamar o método e modificá-la pelas substituições decorrentes.

Ao atualizar esse valor, as ocorrências do tipo **char** referentes ao primeiro argumento são substituídas por ocorrências do tipo **char** referentes ao segundo argumento.

Veja o exemplo a seguir:



```
ExemploStringReplace.java
1
2 public class ExemploStringReplace {
3     public static void main(String args[]){
4         String x = "Pressuposto";
5         System.out.println(x.replace('s', 'S'));
6     }
7 }
```

Esse exemplo retornará **PreSSuposto**.

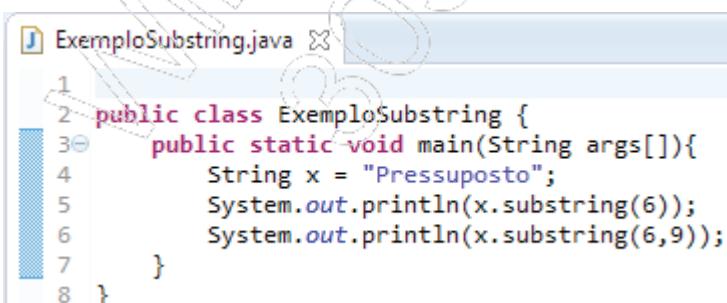
- **substring(Posição_Inicial, Posição_Final)**

A função deste método é retornar uma parte do objeto **String**, o qual foi utilizado na chamada do método. Essa parte do objeto **String** também é denominada **substring**. O primeiro argumento (**Posição_Inicial**) diz respeito ao local em que a substring inicia e começa em zero.

A chamada pode conter um ou dois argumentos. Com um único argumento, a substring retornará todos os caracteres até o final da string original. Porém, se possuir dois argumentos, a substring retornada terminará com o caractere determinado pelo segundo argumento (**Posição_Final**). É importante destacar que somente o segundo argumento não terá início em zero.

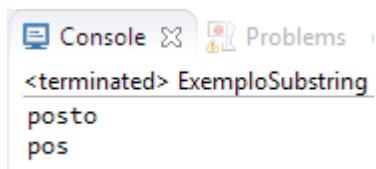
Assim, caso este argumento seja 5, o último caractere da string retornada será da posição 5 da string original, e isso corresponde a 4. Ou seja, como regra geral, o método substring inclui o caractere apontado pelo delimitador inicial e exclui o caractere apontado pelo delimitador final.

Observe o exemplo a seguir:



```
ExemploSubstring.java
1
2 public class ExemploSubstring {
3     public static void main(String args[]){
4         String x = "Pressuposto";
5         System.out.println(x.substring(6));
6         System.out.println(x.substring(6,9));
7     }
8 }
```

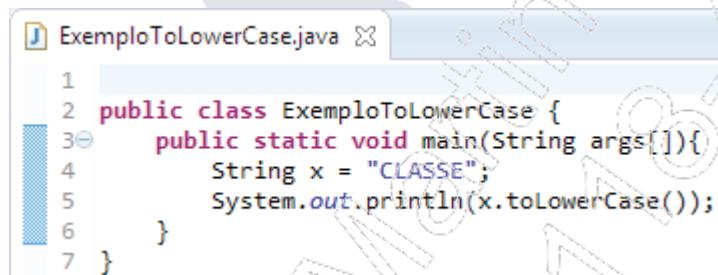
Após a compilação e execução do código anterior, o resultado será como o indicado na imagem a seguir:



The screenshot shows the Eclipse IDE interface. In the top bar, there are tabs for 'Console' and 'Problems'. Below the tabs, it says '<terminated> ExemploSubstring'. The console output window displays two lines of text: 'posto' and 'pos'.

- **toLowerCase()**

A função deste método é retornar um novo objeto **String**, cujo valor refere-se à string que foi utilizada para chamar o método, em caixa baixa. Ao utilizar **toLowerCase()**, o resultado obtido terá todos os seus caracteres convertidos para caixa baixa. Veja:



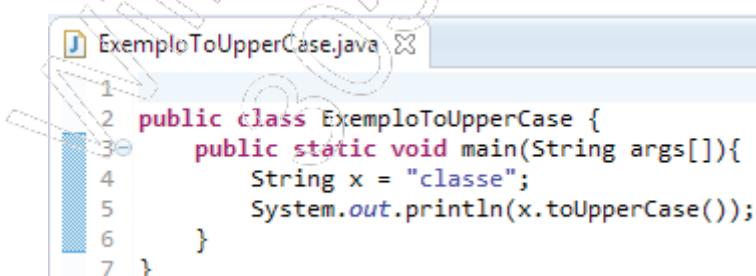
The screenshot shows the Eclipse IDE interface. A file named 'ExemploLowerCase.java' is open in the editor. The code contains a main method that prints the lowercase version of the string 'CLASSE'.

```
1
2 public class ExemploLowerCase {
3     public static void main(String args[]){
4         String x = "CLASSE";
5         System.out.println(x.toLowerCase());
6     }
7 }
```

Este exemplo retornará **classe**.

- **toUpperCase()**

A função deste método é retornar o objeto **String** que contém o valor da string utilizada para chamar o método com todos os caracteres convertidos em caixa alta. Observe:



The screenshot shows the Eclipse IDE interface. A file named 'ExemploUpperCase.java' is open in the editor. The code contains a main method that prints the uppercase version of the string 'classe'.

```
1
2 public class ExemploUpperCase {
3     public static void main(String args[]){
4         String x = "classe";
5         System.out.println(x.toUpperCase());
6     }
7 }
```

Este exemplo retornará **CLASSE**.

- **trim()**

Assim como os outros métodos, **trim()** também retorna um novo objeto **String** que apresenta como valor a string utilizada sem espaços em branco no seu início e fim, caso existam. Veja o exemplo:

```
ExemploTrim.java
1
2 public class ExemploTrim {
3     public static void main(String args[]){
4         String x = "Curso Java ";
5         System.out.println(x + "Programmer");
6         System.out.println(x.trim() + "Programmer");
7     }
8 }
```

Após a compilação e execução do código anterior, o exemplo será como o mostrado na imagem a seguir:

```
Console
<terminated> ExemploTrim
Curso Java Programmer
Curso JavaProgrammer
```

- **toString()**

A função deste método é retornar o valor referente à string utilizada com a finalidade de chamar o método. Lembre-se que todos os objetos da linguagem possuem uma versão deste método, herdado da classe **Object**, o qual retorna uma string capaz de descrever os objetos. Observe a seguir:

```
ExemploToString.java
1
2 public class ExemploToString {
3     public static void main(String args[]){
4         String x = "Conteúdo da String";
5         System.out.println(x.toString());
6     }
7 }
```

Este exemplo retornará **Conteúdo da String**.

2.1.3. Classes **StringBuffer** e **StringBuilder**

Tendo em vista que os objetos **String** são inalteráveis, manipulá-los de forma excessiva pode fazer com que eles fiquem abandonados no pool, gerando uma ocupação desnecessária de memória e perda de performance. Assim, quando quiser realizar alterações em strings de caracteres, você deve usar a classe **StringBuilder** ou **StringBuffer**, uma vez que os objetos do tipo **StringBuilder** e **StringBuffer** podem ser alterados diversas vezes sem que objetos **String** sejam abandonados.

Por serem adequados para a manipulação de blocos de dados, a utilização mais comum dos objetos **StringBuilder** e **StringBuffer** é na entrada e saída de arquivos, nas situações em que o programa manipula diversas alterações e há um grande fluxo de entrada de dados.

Além de serem a melhor opção para a manipulação de dados, estes objetos também são capazes de passá-los adiante após manipulá-los para que, assim, o espaço na memória destinado a essa manipulação possa ser reutilizado.

A classe **StringBuffer** possui determinada sequência de caracteres, que pode ser alterada por meio de chamadas de métodos determinados. Seus métodos são sincronizados, o que torna possível que todas as operações comportem-se como se ocorressem em série, de acordo com a ordem das chamadas de método feitas por cada um dos threads individuais envolvidos.

A classe **StringBuilder** apresenta a mesma API que a classe **StringBuffer**. Entretanto, os métodos de **StringBuilder** são mais rápidos pelo fato de não serem sincronizados (**thread safe**). Por questões de desempenho, sempre que possível, é recomendável utilizar a classe **StringBuilder** em vez de **StringBuffer**. Além disso, a **StringBuilder** foi projetada para ser usada no lugar de **StringBuffer** sempre que o buffer de string estiver sendo usado por um único thread.

2.1.3.1. Métodos da classe **StringBuilder**

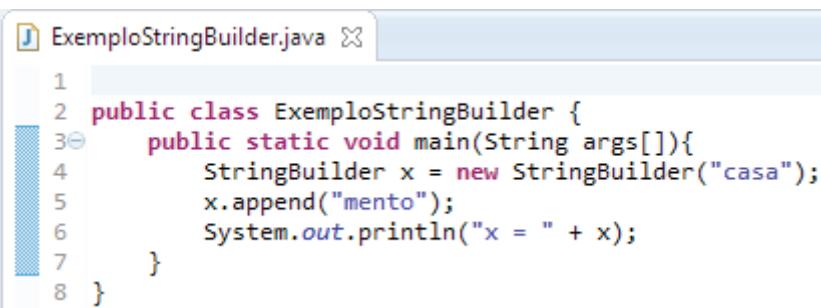
A classe **StringBuilder** possui alguns métodos importantes para a sua utilização. Todos os métodos da classe **StringBuilder** trabalham com o valor do objeto que chama o método. Veja adiante uma descrição dos principais métodos:

- **append (“Texto_a_acrescentar”)**

O método **append()** atualiza o valor do objeto que chamou o método, independentemente de o valor de retorno ter sido atribuído a uma variável. Embora este método utilize argumentos distintos, como **boolean**, **char**, **int**, **long**, entre outros, o que se encontra com mais frequência é o objeto **String**.

Apêndice II – String, Math e LocalDateTime

É importante ressaltar que é possível encadear as chamadas de métodos **append()**, como descrito no seguinte exemplo:

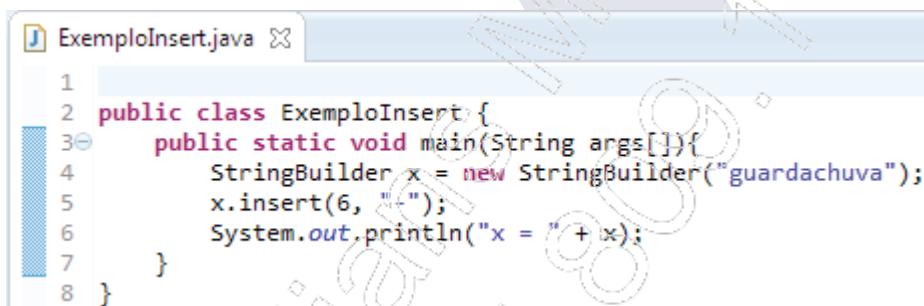


```
ExemploStringBuilder.java
1
2 public class ExemploStringBuilder {
3     public static void main(String args[]){
4         StringBuilder x = new StringBuilder("casa");
5         x.append("mento");
6         System.out.println("x = " + x);
7     }
8 }
```

Este exemplo retornará **casamento**.

- **insert (Posição, Texto_a_ser_inserido)**

Este método é responsável por atualizar o valor do mesmo objeto **StringBuilder** que o chamou, bem como tem a função de retornar o objeto **StringBuilder**. Em ambos os casos, no objeto **StringBuilder** original, será inserido o objeto **String** passado para o segundo argumento, sendo que o início será no primeiro argumento, que começa em zero. Observe:



```
ExemploInsert.java
1
2 public class ExemploInsert {
3     public static void main(String args[]){
4         StringBuilder x = new StringBuilder("guardachuva");
5         x.insert(6, "-");
6         System.out.println("x = " + x);
7     }
8 }
```

Este exemplo retornará **guarda-chuva**.

! Embora o argumento **String** seja visto com mais frequência, este método também permite que diversos tipos de dados sejam passados pelo segundo argumento, como **boolean**, **char**, **int**, **long**, entre outros.

- **reverse()**

Este método é responsável por atualizar o valor do mesmo objeto **StringBuilder** que o chamou, bem como retorna o objeto **StringBuilder**. Em ambos os casos, há uma inversão dos caracteres do objeto **StringBuilder**, como descrito no exemplo a seguir:

```
ExemploReverse.java
1 public class ExemploReverse {
2     public static void main(String args[]){
3         StringBuilder x = new StringBuilder("Roma");
4         x.reverse();
5         System.out.println("x = " + x);
6     }
7 }
8 }
```

Este exemplo retornará **amoR**.

- **toString()**

A função deste método é retornar o valor do objeto **StringBuilder** que chamou o método. Esse valor é retornado como um objeto **String**.

Veja o exemplo a seguir:

```
ConversaoParaString.java
1 public class ConversaoParaString {
2     public static void main(String args[]){
3         StringBuilder x = new StringBuilder("Teste");
4         System.out.println(x.toString());
5     }
6 }
7 }
```

Este exemplo retornará **Teste**.

2.1.4. Métodos encadeados

Neste tópico, apresentamos o conceito de métodos encadeados. Veja, a seguir, a sintaxe de uma instrução que apresenta métodos encadeados:

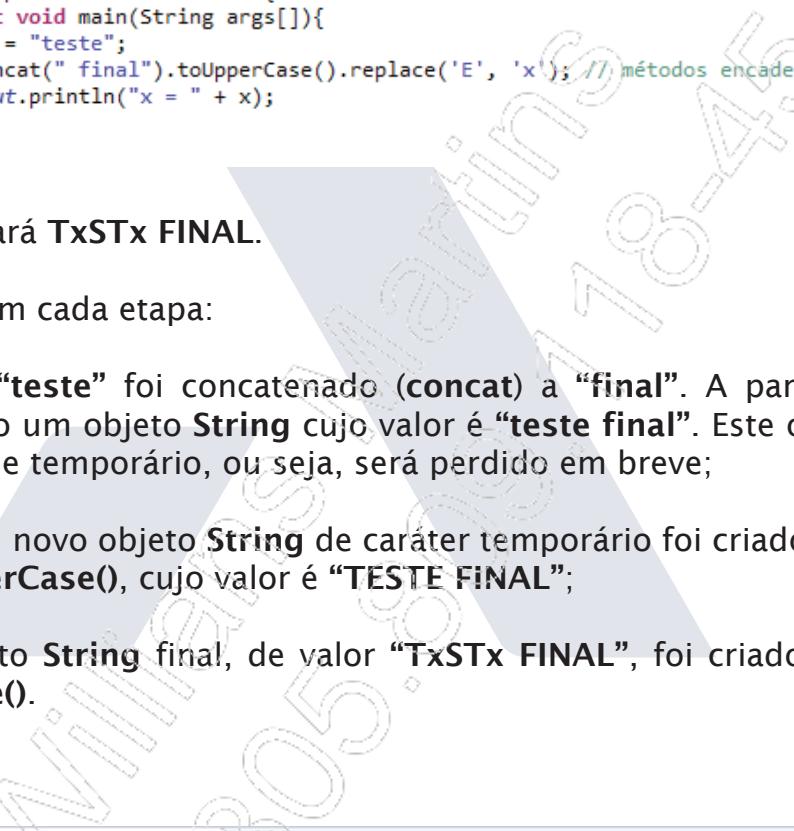
```
resultado = método1().método2().método3();
```

Para compreender esses métodos encadeados, em primeiro lugar, você deve verificar o que será retornado da chamada do método que se encontra na extrema esquerda. Esse valor retornado será chamado de A.

Apêndice II – String, Math e LocalDateTime

Considerando a contagem a partir da esquerda, o valor A será utilizado como um objeto responsável por chamar o segundo método. Caso haja somente dois métodos encadeados, o resultado da expressão será correspondente ao resultado da segunda chamada. Contudo, se houver um terceiro método, o resultado obtido a partir da segunda chamada será utilizado para chamar esse terceiro método. Neste caso, o resultado da expressão será correspondente ao resultado dessa terceira chamada.

Observe o exemplo descrito a seguir para compreender melhor a explicação:



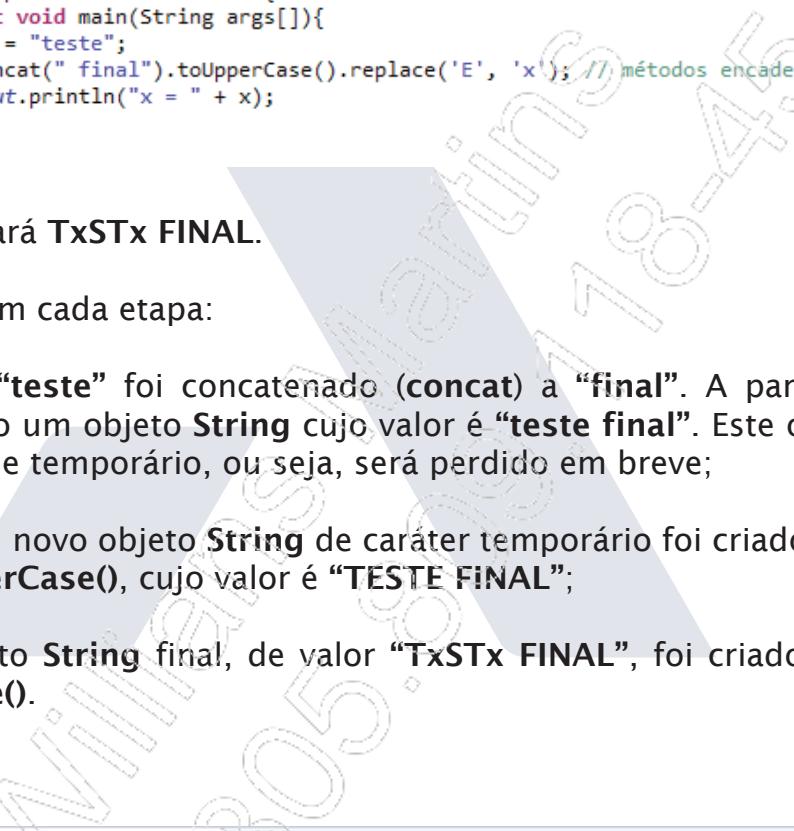
```
J ExemploMetodosEncadeados.java X
1
2 public class ExemploMetodosEncadeados {
3     public static void main(String args[]){
4         String x = "teste";
5         x = x.concat(" final").toUpperCase().replace('E', 'x'); //métodos encadeados
6         System.out.println("x = " + x);
7     }
8 }
```

Este exemplo retornará **TxSTx FINAL**.

Veja o que ocorreu em cada etapa:

- O valor literal “**teste**” foi concatenado (**concat**) a “**final**”. A partir de então, foi criado um objeto **String** cujo valor é “**teste final**”. Este objeto é intermediário e temporário, ou seja, será perdido em breve;
- Em seguida, um novo objeto **String** de caráter temporário foi criado pelo método **toUpperCase()**, cujo valor é “**TESTE FINAL**”;
- Então, um objeto **String** final, de valor “**TxSTx FINAL**”, foi criado pelo método **replace()**.

Veja outro exemplo:



```
J ExemploMetodosEncadeados2.java X
1
2 public class ExemploMetodosEncadeados2 {
3     public static void main(String args[]){
4         String a = "Pessoa Física";
5         String b = a.concat(" Jurídica").toUpperCase().substring(7);
6         System.out.println(b);
7     }
8 }
```

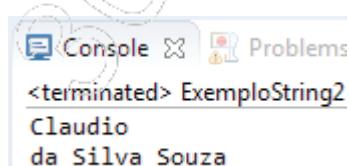
Este exemplo retornará **FÍSICA JURÍDICA** porque **String a** recebeu o texto “**Pessoa Física**”. Perceba que **String b** recebeu o encadeamento dos métodos **concat**, **toUpperCase** e **substring**. Assim, o método **a.concat** acrescentou o texto “**Jurídica**”, o método **toUpperCase** transformou o texto em letras maiúsculas e o método **substring** retornou uma parte do conteúdo da **String b** a partir da 7ª posição.

2.1.5. Utilizando a classe String

A seguir, você encontra um exemplo sobre a utilização dos métodos da classe **String**:

1. Crie uma classe com o nome **ExemploString2** e insira a estrutura básica de um programa Java;
2. Declare três variáveis do tipo **String** com os nomes **nomeCompleto**, **nome** e **sobrenome**;
3. Inicialize a variável **nomeCompleto** com o seu nome completo e as outras duas variáveis com uma string vazia;
4. Faça um **for** que percorra cada caractere da variável **nomeCompleto** e, dentro dele, verifique se o caractere da posição atual é igual ao caractere vazio (‘ ’). Se sim, atribua a substring de 0 até a posição atual à **nome** e da posição atual até o final à **sobrenome**, depois insira um **break** para parar o loop **for**;
5. Imprima as variáveis **nome** e **sobrenome** na tela sem espaços no começo e fim das strings;
6. Compile e execute o programa.

O resultado será o seguinte:



```
Console X Problems
<terminated> ExemploString2
Claudio
da Silva Souza
```

2.2. Classe Math

Você deve usar a classe **Math** com a finalidade de efetuar operações matemáticas comuns. Uma vez que todos os métodos da classe **Math** são definidos como **static**, eles não precisam ser instanciados para serem utilizados.

A classe **Math** tem duas limitações que devem ser destacadas:

- Em virtude de seu construtor ser privado, não é possível criar uma instância da classe **Math**, afinal, ela é uma classe criada para ser utilizada com cunho utilitário;
- Essa classe também não pode ser estendida por ser definida como **final**.

Além disso, a classe **Math** é responsável por determinar aproximações para as constantes **pi** e **e**, cujas assinaturas são **public final static double Math.PI** e **public final static double Math.E**, respectivamente.

2.2.1. Métodos

Em razão de serem estáticos, os métodos da classe **Math** podem ser acessados a partir do nome da classe. Normalmente, esses métodos são chamados da seguinte maneira:

```
resultado = Math.umMétodoMathEstático();
```

A seguir, descrevemos esses métodos **Math** seguidos pelos exemplos de sua utilização:

- **abs()**

Este método retorna o valor absoluto de um número. Possui quatro assinaturas:

- **public static double abs (double a);**
- **public static float abs (float a);**
- **public static int abs (int a);**
- **public static long abs (long a).**

Veja um exemplo:

```
int x = Math.abs(-1);
```

O retorno será 1.

- **max()**

Este método retorna o maior entre dois números distintos. Possui quatro assinaturas:

- **public static double max (double a, double b);**
- **public static float max (float a, float b);**
- **public static int max (int a, int b);**
- **public static long max (long a, long b).**

Veja um exemplo:

```
int x = Math.max(920, 810);
```

O retorno será 920.

- **min()**

Oposto ao **max()**, retorna o menor entre dois números distintos. Também possui quatro assinaturas:

- **public static double min (double a, double b);**
- **public static float min (float a, float b);**
- **public static int min (int a, int b);**
- **public static long min (long a, long b).**

Veja um exemplo:

```
int x = Math.min(920, 810);
```

O retorno será 810.

- **random()**

Sem utilizar quaisquer parâmetros, retorna um valor entre 0.0 e 1.0, do tipo **double**. A assinatura deste método é **public static double random ()**.

Veja um exemplo:

```
double x = Math.random();
```

Retorna um valor aleatório entre 0.0 e 1.0.

- **ceil()**

Retorna um número inteiro correspondente ao inteiro superior mais próximo ao argumento. Este número está em formato **double**. A assinatura deste método é **public static double ceil (double a)**.

Veja um exemplo:

```
double x = Math.ceil(3.9);
```

O retorno será 4.

- **floor()**

Diferente do método **ceil()**, retorna um número inteiro correspondente ao inteiro inferior mais próximo ao argumento, também em formato **double**. A assinatura deste método é **public static double floor (double a)**.

Veja um exemplo:

```
double x = Math.floor(3.9);
```

O retorno será 3.

- **round()**

Este método retorna um número inteiro que esteja mais próximo ao número utilizado no argumento. Possui duas assinaturas:

- **public static int round (float a);**
- **public static long round (double a).**

Veja um exemplo:

```
double x = Math.round(-6.5);
```

O retorno será **-6.0**.

- **sin()**

Este método retorna o seno de um ângulo, sendo que o argumento é um **double** que refere-se ao ângulo em radianos. Por meio de **Math.toRadians()**, é possível converter graus em radianos. A assinatura deste método é **public static double sin (double a)**.

Veja os seguintes exemplos:

```
double x = Math.sin(90);
```

Este retorna **0.8939966676...**

```
double x = Math.sin(Math.toRadians(90));
```

Aqui, o retorno será **1**.

- **cos()**

Este método retorna o cosseno de um ângulo. O argumento é um **double** que se refere ao ângulo em radianos. A assinatura deste método é **public static double cos (double a)**.

Veja um exemplo:

```
double x = Math.cos(0);
```

O retorno será **1**.

- **tan()**

Retorna a tangente de um ângulo. O argumento é um **double** que se refere ao ângulo em radianos. A assinatura deste método é **public static double tan (double a)**.

Veja um exemplo:

```
double x = Math.tan(Math.toRadians(45.0));
```

O retorno será **0.999999999....**

- **sqrt()**

Retorna a raiz quadrada de um valor **double**. A assinatura deste método é **public static double sqrt (double a)**.

Veja um exemplo:

```
double x = Math.sqrt(9.0);
```

O retorno será **3**.

 Se o valor especificado for negativo, o método retorna **NaN (Not a Number)**, um padrão de bits que indica que o resultado não é um número.

- **toDegrees()**

Este método retorna um ângulo em graus a partir do argumento, o qual representa o ângulo em radianos. A assinatura deste método é **public static double toDegrees (double a)**.

Veja um exemplo:

```
double x = Math.toDegrees(Math.PI * 2);
```

O retorno será **360**.

- **toRadians()**

Diferente do método anterior, retorna um ângulo em radianos a partir do argumento, o qual representa o ângulo em graus. A assinatura deste método é **public static double toRadians (double a)**.

Veja um exemplo:

```
double x = Math.toRadians(360 / 2);
```

O retorno será **3.14 (o valor de PI)**.

2.2.2. Utilizando a classe Math

A seguir, você encontra um exemplo sobre a utilização dos métodos da classe **Math**:

1. Crie uma classe com o nome **ExemploMath** e insira a estrutura básica de um programa Java;
2. Declare um array do tipo **int** com o nome **numeros** de tamanho **10** e uma variável do tipo **int** com o nome **maior** e a inicialize com **0**;
3. Faça um loop **for** que percorra todo o array **numeros**. Dentro do loop, declare uma variável com o nome **n** do tipo **Double** e atribua a ele um número randômico entre **1** e **100**;
4. Atribua o valor inteiro de **n** à posição atual do array **numeros**;
5. Utilizando o método **max()** da classe **Math**, verifique se o número da posição atual do array **numeros** é maior que a variável **maior** e atribua o valor retornado pelo método à variável **maior**;
6. Imprima o número da posição atual do array **numeros** e feche o loop **for**;
7. Logo após o loop **for**, imprima a variável **maior** na tela;
8. Compile e execute o programa.

O resultado será o seguinte:

```
Console Problems @ Javi
<terminated> ExemploMath [Java Appli]
42 29 48 61 88 75 87 57 92 56
Maior número do array é 92
```

2.3. Classe LocalDateTime

A classe **LocalDateTime**, pertencente ao pacote **java.time**, faz parte da nova API de manipulação de data e hora lançada pelo Java 8.

Através desta classe, podemos, dentre outras coisas:

- Obter detalhes sobre um instante do tempo, como dia, hora, mês;
- Comparar datas e efetuar cálculos;
- Formatar datas e horas.

Uma instância da classe **LocalDateTime** representa um instante “congelado” no tempo, que possui data e hora com precisão de nanossegundos, podendo ser utilizada tanto para cálculos de data/hora como para formatação e exibição de datas.

2.3.1. Método now()

Utilizamos o método estático **now()** para obter uma instância da classe **LocalDateTime** contendo o instante atual computado a partir do relógio do sistema operacional:

```
LocalDateTime variavel = LocalDateTime.now();
```

2.3.2. Método of()

Utilizamos o método estático **of()** para obter uma instância da classe **LocalDateTime** contendo um momento específico no tempo. Ao executar este método, devemos informar ano, mês, dia, hora e minutos do instante desejado (já segundos e milissegundos são opcionais):

```
LocalDateTime variavel = LocalDateTime.of(  
    ANO, MES, DIA, HORA, MINUTO [, SEGUNDO [, MILIS]]);
```

2.3.3. Método parse()

Outra forma de obtermos uma instância de **LocalDateTime** é através do método estático **parse()**, pelo qual podemos obter a data e hora a partir de uma string:

```
LocalDateTime variavel =  
    LocalDateTime.parse("0000-00-00T00:00:00.000");
```

Observe que a string especificada precisa estar no formato “0000-00-00T00:00:00.000”.

O método **parse()** permite a utilização da classe auxiliar **DateTimeFormatter** do pacote **java.time.format** para converter strings de diferentes formatos. Um **DateTimeFormatter** representa uma máscara de conversão entre string e data, podendo estar padronizada no formato escolhido pelo programador.

Para criar uma máscara de conversão com o padrão desejado, utilize o método estático **ofPattern()** da classe **DateTimeFormatter**:

```
DateTimeFormatter mascara =  
DateTimeFormatter.ofPattern("[FORMATO_MASCARA]");
```

Podemos, então, aplicar esta máscara ao método **parse()** para obter a data/hora a partir de um string no formato especificado:

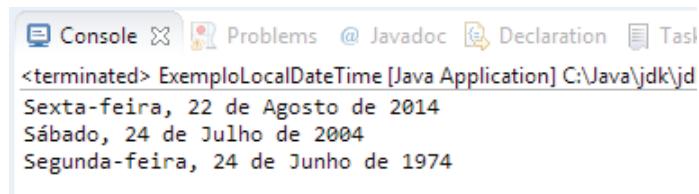
```
LocalDateTime variavel =  
LocalDateTime.parse("[DATA_HORA]", mascara);
```

Observe um exemplo completo utilizando os três métodos de criação de data/hora:

```
J ExemploLocalDateTime.java X  
1@ import java.time.LocalDateTime;  
2 import java.time.format.DateTimeFormatter;  
3  
4 public class ExemploLocalDateTime {  
5  
6@     public static void main(String[] args) {  
7  
8         DateTimeFormatter mascara1 =  
9             DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm:ss");  
10  
11         LocalDateTime agora = LocalDateTime.now();  
12         LocalDateTime casa = LocalDateTime.of(2004, 7, 24, 21, 18, 30, 500);  
13         LocalDateTime nasc = LocalDateTime.parse("24/06/1974 03:15:30", mascara1);  
14  
15         DateTimeFormatter mascara2 =  
16             DateTimeFormatter.ofPattern("eeee, d 'de' MMMM 'de' yyyy");  
17  
18         System.out.println(agora.format(mascara2));  
19         System.out.println(casa.format(mascara2));  
20         System.out.println(nasc.format(mascara2));  
21     }  
22 }
```

Apêndice II – String, Math e LocalDateTime

Agora, confira o resultado:



```
Console Problems @ Javadoc Declaration Tasks
<terminated> ExemploLocalDateTime [Java Application] C:\Java\jdk\jd
Sexta-feira, 22 de Agosto de 2014
Sábado, 24 de Julho de 2004
Segunda-feira, 24 de Junho de 1974
```

Os seguintes caracteres podem ser utilizados na criação de máscaras de data e hora:

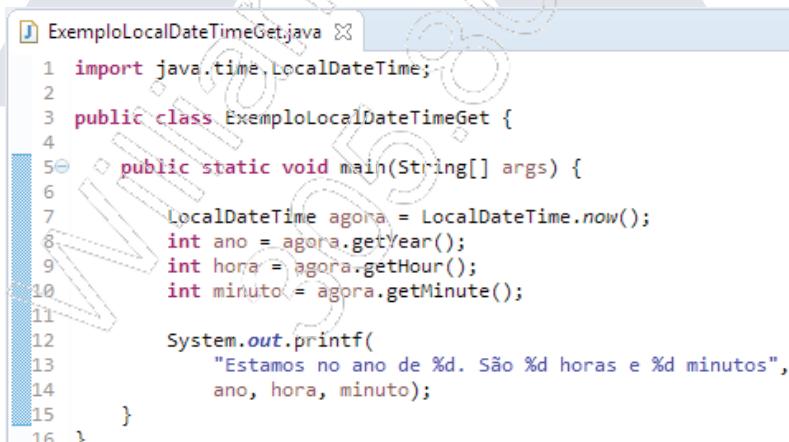
Símbolo	Descrição
y	Ano
M	Mês
d	Dia do mês (1-31)
h	Hora (1-12)
H	Hora (0-23)
m	Minuto
s	Segundo
S	Milissegundo
E	Dia da semana
D	Dia do ano
w	Semana do ano
W	Semana do mês
a	Marcador AM/PM

2.3.4. Métodos get()

Conforme já visto, uma instância da classe **LocalDateTime** contém as informações a respeito de um instante na linha do tempo. Esta classe possui diversos métodos **get()**, a partir dos quais podemos obter cada um dos detalhes isoladamente, como ano, hora, dia, minuto etc.

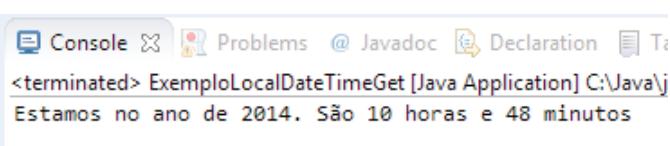
Método	Descrição
getYear()	Retorna o ano.
getMonth()	Retorna uma das instâncias da enumeração Month .
getMonthValue()	Retorno o número do mês (1-12).
getDayOfMonth()	Retorna o dia do mês (1-31).
getDayOfWeek()	Retorna uma das instâncias da enumeração DayOfWeek .
getHour()	Retorna a hora do dia (0-23).
getMinute()	Retorna o minuto (0-59).
getSecond()	Retorna o segundo (0-59).
getNano()	Retorna o nanossegundo.

Veja um exemplo:



```
ExemploLocalDateTimeGet.java
1 import java.time.LocalDateTime;
2
3 public class ExemploLocalDateTimeGet {
4
5     public static void main(String[] args) {
6
7         LocalDateTime agora = LocalDateTime.now();
8         int ano = agora.getYear();
9         int hora = agora.getHour();
10        int minuto = agora.getMinute();
11
12        System.out.printf(
13             "Estamos no ano de %d. São %d horas e %d minutos",
14             ano, hora, minuto);
15    }
16 }
```

Agora, confira o resultado:



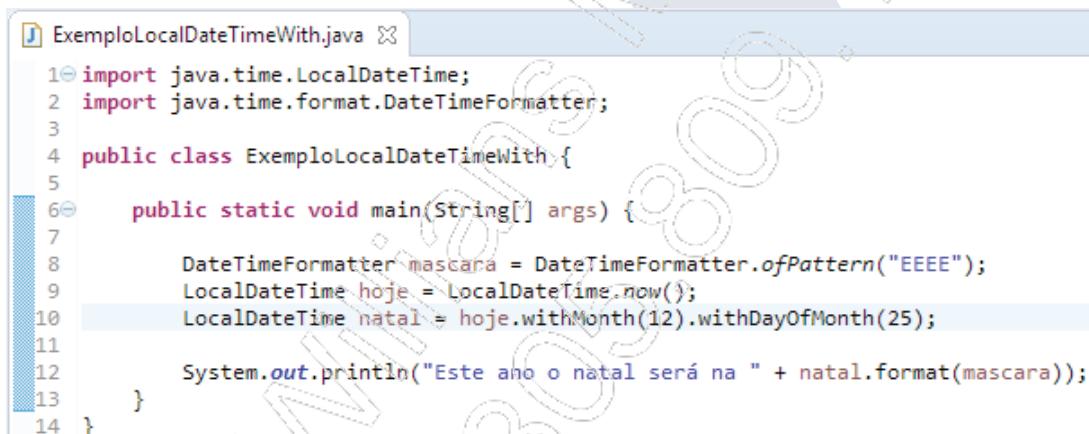
```
Console Problems @ Javadoc Declaration Terminal
<terminated> ExemploLocalDateTimeGet [Java Application] C:\Java\j
Estamos no ano de 2014. São 10 horas e 48 minutos
```

2.3.5. Métodos with()

Os métodos **with()** são utilizados para gerar um novo instante no tempo a partir de um instante original, copiando todos os seus dados e modificando apenas o atributo especificado.

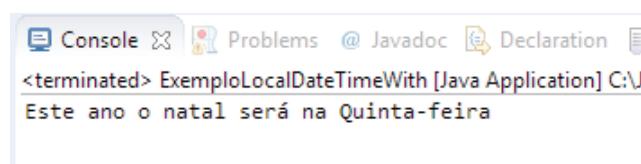
Método	Descrição
withYear(int)	Gera uma cópia com o ano modificado.
withMonth(int)	Gera uma cópia com o mês modificado.
withDayOfMonth(int)	Gera uma cópia com o dia modificado.
withHour(int)	Gera uma cópia com a hora modificada.
withMinute(int)	Gera uma cópia com o minuto modificado.
withSecond(int)	Gera uma cópia com o segundo modificado.
withNano(int)	Gera uma cópia com o nanossegundo modificado.

Os métodos **with()** geralmente são utilizados em cascata quando desejamos manipular mais de um atributo de data/hora. Veja o exemplo:

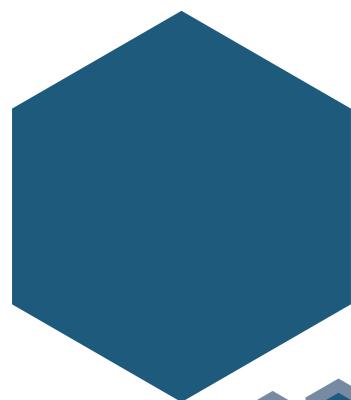


```
ExemploLocalDateTimeWith.java
1 import java.time.LocalDateTime;
2 import java.time.format.DateTimeFormatter;
3
4 public class ExemploLocalDateTimeWith {
5
6     public static void main(String[] args) {
7
8         DateTimeFormatter mascara = DateTimeFormatter.ofPattern("EEEE");
9         LocalDateTime hoje = LocalDateTime.now();
10        LocalDateTime natal = hoje.withMonth(12).withDayOfMonth(25);
11
12        System.out.println("Este ano o natal será na " + natal.format(mascara));
13    }
14 }
```

Agora, confira o resultado:



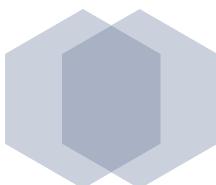
```
Console <terminated> ExemploLocalDateTimeWith [Java Application] C:\J
Este ano o natal será na Quinta-feira
```

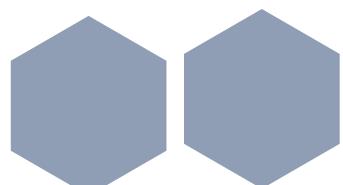
Apêndice III

JavaFX

Williams Martins
305.809.718-45



Editora
IMPACTA



3.1. Introdução

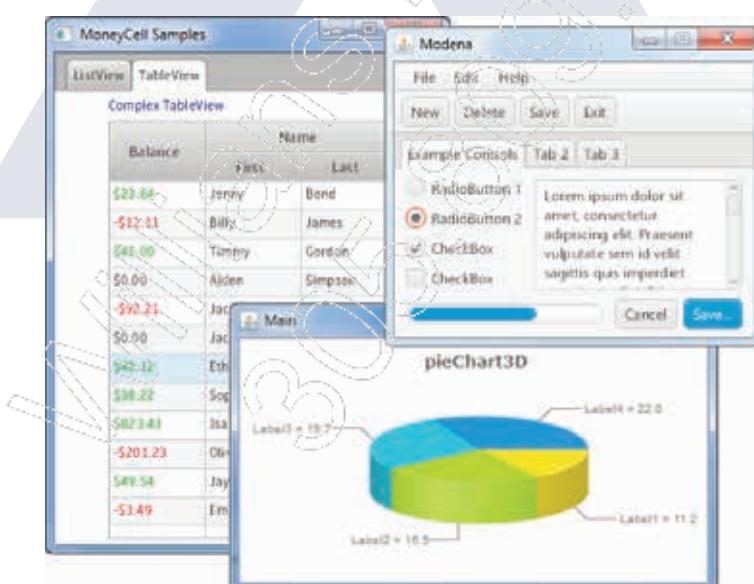
Com o Java, podemos criar aplicações que interagem com o usuário através de elementos como caixas de diálogos e janelas, que podem conter botões, caixas de texto, ícones, menus, e outros elementos interativos. De uma forma geral, chamamos a estes componentes de interface gráfica, cujo principal objetivo é tornar a sua aplicação amigável e de fácil utilização pelo usuário.

A elaboração de interfaces gráficas com Java vem sofrendo uma série de inovações à medida que a linguagem evolui. Diversas bibliotecas já foram criadas e modificadas visando à construção de tais aplicações por programadores Java. As bibliotecas mais conhecidas são: **AWT**, **Swing** e **SWT**.

O **JavaFX** é a mais recente biblioteca elaborada para a construção de aplicações gráficas. Suas principais características são:

- Facilidade na elaboração de janelas e outros componentes visuais através da ferramenta **Scene Builder**;
- Permite a criação de layouts sofisticados e efeitos visuais de altíssima complexidade através do uso de folhas de estilos CSS.

Observe:



Outra característica marcante do JavaFX é a possibilidade de total isolamento entre interface gráfica e código Java responsável pelas regras de negócio, podendo, inclusive, ser reutilizado em ambientes não-desktop, como dispositivos mobile ou em páginas Web na forma de plugins.

Este isolamento é possível através da criação de arquivos **FXML**. Trata-se de um formato de arquivo texto que usa o padrão XML para descrever toda a aparência da interface gráfica a ser utilizada.

Um arquivo FXML contém posição, tamanho, aparência, cor e todas as outras características de cada componente utilizado em uma tela.

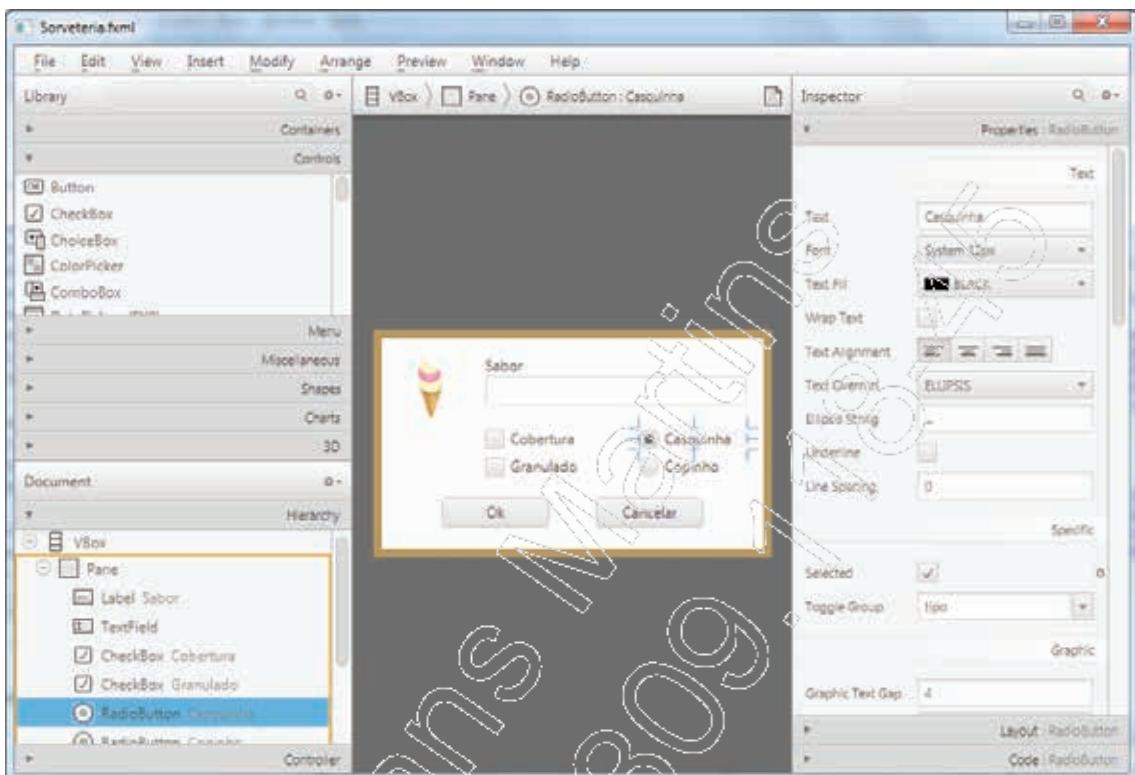
Veja um exemplo:

```
① Sorveteria.fxml ✘
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import javafx.scene.image.*?>
4 <?import javafx.scene.control.*?>
5 <?import java.lang.*?>
6 <?import javafx.scene.layout.*?>
7 <?import javafx.scene.layout.BorderPane?>
8
9<VBox alignment="CENTER" xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fx
10<children>
11<Pane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-In
12<children>
13    <Label layoutX="86.0" layoutY="14.0" prefHeight="17.0" prefWidth="217.0" text=
14        <TextField fx:id="txtSabor" layoutX="86.0" layoutY="31.0" prefHeight="25.0" pr
15        <CheckBox layoutX="86.0" layoutY="75.0" mnemonicParsing="false" prefHeight="17
16        <CheckBox layoutX="86.0" layoutY="98.0" mnemonicParsing="false" prefHeight="17
17        <RadioButton layoutX="214.0" layoutY="75.0" mnemonicParsing="false" prefHeight
18        <toggleGroup>
19            <ToggleGroup fx:id="tipo" />
20        </toggleGroup>
21    </RadioButton>
22    <RadioButton layoutX="214.0" layoutY="98.0" mnemonicParsing="false" prefHeight
23    <Button fx:id="btnOk" layoutX="50.0" layoutY="132.0" mnemonicParsing="false" o
24    <Button fx:id="btnCancelar" layoutX="178.0" layoutY="132.0" mnemonicParsing="f
25    <Image fitHeight="61.0" fitWidth="52.0" layoutX="14.0" layoutY="14.0" pick
26        <image>
27            <Image url="@resources/sorvete.png" />
28        </image>
```

3.2. Scene Builder

Embora seja possível criar e editar interfaces gráficas FXML com qualquer editor de texto, a Oracle dispõe de uma ferramenta IDE especificamente criada para isso: o **Scene Builder**.

Observe:

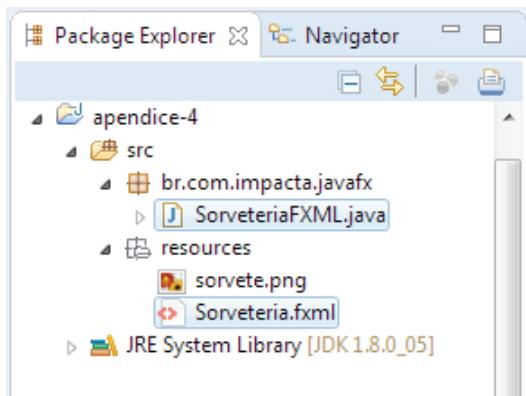


Através dele, podemos elaborar interfaces ricas utilizando todos os recursos do JavaFX, em que podemos manipular tamanho e posição dos componentes com o clique e arraste e alterar as suas características pelo painel de propriedades.

O **Scene Builder** pode ser obtido gratuitamente na página de downloads do site da Oracle/Java. Para isso, acesse a opção **JavaFX Scene Builder**, da seção **Additional Resources**, no seguinte endereço: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Ao salvar a sua interface gráfica com o **Scene Builder**, será gerado um arquivo FXML que poderá ser posteriormente editado pela própria ferramenta ou em algum editor de texto.

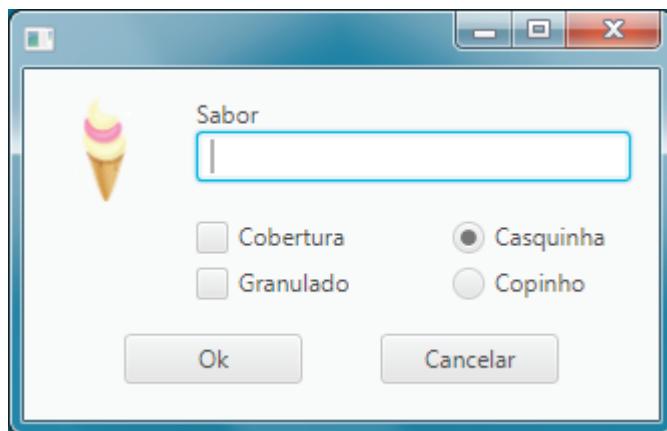
Após gerado, o arquivo **.fxml** deverá ser incorporado à sua aplicação Java:



Em seguida, poderá ser renderizado pela biblioteca JavaFX, através da classe **FXMLLoader**:

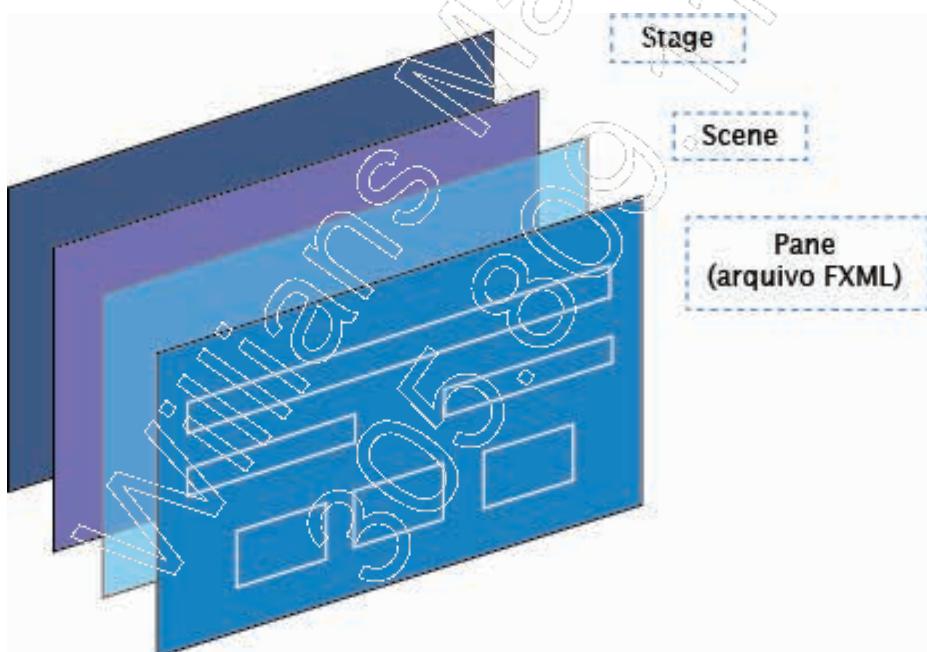
```
J SorveteriaFXML.java x
1 package br.com.impacta.javafx;
2
3 import javafx.application.Application;
4 import javafx.fxml.FXMLLoader;
5 import javafx.scene.Parent;
6 import javafx.scene.Scene;
7 import javafx.stage.Stage;
8
9 public class SorveteriaFXML extends Application {
10
11     public static void main(String[] args) {
12         launch(args);
13     }
14
15     public void start(Stage stage) throws Exception {
16
17         FXMLLoader loader = new FXMLLoader(
18             getClass().getResource("/resources/Sorveteria.fxml"));
19         Parent parent = loader.load();
20         Scene scene = new Scene(parent);
21
22         stage.setScene(scene);
23         stage.show();
24     }
25 }
```

Isso exibirá a janela equivalente ao arquivo **Sorveteria.fxml**:



3.3. Principais componentes

Cada componente de uma janela, inclusive a própria janela, é representado por uma classe da biblioteca JavaFX. A seguir, veremos os principais componentes visuais desta biblioteca:



3.3.1. Stage

Todos os componentes visuais do JavaFX são renderizados em uma região denominada **Window**. A classe `javafx.stage.Window` trata-se de uma abstração que possui diferentes implementações, em conformidade com o ambiente/sistema operacional em que a aplicação está sendo executada.

Em uma aplicação para ambiente desktop, os componentes são renderizados em um **Stage** (classe `javafx.stage.Stage`), que representa uma janela contendo as funcionalidades básicas de maximizar, minimizar, exibir título etc. Suas principais propriedades podem ser assinaladas pelo métodos:

Propriedade/Método	Descrição
<code>setTitle()</code>	Permite assinalar o texto a ser exibido na barra de títulos da janela.
<code>setWidth()</code> e <code>setHeight()</code>	Definem a largura e altura inicial da janela, respectivamente.
<code>setX()</code> e <code>setY()</code>	Definem a posição da janela em relação à tela do computador (área de trabalho do sistema operacional).
<code>setResizable()</code>	Define se a janela possuirá tamanho fixo (false) ou variável (true).
<code>show()</code>	Exibe a janela.

A classe **Stage** pode ser normalmente instanciada para a exibição de diversas janelas de sua aplicação. No entanto, a primeira janela a ser aberta (janela principal) não poderá ser instanciada diretamente.

Para isso, crie uma classe filha de `javafx.application.Application` e implemente o método `start()`, que receberá o **Stage** principal de sua aplicação.

Para executar esta classe, utilize sempre o método `lauch()`, conforme exemplo anterior.

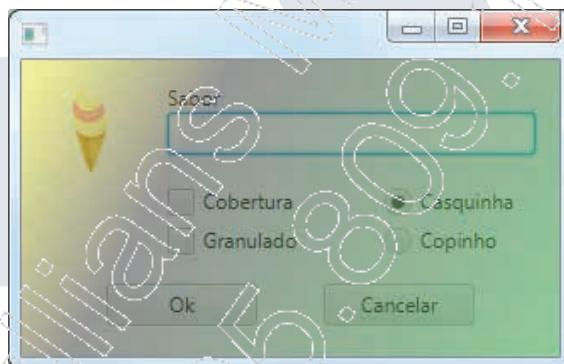
3.3.2. Scene

A classe **javafx.scene.Scene** representa a película de fundo da região em que os elementos gráficos serão exibidos. Através do método **setFill()**, podemos definir uma cor, foto ou efeito de fundo para sua janela. Este método aceita os seguintes tipos como argumentos:

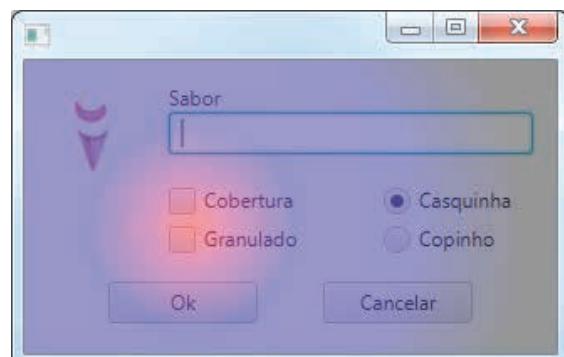
Tipo	Descrição
ImagePattern	Permite definir uma imagem de fundo para sua janela.
LinearGradient	Permite definir um efeito de mudança linear de cores.
RadialGradient	Permite definir um efeito de mudança radial de cores.
Color	Define uma simples cor de fundo para sua janela.

Veja os exemplos a seguir:

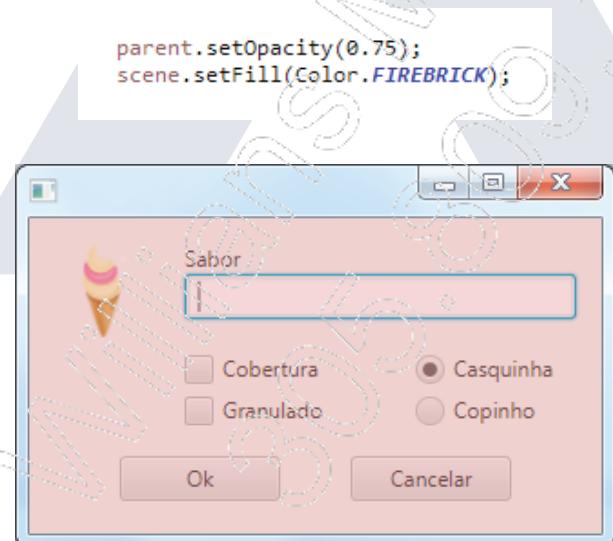
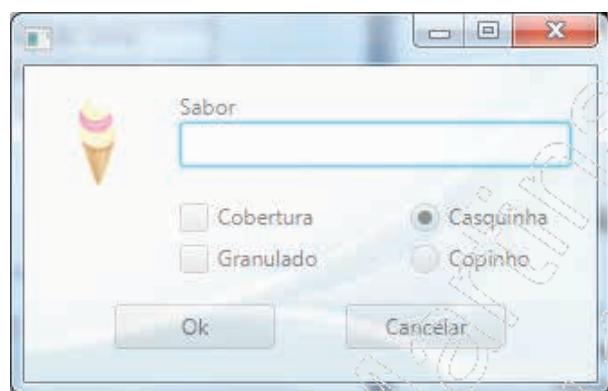
```
parent.setOpacity(0.5);
scene.setFill(LinearGradient.valueOf(
    "from 0% 0% to 100% 100%", yellow 0%, black 30%, green 100%));
```



```
parent.setOpacity(0.5);
scene.setFill(RadialGradient.valueOf(
    "center 100px 100px, radius 200px, red 0%, blue 30%, black 100%"));
```



```
parent.setOpacity(0.75);
scene.setFill(new ImagePattern(
    new Image(getClass().getResourceAsStream("/resources/background.jpg"))));
```

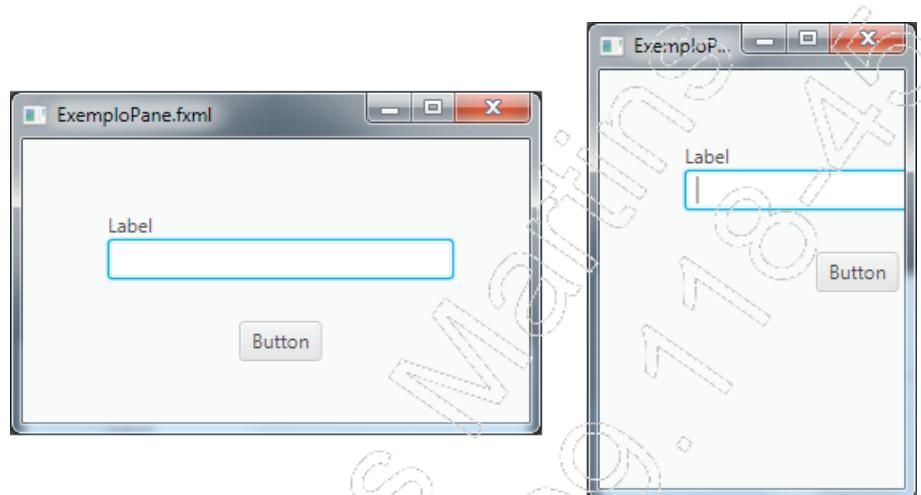


3.3.3. Pane

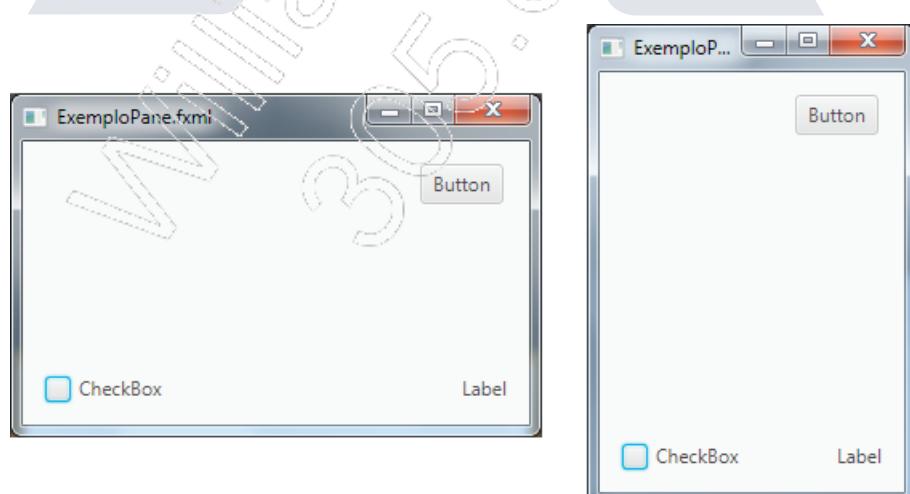
javafx.scene.layout.Pane é a classe base para os diversos tipos de painéis da biblioteca JavaFX. Utilizamos painéis para especificar o posicionamento dos demais componentes em tela, de forma a se reorganizarem conforme o tamanho da janela.

Segue adiante uma breve descrição dos principais painéis, todos pertencentes ao pacote **javafx.scene.layout**:

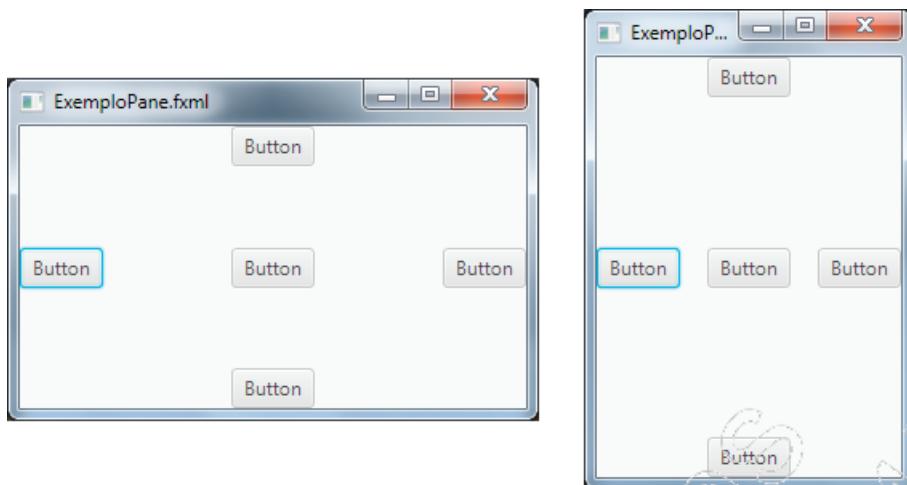
- **Pane**: Principal tipo de painel, cujos elementos internos possuem tamanho e posição fixa:



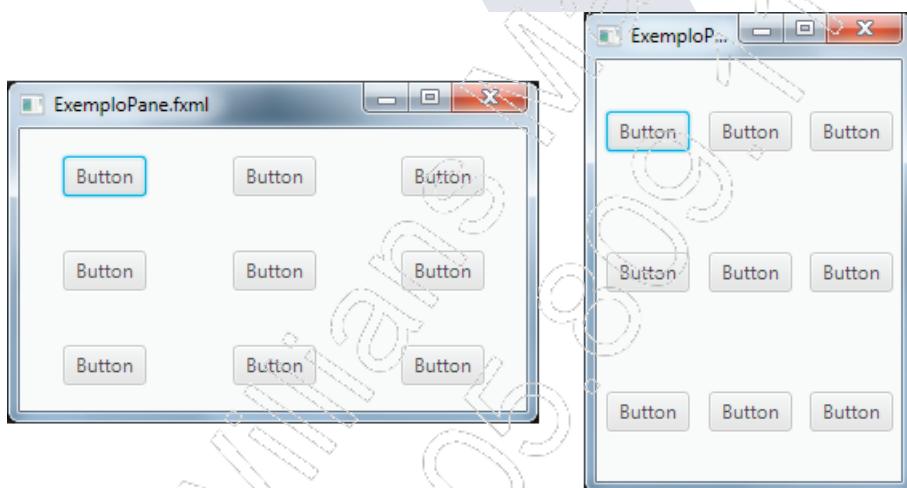
- **AnchorPane**: Permite criar elementos que aderem a um dos quatro cantos da tela:



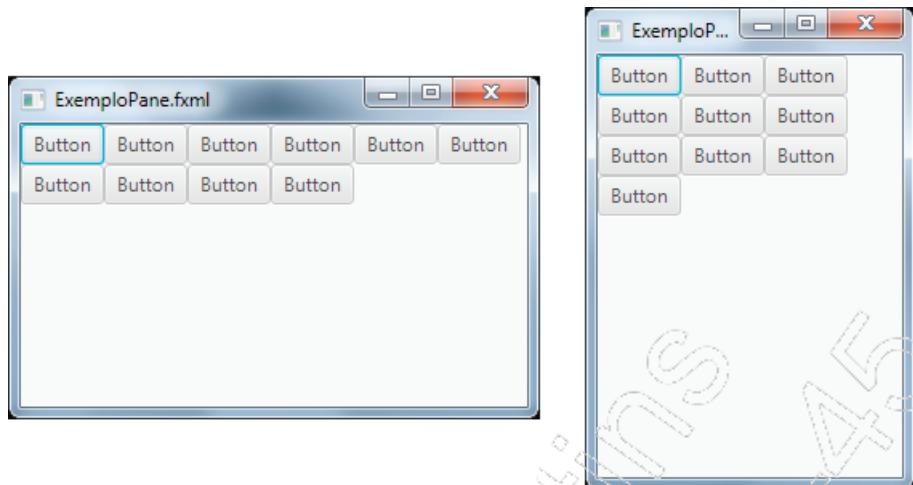
- **BorderPane**: Divide a tela em até cinco regiões, em que os elementos aderem às bordas da tela:



- **GridPane**: Divide a tela em uma região tabular com a quantidade desejada de linhas e colunas, formando um grid onde todos os elementos são uniformemente distribuídos quando a janela é redimensionada:



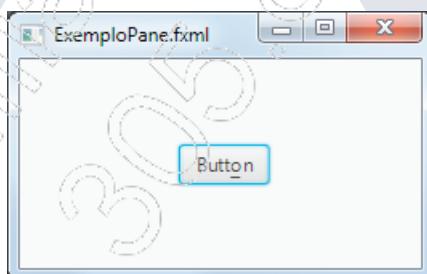
- **FlowPane**: Organiza os elementos na tela, um após o outro, horizontalmente. Elementos que não couberem, serão movidos para uma próxima linha. Desta forma, os elementos são disponibilizados como um texto de um livro:



Cada um destes painéis também funciona como um componente a ser inserido no painel raiz, contendo as suas propriedades isoladamente. Em outras palavras, é possível aninhar painéis de tipos diferentes combinando-os para formar layouts mais sofisticados.

3.3.4. Button

Um componente **Button** representa um botão em que o usuário pode clicar ou pressionar a barra de espaço para que algum evento seja disparado.

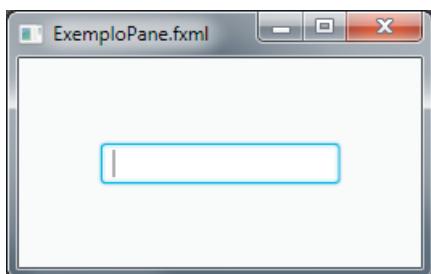


As principais propriedades deste componente são:

- **Text**: Define o rótulo a ser exibido pelo botão. Utilize o caractere “_” (underline) para definir a tecla de acesso. O underline deve preceder a letra desejada e a propriedade **Mnemonic Parsing** precisa estar habilitada;
- **Disable**: Permite desabilitar o botão, impedindo o usuário de clicá-lo;
- **Visible**: Permite ocultar o botão.

3.3.5. TextField

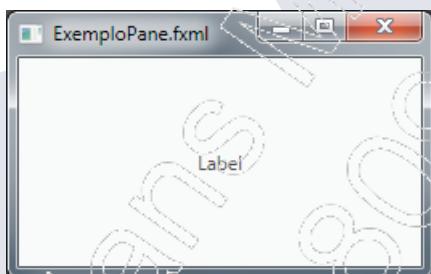
Este componente representa uma caixa de texto em que o usuário poderá digitar. Observe:



Sua principal propriedade é a **Text**, que permite definir um texto inicial a ser exibido pela caixa. Utilize os métodos **getText()** e **setText()** para manipular programaticamente o conteúdo digitado pelo usuário.

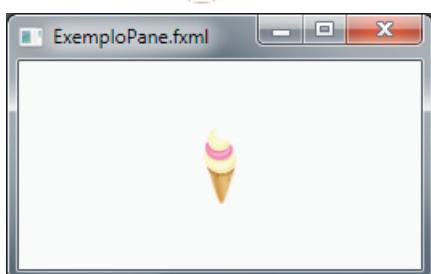
3.3.6. Label

Um rótulo utilizado na exibição de mensagens fixas em sua janela:



3.3.7. ImageView

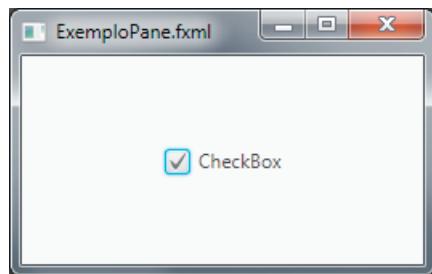
Permite adicionar uma imagem ou ícone em uma posição específica. Veja um exemplo:



Utilize a propriedade **Image** para selecionar a imagem a ser exibida.

3.3.8. CheckBox

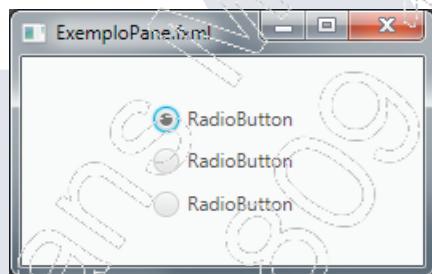
Componente que pode ser marcado ou desmarcado pelo usuário:



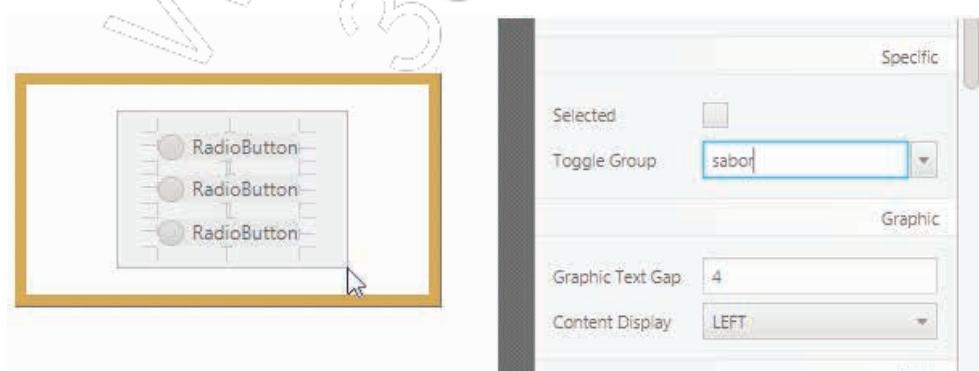
Utilize a propriedade **Selected** para que a **Checkbox** seja inicialmente exibida como marcada. Utilize os métodos **isSelected()** e **setSelected()** para verificar e assinalar programaticamente este componente como selecionado (checado).

3.3.9. RadioButton

Possui a mesma funcionalidade de uma **CheckBox**, porém, pode ser utilizado em grupos em que podemos ter a seleção exclusiva. Observe:



Para ter o efeito de seleção exclusiva, no **Scene Builder**, selecione todos os componentes de um mesmo grupo e digite um nome na propriedade **Toggle Group**:



3.4. Manipulação de eventos

A manipulação de eventos em sua janela é realizada por uma classe denominada **controladora**. Através desta classe, podemos controlar dinamicamente o comportamento, aparência e estado de seus componentes.

3.4.1. Identificadores

Cada componente a ser controlado dinamicamente deve possuir um identificador único dentro do arquivo FXML. Para assinalar o id (identificador) de um componente, basta selecionar o componente desejado e, na seção **Code** do **Inspector**, digitar o id desejado na caixa **fx:id**:



! Cada identificador de componente de sua janela será posteriormente transformado em um atributo da classe controladora. Portanto, utilize a mesma nomenclatura usada na criação de variáveis do Java.

3.4.2. Eventos

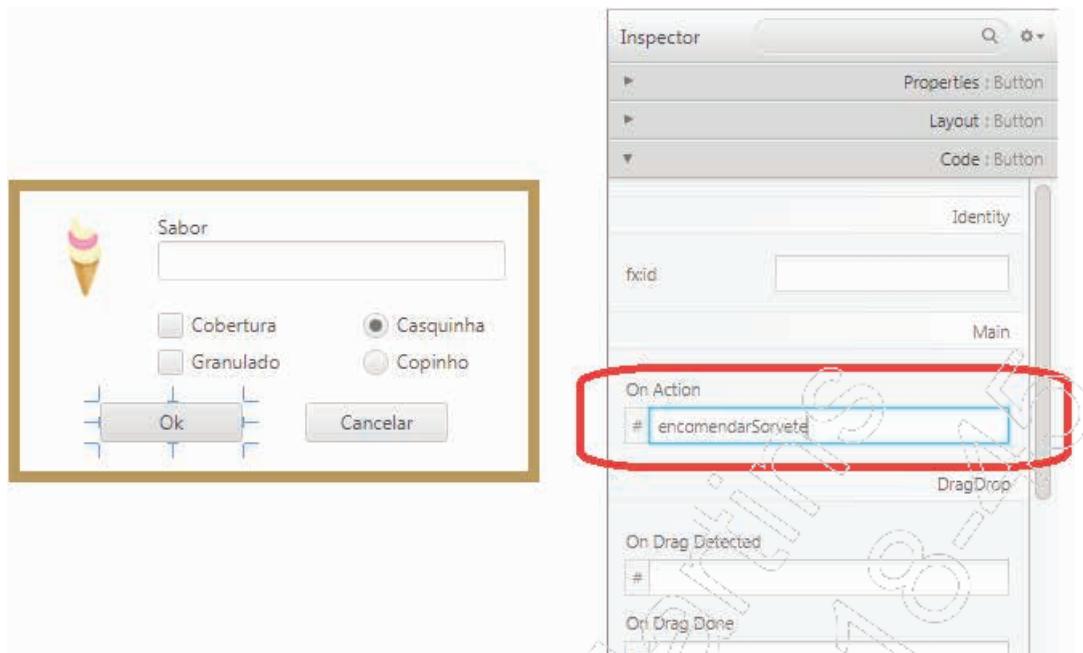
Os eventos são as ações realizadas sobre os componentes de uma interface gráfica, como um clique, pressionar de tecla, fechamento de janela etc.

Ao selecionar um componente de sua interface gráfica, o **Scene Builder** exibe a lista de eventos associados no painel **Code**.

Veja, na tabela a seguir, a descrição de alguns dos principais eventos:

Evento	Descrição
On Action	Executado ao clicar ou ativar (pressionar da tecla ENTER) sobre o componente.
On Mouse Clicked	Executado ao clicar sobre o componente.
On Mouse Entered	Executado ao passar o ponteiro sobre o componente.
On Mouse Exited	Executado ao sair com o ponteiro de cima do componente.
On Mouse Pressed	Executado ao baixar um dos botões do mouse sobre o componente.
On Mouse Released	Executado ao liberar o botão do mouse que havia sido baixado sobre o componente.
On Key Typed	Executado quando o usuário digita algum caractere sobre o componente.
On Key Pressed	Executado quando o usuário baixa alguma tecla sobre o componente.
On Key Released	Executado quando o usuário libera a tecla que foi baixada sobre o componente.

Para manipular um evento de algum dos elementos de sua interface gráfica, selecione o componente e, na seção **Code** do **Inspector**, procure pelo evento desejado. Digite um nome de identificador para este evento:



O identificador de evento será posteriormente transformado em um método da classe controladora. Portanto, utilize a mesma nomenclatura usada na criação de métodos do Java.

3.4.3. Classe controladora

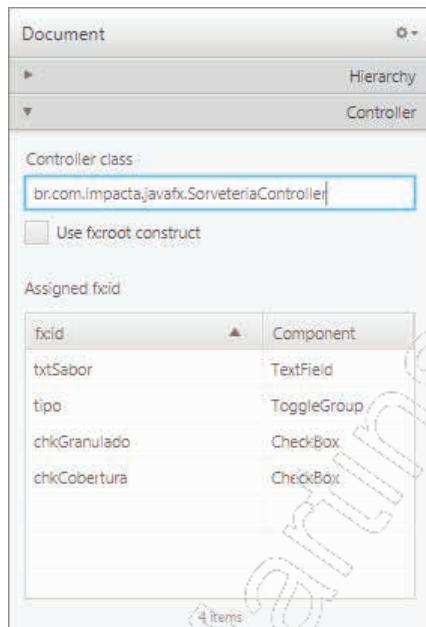
A classe controladora é uma simples classe criada pelo desenvolvedor e utilizada na manipulação dos eventos de uma janela.

Embora o **Scene Builder** não permita a elaboração de código Java diretamente, ele fornece uma maneira simplificada de desenvolver a classe controladora para a interface gráfica que está sendo editada.

Para criar a classe controladora, siga os passos adiante:

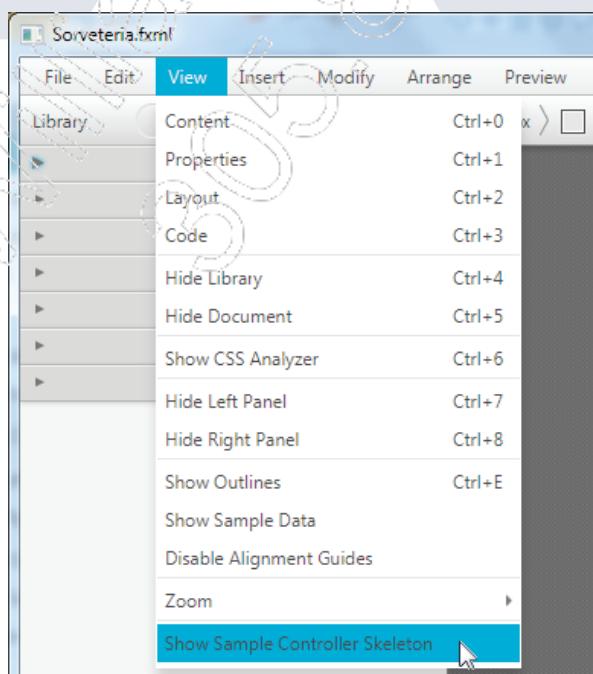
1. Elabore toda a interface gráfica, posicionando os elementos e assinalando as suas propriedades, conforme necessário;
2. Preencha o identificador de cada componente a ser tratado dinamicamente pela aplicação;
3. Assinale os identificadores para os eventos desejados;

4. No painel **Document** (lado inferior esquerdo do **Scene Builder**), selecione a aba **Controller** e preencha o campo **Controller class** com o nome da classe controladora a ser criada. Este deve ser o nome completo da classe, incluindo o package ao qual ela irá pertencer:

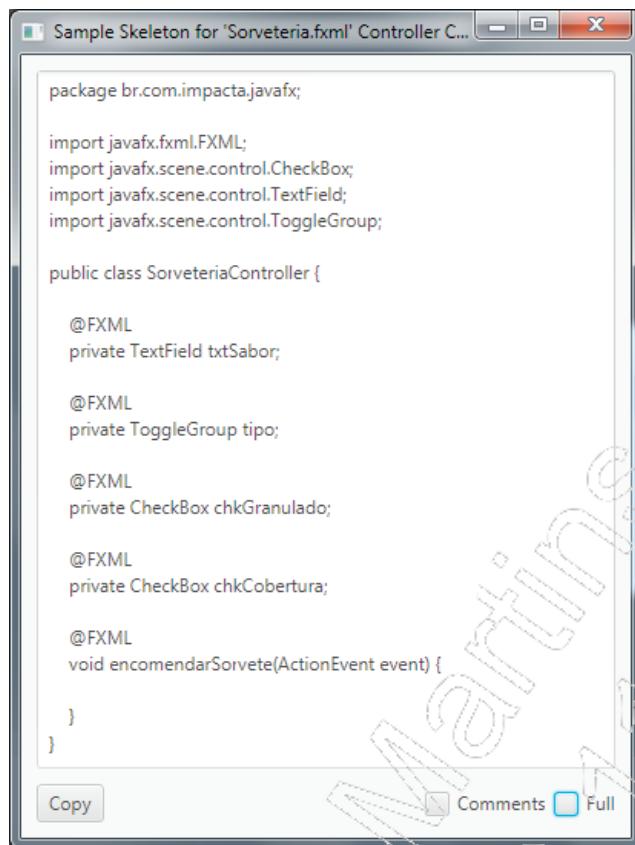


! Repare que a aba **Controller** exibe todos os identificadores assinalados para esta tela.

5. Clique no menu **View**, opção **Show Sample Controller Skeleton**:



Observe que o **Scene Builder** exibirá um template para a classe controladora:



A screenshot of a Java code editor showing a template for a JavaFX controller class. The window title is "Sample Skeleton for 'Sorveteria.fxml' Controller C...". The code includes imports for JavaFX FXML and scene controls, and defines a public class SorveteriaController with fields for a text field and a toggle group, and a method to handle an action event.

```
package br.com.impacta.javafx;

import javafx.fxml.FXML;
import javafx.scene.control.CheckBox;
import javafx.scene.control.TextField;
import javafx.scene.control.ToggleGroup;

public class SorveteriaController {

    @FXML
    private TextField txtSabor;

    @FXML
    private ToggleGroup tipo;

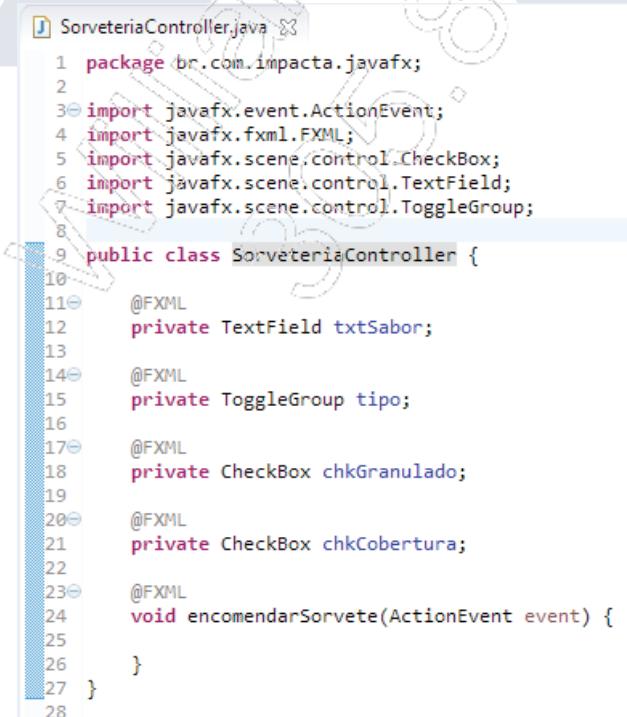
    @FXML
    private CheckBox chkGranulado;

    @FXML
    private CheckBox chkCobertura;

    @FXML
    void encomendarSorvete(ActionEvent event) {
    }
}
```

At the bottom of the editor, there are buttons for "Copy", "Comments", and "Full".

6. Crie uma classe com o mesmo nome e package especificados e cole o template gerado pelo **Scene Builder**:



A screenshot of a Java code editor showing the generated controller template. The file is named "SorveteriaController.java". The code is identical to the one shown in the Scene Builder screenshot above, including the package declaration, imports, class definition, and method.

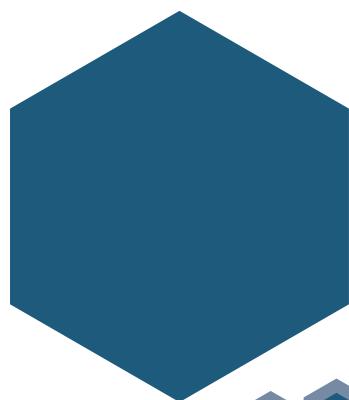
```
1 package br.com.impacta.javafx;
2
3 import javafx.event.ActionEvent;
4 import javafx.fxml.FXML;
5 import javafx.scene.control.CheckBox;
6 import javafx.scene.control.TextField;
7 import javafx.scene.control.ToggleGroup;
8
9 public class SorveteriaController {
10
11     @FXML
12     private TextField txtSabor;
13
14     @FXML
15     private ToggleGroup tipo;
16
17     @FXML
18     private CheckBox chkGranulado;
19
20     @FXML
21     private CheckBox chkCobertura;
22
23     @FXML
24     void encomendarSorvete(ActionEvent event) {
25
26     }
27
28 }
```

7. Implemente os métodos associados aos eventos de sua tela:

```
25@  
26  
27  
28     String descricao = txtSabor.getText() + " - "  
29         + ((RadioButton) tipo.getSelectedToggle()).getText();  
30  
31     if (chkCobertura.isSelected() && chkGranulado.isSelected()) {  
32         descricao += " - " + " Cobertura e Granulado";  
33     } else if (chkCobertura.isSelected()) {  
34         descricao += " - " + " Cobertura";  
35     } else if (chkGranulado.isSelected()) {  
36         descricao += " - " + " Granulado";  
37     }  
38  
39     ((Stage) txtSabor.getScene().getWindow()).setTitle(descricao);  
40 }
```

Desta forma, podemos perceber o total isolamento entre a interface gráfica (arquivos FXML) e as regras de negócios (chamadas a partir da classe controladora).

Utilizando estes conceitos, podemos desenvolver diversos tipos de aplicações para ambiente desktop, como aplicações cliente/servidor com acesso à base de dados, inteiramente baseadas no JavaFX.



Apêndice IV

Java em módulos - Jigsaw



Editora
IMPACTA



4.1. Dependências - Como funciona hoje

A JVM busca por classes necessárias à execução de uma aplicação de forma dinâmica e numa sequência que não dá para ser estabelecida antecipadamente. Assim, no caso de qualquer problema que ocorrer neste processo de busca, deve-se levantar exceções enquanto o sistema já estiver executando.

Os principais problemas nesta abordagem podem se resumir a dois mais clássicos:

- Não encontrar uma classe no classpath definido;
- Diferentes versões de uma mesma biblioteca ocasionando conflitos entre classes com mesmo nome e pacote.

4.2. Introdução aos módulos

Visando resolver esse e outros problemas recorrentes e também procurando otimizar tal processo, a partir do Java 9 surge o **Java Platform Module System (JPMS)**, conhecido na comunidade como projeto **Jigsaw**.

Com esta abordagem, apareceu o conceito de módulos, que proporcionou uma forma de que todas as dependências que o desenvolvedor gerou como módulo possam ser verificadas e definidas em tempo de compilação.

Todo o JDK foi reestruturado para o novo formato de módulos, porém de uma forma ainda pouco invasiva quanto à antiga forma de classpaths já utilizada em versões anteriores.

Para conhecer com profundidade a proposta e a nova estrutura de projetos Java, acesse a documentação oficial:

<http://openjdk.java.net/projects/jigsaw/spec/sotms/>