

Guia Prático para Certificação

**Atualize sua certificação
Java 6 para Java 8**

Rinaldo Pitzer Jr. e Rodrigo Moutinho

Guia Prático para Certificação

Atualize sua certificação Java 6 para Java 8

Rinaldo Pitzer Jr., Rodrigo Moutinho

Version v1.0.0, 2019-09-03

Índice

Licença	1
Contribuidores	2
Começando	3
Introdução	3
Objetivos	3
Language Enhancements	4
Objetos Strings	4
Try com recursos	10
Múltiplas Exception no mesmo catch	18
Métodos static e default em Interfaces	21
Localization	29
Localização (Locale)	29
Pacote de Recursos (Resource Bundle)	35
Data e Hora	45
Formatação de Números e Data	66
Fusos Horários	82
Lambda	88
Interfaces Funcionais (Functional Interfaces)	88
Expressões Lambda (Lambda Expression)	93
Interfaces Funcionais Pré-Construídas (Built-in Interfaces)	101
Referências a Métodos (Method Reference)	113
Java Streams	120
Utilizando Streams	120
Streams Paralelos	140
Concurrency	147
Pacote Concurrent	147
Locks	151
Executando tarefas com ExecutorService	160
Framework Fork/Join	176
Java File I/O (NIO.2)	181
Paths	181
Files	191
DirectoryStream e FileVisitor	207
Files com Streams	215
WatchService	221
Java Collections	227
Diamond Operator (Operador Diamante)	227
Collections e lambda	228

Buscar por dados	229
Fazendo cálculos e coletando resultados de Streams	231
Melhorias de Java 8 em Coleções	233
Maps e Streams	235
Assume the following	238
Apêndice A: Dicas para ter sucesso na prova!	239
Cenário 1	239
Cenário 2	240
Cenário 3	240
Apêndice B: Teste seu conhecimento!	242
Gratuito	242
Pagos	242
Apêndice C: Referências	243
Material Complementar	243

Licença

MIT License

Copyright (c) 2019 Rinaldo Pitzer Jr. e Rodrigo Moutinho

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Contribuidores

Por ser um livro de código aberto, recebemos várias mudanças de conteúdo e erratas ao longo de seu desenvolvimento. Aqui estão todas as pessoas que contribuíram para a versão em português do livro como um projeto de código aberto. Obrigado a todos por ajudarem a tornar este livro melhor para todos.

Rinaldo Pitzer Júnior

Rodrigo Moutinho

semantic-release-bot

Esse texto é praticamente o mesmo utilizado no **Pro Git 2**. Acesse o texto original [aqui](#).

Começando

Introdução

Este projeto serve como material de apoio na realização do exame [1Z0-813](#) que atualiza qualquer profissional com certificação Java 6 ou inferior, para a versão 8. No momento desta documentação, o *voucher* do exame custa R\$ 597,00 no Brasil.

Objetivos

Como guia para criação deste livro foi utilizado todos os objetivos citados na seção "*Review Exam Topics*", de acordo com o [site da certificação](#).

A missão deste livro é criar o maior número de exemplos práticos possíveis para ajudar você a colocar a mão na massa. Quanto maior for seu contato com os códigos da versão 8 do Java, mais confiante estará na hora do exame, e também para lidar com projetos em Java 8 no mercado de trabalho.

Language Enhancements

Objetos Strings

Objetivo

Develop code that uses String objects in the switch statement, binary literals, and numeric literals, including underscores in literals.

- Desenvolver código que utilize objetos String em instruções Switch, binários literais, e numéricos literais, incluindo underscore (_) em literais.

String em instruções Switch

É esperado que o candidato saiba compreender e analisar o uso de Strings em instruções `switch`, como no seguinte exemplo.

`src/org/j6toj8/languageenhancements/stringinswitch/StringInSwitch_Complete.java`

```
public static void main(String[] args) {  
  
    String mes = "jan";  
  
    switch (mes) {  
        case "jan":  
            System.out.println("Janeiro");  
            break;  
        case "fev":  
            System.out.println("Fevereiro");  
            break;  
        case "mar":  
            System.out.println("Março");  
            break;  
        default:  
            break;  
    }  
}
```

Apesar da certificação ter foco nas atualizações trazidas pelo Java 7 e 8, é esperado que o candidato entenda também conceitos de versões anteriores do Java. Por isso, serão apresentadas algumas regras que talvez você já conheça sobre `switch`, mas utilizando `String` no `switch`.

1. Todo `case` deve ser único, não pode se repetir.
2. O `default` pode aparecer em qualquer posição no `switch`.

```
public static void main(String[] args) {  
  
    String mes = "jan";  
  
    switch (mes) {  
        case "jan":  
            System.out.println("Janeiro");  
            break;  
        default: // COMPILA - O default pode estar em qualquer posição  
            break;  
        case "jan": // NÃO COMPILA - Já existe o case "jan"  
            System.out.println("Janeiro2");  
            break;  
        case "mar":  
            System.out.println("Março");  
            break;  
    }  
}
```

3. Tipos suportados em `switch`.

- int e Integer
- byte e Byte
- short e Short
- char e Character
- String
- valores de Enums

4. Tipos não suportados em `switch`.

src/org/j6toj8/languageenhancements/stringinswitch/StringInSwitch_Type.java

```
public static void main(String[] args) {  
  
    Long mes = 1L;  
  
    switch (mes) { // NÃO COMPILA - Long não é um tipo suportado  
        case 1L:  
            System.out.println("Janeiro");  
            break;  
        case 2L:  
            System.out.println("Fevereiro");  
            break;  
        default:  
            break;  
    }  
}
```

5. A execução se inicia em um **case** e somente para ao encontrar um **break**.

src/org/j6toj8/languageenhancements/stringinswitch/StringInSwitch_Break.java

```
public static void main(String[] args) {  
  
    String mes = "jan";  
  
    switch (mes) {  
        case "jan":  
            System.out.println("Janeiro");  
        default:  
            System.out.println("Não é um mês");  
        case "fev":  
            System.out.println("Fevereiro");  
            break;  
        case "mar":  
            System.out.println("Março");  
            break;  
    }  
}
```

saída no console

```
Janeiro  
Não é um mês  
Fevereiro
```

Nesse caso a execução inicia no **case "jan"**, passar pelo **default** e pelo **case "fev"** até parar no **break**, por isso as 3 strings aparecem no console.

6. Um **switch** vazio é válido, mesmo que não tenha utilidade.

src/org/j6toj8/languageenhancements/stringinswitch/StringInSwitch_Empty.java

```
public static void main(String[] args) {  
  
    String mes = "jan";  
    switch (mes) {} // COMPILA - switch pode estar vazio, mesmo que seja inútil  
}
```

7. Todos os valores de **case** precisam ser constantes, ou seja, variáveis finais em tempo de compilação. Se o valor do **case** puder mudar em tempo de execução, o código não compila.

src/org/j6toj8/languageenhancements/stringinswitch/StringInSwitch_ConstantOnly.java

```
private static final String FEV = "fev";  
private static String jan = "jan";  
  
public static void getNomeMes(final String mai) {  
  
    String mes = "jan";  
  
    final String mar = "mar";  
    String abr = "abr";  
  
    switch (mes) {  
        case jan: // NÃO COMPILA - jan é um atributo comum, pode mudar em tempo de execução  
            System.out.println("Janeiro");  
            break;  
        case FEV: // COMPILA - FEV é uma constante em tempo de compilação, seu valor nunca muda  
            System.out.println("Fevereiro");  
            break;  
        case mar: // COMPILA - mar é uma constante em tempo de compilação, seu valor nunca muda  
            System.out.println("Março");  
            break;  
        case abr: // NÃO COMPILA - abr é uma variável comum, pode mudar em tempo de execução  
            System.out.println("Março");  
            break;  
        case mai: // NÃO COMPILA - mai é final, mas não é constante, pode mudar em tempo de execução  
            System.out.println("Março");  
            break;  
    }  
}
```

Pronto, essas são as regras de `switch`. Você provavelmente já conhece algumas referentes à versões anteriores do Java, mas agora você as viu em `switch` que utilizam Strings. Isso não era possível antes do Java 7.

Literais Binários e Numéricos, incluindo underscore(_)

É esperado que o candidato saiba compreender e analisar o uso de literais binários e numéricos, como no seguinte exemplo.

`src/org/j6toj8/languageenhancements/literals/Literals_Complete.java`

```
int i1 = 1; // int
int i2 = 1_000_000; // int com underscore
int i3 = 0567; // octadecimal
int i4 = 0xFA1; // hexadecimal
int i5 = 0b0101; // binário

long l1 = 1L; // long com L
long l2 = 1l; // long com l
long l3 = 12_345_6_7890_123456_789L; // long com underscore
long l4 = 0xFA1L; // long hexadecimal

double d1 = 1.00; // double
double d2 = 100_000.01; // double com underscore
double d3 = 1D; // double com D
double d4 = 3.1E2; // notação científica = 3.1 * 10^2 = 3.1 * 100 = 310.0

float f1 = 1.00F; // float
```

Apesar da certificação ter foco nas atualizações trazidas pelo Java 7 e 8, é esperado que o candidato entenda também conceitos de versões anteriores do Java. Por isso, serão apresentadas algumas regras que talvez você já conheça sobre literais.

1. No Java, *Literal* é qualquer número escrito diretamente no código, como todos do exemplo acima.
2. Por padrão, o Java interpreta literais como `int`. Ou seja, se não houver um sufixo no número para mudar seu tipo, ele é um `int`.

`src/org/j6toj8/languageenhancements/literals/Literals_Suffix.java`

```
int i1 = 1; // por padrão é int
long l1 = 1L; // com um L no final, é um long
double d1 = 1.0;
double d2 = 1.0D; // com ou sem D no final, se tiver casa decimal é um double por
// padrão
float f1 = 1.0F; // com um F no final, é um float
```

3. Por padrão, o Java interpreta literais como sendo decimais. Existem prefixos que mudam o sistema numérico do literal.

src/org/j6toj8/languageenhancements/literals/Literals_Prefix.java

```
int i1 = 0567; // octadecimal - base 8 - começa com 0
int i2 = 0xFA1; // hexadecimal - base 16 - começa com 0x
int i3 = 0b0101; // binário - base 2 - começa com 0b

long l1 = 0xABCL; // long também pode ser hexadecimal - começa com 0x e termina com L
```

4. A partir do Java 7, é possível utilizar underscore (_) para separar visualmente um número. Isso não muda o valor do número, e serve apenas para tornar o código mais legível.

src/org/j6toj8/languageenhancements/literals/Literals_Underscore.java

```
int i1 = 1_000_000; // int com underscore - é o mesmo que escrever 1000000
int i2 = 10_00_00_0; // o underscore pode estar em qualquer posição entre 2 números
int i3 = _1000; // NÃO COMPILA - o underscore não pode estar no início
int i4 = 1000_; // NÃO COMPILA - o underscore não pode estar no final
int i5 = 1__000; // COMPILA - vários underscore é permitido, desde que estejam entre 2 números
int i6 = 0x_100; // NÃO COMPILA - entre marcador de base não é permitido
int i7 = 0xF_F; // COMPILA - apesar de serem letras, representam valores numéricos dessa base

long l1 = 12_345_6_7890_123456_789L; // long com underscore
long l2 = 12_345_6_789_L; // NÃO COMPILA - não pode ficar ao lado de um marcador de tipo

double d1 = 100_000.01; // double com underscore
double d2 = 10_.01; // NÃO COMPILA - o underscore deve estar entre números
double d3 = 10._01; // NÃO COMPILA - o underscore deve estar entre números
```

Referências

Strings em Switch

- Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 598). Wiley. Edição do Kindle.
- [Strings in switch Statements](#). Java Documentation.
- [New Java 7 Feature: String in Switch support](#). DZone.

Literais

- Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 597). Wiley. Edição do Kindle.
- [Java/Literais](#). Wikibooks.

Try com recursos

Objetivo

Develop code that uses try-with-resources statements, including using classes that implement the AutoCloseable interface.

- Desenvolver código que utilize instruções try-with-resources, incluindo o uso de classes que implementam a interface AutoCloseable.

É esperado que o candidato saiba compreender e analisar o uso da instrução *try-with-resources*, incluindo classes que implementam a interface *AutoClosable*.

Antes de continuar, com base no exemplo a seguir, entenda a execução do método `main` e o que é apresentado no console após sua execução.

`src/org/j6toj8/languageenhancements/trywithresources/TryWithResouces_Complete.java`

```
public static void main(String[] args) throws IOException {
    // criação de 2 arquivos
    File file = new File("arquivo.txt");
    File file2 = new File("arquivo2.txt");

    // Exemplo try-with-resources com PrintWriter
    try (PrintWriter writer = new PrintWriter(file)) {
        // escreve no arquivo.txt
        writer.println("Olá Mundo!");
    }

    // Exemplo try-with-resources com BufferedReader
    try (BufferedReader reader = Files.newBufferedReader(file.toPath())) {
        // imprime no console uma linha do arquivo.txt
        System.out.println(reader.readLine());
    }

    // Exemplo try-with-resources com BufferedReader e BufferedWriter
    try (BufferedReader reader = Files.newBufferedReader(file.toPath());
         BufferedWriter writer = Files.newBufferedWriter(file2.toPath())) {
        // lê a linha do arquivo.txt e escreve no arquivo2.txt
        writer.write(reader.readLine() + "2");
    }

    // Exemplo try-with-resources com BufferedReader
    try (BufferedReader reader = Files.newBufferedReader(file2.toPath())) {
        // imprime no console uma linha do arquivo2.txt
        System.out.println(reader.readLine());
    }
    // todos os Reader e Writer já foram fechados.
}
```

O código acima utiliza dois arquivos, e faz leitura e escrita neles. A grande novidade desse código é a declaração de variáveis dentro de parênteses após a instrução `try`. Isso é a sintaxe chamada *try-with-resources*, e ela chama automaticamente o método `close()` dos recursos que estão sendo utilizados. Até o Java 6, seria necessário chamar o `close()` manualmente, como no exemplo abaixo.

src/org/j6toj8/languageenhancements/trywithresources/TryWithResouces_Java6.java

```
public static void main(String[] args) throws FileNotFoundException {
    File file = new File("arquivo.txt");
    PrintWriter writer = null;
    try {
        writer = new PrintWriter(file);
        writer.println("Olá Mundo!");
    } finally {
        if (writer != null) {
            writer.close(); // fechando o writer manualmente
        }
    }
}
```

1. A instrução *try-with-resources* fecha automaticamente os recursos que implementam a interface `AutoCloseable`.
2. Ela não precisa de `catch` nem `finally` explícitos.

src/org/j6toj8/languageenhancements/trywithresources/TryWithResouces_AutoCloseable.java

```
static class Porta implements AutoCloseable {
    @Override
    public void close() { // chamado automaticamente pelo try-with-resources
        System.out.println("Porta fechada.");
    }
}

public static void main(String[] args) throws FileNotFoundException {
    try (Porta porta = new Porta()) { // Porta instanciada dentro da instrução try-with-resources
        System.out.println("try");
    }
}
```

Saída no console

```
try
Porta fechada.
```

3. A instrução *try-with-resources* ainda pode ter `catch` e `finally`, apesar de não ser necessário. Nesse caso, os recursos são fechados depois do `try` e antes de qualquer `catch` ou `finally`.
4. O método `close` pode lançar uma exceção sendo capturada pelo `catch`, caso exista.

```
static class Porta implements AutoCloseable {
    @Override
    public void close() throws Exception { // chamado automaticamente pelo try-with-
resources
        System.out.println("Porta fechada.");
        throw new Exception(); // lança Exception
    }
}

public static void main(String[] args) throws FileNotFoundException {
    try (Porta porta = new Porta()) { // Porta instanciada dentro da instrução try-
with-resources
        System.out.println("try");
    } catch (Exception e) {
        System.out.println("catch");
    } finally {
        System.out.println("finally");
    }
}
```

Saída no console

```
try
Porta fechada.
catch
finally
```

Ou seja, primeiro o **try** é chamado. Logo depois é chamado o método **close()** que ao final lança uma exceção. O **catch** captura essa exceção. E finalmente o **finally** é chamado.

5. Os recursos declarados na instrução *try-with-resources* são fechados na ordem inversa da declaração.

```
static class Porta implements AutoCloseable {
    @Override
    public void close() { // chamado automaticamente pelo try-with-resources
        System.out.println("Porta fechada.");
    }
}

static class Gaveta implements AutoCloseable {
    @Override
    public void close() { // chamado automaticamente pelo try-with-resources
        System.out.println("Gaveta fechada.");
    }
}

public static void main(String[] args) {
    try (Porta porta = new Porta();
         Gaveta gaveta = new Gaveta()) {
        System.out.println("Olá.");
    }
}
```

Saída no console

```
Olá.
Gaveta fechada.
Porta fechada.
```

Ou seja, como a ordem de declaração dentro do *try-with-resources* foi **Porta** e depois **Gaveta**, a ordem de chamada do método **close** é inversa: **Gaveta** e depois **Porta**.

6. Os recursos declarados no *try-with-resources* só estão disponível dentro do bloco **try**.

```
static class Porta implements AutoCloseable {
    @Override
    public void close() { // chamado automaticamente pelo try-with-resources
        System.out.println("Porta fechada.");
    }
}

public static void main(String[] args) {
    try (Porta porta = new Porta()) {
        porta.toString();
    } catch (Exception e) {
        porta.toString(); // NÃO COMPILA - variável porta só disponível dentro do bloco
    try
    } finally {
        porta.toString(); // NÃO COMPILA - variável porta só disponível dentro do bloco
    try
    }
}
```

7. Somente classes que implementam `AutoCloseable` podem ser utilizadas dentro do *try-with-resources*.

```
static class Prateleira {}

public static void main(String[] args) {
    try (Prateleira prateleira = new Prateleira()) { // NÃO COMPILA - Prateleira não
    implementa AutoClosable
        System.out.println("Olá");
    }
}
```

8. Caso o método `close()` lance uma exceção checada (ou seja, que herda de `Exception`), o código só compila se existir um `catch` que capture aquela exceção, ou o método declare o `throws`.

```
static class Porta implements AutoCloseable {
    @Override
    public void close() throws Exception { // declara throws Exception
        obrigatoriamente
        throw new Exception();
    }
}

void try1() {
    try (Porta porta = new Porta()) { // NÃO COMPILA - exceção do close() não é
        capturada nem declarada
        System.out.println("Olá 1");
    }
}

void try2() {
    try (Porta porta = new Porta()) {
        System.out.println("Olá 2");
    } catch (Exception e) { // COMPILA - exceção capturada
    }
}

void try3() throws Exception { // COMPILA - exceção declarada no método
    try (Porta porta = new Porta()) {
        System.out.println("Olá 3");
    }
}
```

9. O Java 5 já possuía uma interface chamada `Closeable`, porém ela permite lançar apenas `IOException`. A nova interface `AutoCloseable` permite lançar qualquer exceção. Como `Closeable` atende a implementação de `AutoCloseable`, ela agora estende `AutoCloseable`. Logo, todas as classes que já implementavam `Closeable` podem ser utilizadas dentro do *try-with-resources*. Veja abaixo como era a interface `Closeable` antes e a partir do Java 7:

Antes do Java 7

```
public interface Closeable {
    public void close() throws IOException;
}
```

A partir do Java 7

```
public interface Closeable extends AutoCloseable {
    public void close() throws IOException;
}
```

10. Um comportamento novo são as exceções suprimidas (suppressed). Se ambos o bloco `try` e o

método `close` lançam exceção, a do `close` fica suprimida, pois a do `try` é lançada primeiro.

`src/org/j6toj8/languageenhancements/trywithresources/TryWithResources_Suppressed.java`

```
static class Porta implements AutoCloseable {
    @Override
    public void close() {
        System.out.println("close");
        throw new RuntimeException("erro no close"); // lança RuntimeException, mas só
depois do try
    }
}

public static void main(String[] args) {
    try (Porta porta = new Porta()) {
        System.out.println("try");
        throw new RuntimeException("erro no try"); // lança RuntimeException
    } catch (RuntimeException e) { // captura RuntimeException - qual foi capturada?
        System.out.println(e.getMessage()); // apresenta a mensagem da exceção do try
        System.out.println(e.getSuppressed()[0].getMessage()); // apresenta a mensagem
da exceção suprimida, ou seja, do close
    }
}
```

Saída no console

```
try
close
erro no try
erro no close
```

Ou seja, a exceção que de fato foi capturada foi a do bloco `try`, pois foi lançada primeiro. A exceção lançada pelo método `close` ficou suprimida, e fica disponível em um array no método `getSuppressed()` da exceção.

11. E por fim, é necessário lembrar que a instrução `try` "comum" ainda precisa obrigatoriamente de um `catch` ou `finally`.

```
public static void main(String[] args) {  
    try {  
        System.out.println("try");  
    } // NÃO COMPILA - try "comum" sem catch ou finally  
  
    try {  
        System.out.println("try");  
    } catch (Exception e) {  
    } // COMPILA - try "comum" só com catch  
  
    try {  
        System.out.println("try");  
    } finally {  
    } // COMPILA - try "comum" só com finally  
  
    try {  
        System.out.println("try");  
    } catch (Exception e) {  
    } finally {  
    } // COMPILA - try "comum" com catch e finally  
}
```

Alguns tipos que implementam *Closeable*

- `InputStream` e suas subclasses (`FileInputStream`, `ObjectInputStream`, etc)
- `OutputStream` e suas subclasses (`ByteArrayOutputStream`, `FileOutputStream`, etc)
- `Reader` e suas subclasses (`BufferedReader`, `CharSequenceReader`)
- `Writer` e suas subclasses (`BufferedWriter`, `PrintWriter`, etc)

Referências

- Using Try-With-Resources

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 296). Wiley. Edição do Kindle.

- Java – Try with Resources.
- The `try-with-resources Statement`. Java Documentation.
- [Como funciona o try-with-resources?](#)

Múltiplas Exception no mesmo catch

Objetivo

Develop code that handles multiple Exception types in a single catch block.

-

Desenvolver código que lide com múltiplos tipos de Exception em um único bloco catch.

É esperado que o candidato saiba compreender e analisar o uso da instrução *try-catch* com múltiplos tipos de **Exception** no mesmo bloco **catch**.

Antes de continuar, com base no exemplo a seguir, entenda a execução do método **main** e o que é apresentado no console após sua execução.

src/org/j6toj8/languageenhancements/multipleexception/MultipleException_Complete.java

```
public static void main(String[] args) {  
  
    try {  
        throw new NullPointerException();  
    } catch (NullPointerException | IllegalArgumentException | IllegalStateException e)  
    {  
        System.out.println("Exceção capturada: " + e);  
    } catch (Exception e) {  
        System.out.println("Exceção capturada: " + e);  
    }  
}
```

O código anterior possui um bloco *try-catch* que você provavelmente já conhece. A novidade neste código está no primeiro bloco **catch**, onde várias exceções são declaradas e capturadas ao mesmo tempo.

Saída no console

Exceção capturada: java.lang.NullPointerException

1. Desde o Java 7, múltiplas exceções podem ser capturadas no mesmo **catch**.
2. Apenas uma variável é permitida para um bloco **catch**, e deve estar localizada no final.

```
public static void main(String[] args) {  
  
    try {  
        throw new NullPointerException();  
    } catch (NullPointerException | IllegalArgumentException e) { // COMPILA - m  
    últiplas exceções no mesmo catch, só uma variável no final  
        System.out.println("Exceção capturada: " + e);  
    } catch (IllegalStateException ise | UnsupportedOperationException uoe) { // NÃO  
    COMPILA - mas de uma variável declarada  
        System.out.println("Exceção capturada: " + ise);  
    } catch (ClassCastException cce | ConcurrentModificationException) { // NÃO  
    COMPILA - só uma variável, mas no lugar errado  
        System.out.println("Exceção capturada: " + cce);  
    }  
}
```

3. Não é permitido declarar exceções diferentes, mas que seriam redundantes levando em consideração a herança.

```
public static void main(String[] args) {  
  
    try {  
        throw new NullPointerException();  
    } catch (RuntimeException | IllegalArgumentException e) { // NÃO COMPILA -  
    IllegalArgumentException herda de RuntimeException, logo seria redundante  
        System.out.println("Exceção capturada: " + e);  
    }  
}
```

4. Ao fazer **catch** de múltiplas **Exception**, não é permitido sobrescrever a variável da exceção. Mas é possível se for apenas uma **Exception** no **catch**.

src/org/j6toj8/languageenhancements/multipleexception/MultipleException_OverrideVar.java

```
public static void main(String[] args) {  
  
    try {  
        throw new NullPointerException();  
    } catch (NullPointerException | IllegalArgumentException e) {  
        e = new IllegalStateException(); // NÃO COMPILA - a variável não pode ser  
        // sobrescrita quando está em um multi-catch  
    } catch (Exception e) {  
        e = new IllegalStateException(); // COMPILA - ainda é possível sobrescrever a  
        // variável quando não é um multi-catch  
    }  
}
```

5. Assim como nas versões anteriores, tipos mais genéricos de `Exception` devem vir depois, mas abaixo nos *catch*'s.

src/org/j6toj8/languageenhancements/multipleexception/MultipleException_GenericsLower.java

```
public static void main(String[] args) {  
  
    try {  
        throw new NullPointerException();  
    } catch (Exception e) {  
        System.out.println("Exceção capturada: " + e);  
    } catch (NullPointerException | IllegalArgumentException e) { // NÃO COMPILA -  
        // NullPointerException é mais específico que Exception, logo deveria ser capturada  
        // antes de Exception  
        System.out.println("Exceção capturada: " + e);  
    }  
}
```

6. Assim como nas versões anteriores, Exceções repetidas ainda são proibidas.

src/org/j6toj8/languageenhancements/multipleexception/MultipleException_RepeatException.java

```
public static void main(String[] args) {  
  
    try {  
        throw new NullPointerException();  
    } catch (NullPointerException | IllegalArgumentException e) {  
        System.out.println("Exceção capturada: " + e);  
    } catch (IllegalStateException | NullPointerException e) { // NÃO COMPILA -  
        // NullPointerException já foi capturada no catch anterior  
        System.out.println("Exceção capturada: " + e);  
    }  
}
```

7. Assim como nas versões anterior, Exceções checadas (aqueles que herdam de `Exception`) só podem estar em um `catch` caso algo no `try` lance elas.

`src/org/j6toj8/languageenhancements/multipleexception/MultipleException_CheckedException.java`

```
public static void main(String[] args) {  
  
    try {  
        throw new NullPointerException();  
    } catch (NullPointerException | IOException e) { // NÃO COMPILA - IOException não é lançada dentro do bloco try  
        System.out.println("Exceção capturada: " + e);  
    }  
}
```

Referências

- Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 291). Wiley. Edição do Kindle.
- [Catching Multiple Exception Types and Rethrowing Exceptions with Improved Type Checking](#). Java Documentation.

Métodos `static` e `default` em Interfaces

Objetivo

Use static and default methods of an interface including inheritance rules for a default method.

-

Usar métodos `static` e `default` de uma interface, incluindo regras de herança para um método `default`.

É esperado que o candidato saiba compreender e analisar o uso da dos modificadores `static` e `default` em métodos de interfaces.

Antes de continuar, com base no exemplo a seguir, entenda a execução do método `main` e o que é apresentado no console após sua execução.

```
interface Corredor {  
    static double calculeVelocidade(int distancia, int tempo) {  
        return distancia / tempo;  
    }  
  
    default String correr() {  
        return "Correndo";  
    }  
  
    String correrRapido();  
}  
  
static class Pessoa implements Corredor {  
    @Override  
    public String correrRapido() {  
        return "Pessoa Correndo Rápido";  
    }  
  
    public static void main(String[] args) throws IOException {  
        System.out.println(Corredor.calculeVelocidade(100, 10));  
        System.out.println(new Pessoa().correr());  
        System.out.println(new Pessoa().correrRapido());  
    }  
}
```

Saída no console

```
10.0  
Correndo  
Pessoa Correndo Rápido
```

O código anterior possui dois modificadores novos para interfaces, possíveis desde o Java 8: **default** e **static**. É possível perceber que esses dois métodos possuem corpo, algo que não era possível antes em uma interface. Então, vamos entender quais são as novas possibilidades.

1. Desde o Java 8, Interfaces podem ter métodos com o modificador **static**.
2. Métodos com o modificador **static** em interfaces são chamados iguais aos de uma classe comum, ou seja, não fazem parte da API da interface. Dessa forma, não são herdados pelas classes que implementam essa interface.

src/org/j6toj8/languageenhancements/staticdefaultininterfaces/StaticDefaultInInterfaces_Static.java

```
interface Corredor {  
    static double calculeVelocidade(int distancia, int tempo) {  
        return distancia / tempo;  
    }  
}  
  
static class Pessoa implements Corredor {  
    public static void main(String[] args) throws IOException {  
        System.out.println(Corredor.calculeVelocidade(100, 50)); // COMPILA - método  
        static de uma interface sendo chamado como se fosse de uma classe comum  
        System.out.println(Pessoa.calculeVelocidade(100, 50)); // NÃO COMPILA - o m  
        étodo static não é herdado, nem implementado, pela classe Pessoa  
    }  
}
```

3. Desde o Java 8, Interfaces podem ter métodos com o modificador **default**.
4. Métodos **default** não precisam, mas podem, ser sobrescritos.

src/org/j6toj8/languageenhancements/staticdefaultininterfaces/StaticDefaultInInterfaces_Default.java

```
interface Corredor {  
    default String correr() {  
        return "Correndo";  
    }  
}  
  
static class Pessoa implements Corredor {  
}  
  
static class Cavalo implements Corredor {  
    @Override  
    public String correr() {  
        return "Galopando";  
    }  
  
    public static void main(String[] args) throws IOException {  
        System.out.println(new Pessoa().correr());  
        System.out.println(new Cavalo().correr());  
    }  
}
```

Veja que a classe **Pessoa** não sobrescreve o método **correr()**, mantendo o comportamento padrão da implementação feita na interface **Corredor**.

A classe **Cavalo**, por outro lado, sobrescreve o método **correr()** para ter sua própria implementação.

Saída no console

```
Correndo  
Galopando
```

5. Assim como os outros método de uma interface, os métodos **static** e **default** **são sempre public**, e não podem ser modificados para **private** ou **protected**.

src/org/j6toj8/languageenhancements/staticdefaultininterfaces/StaticDefaultInInterfaces_AccessModifiers.java

```
interface Corredor {  
    default String correr() { // COMPILA - não há modificador de acesso declarado, é  
    automaticamente público  
        return "Correndo";  
    }  
    public default String correrRapido() { // COMPILA - modificador de acesso público  
    explícito  
        return "Correndo Rápido";  
    }  
    protected default String correrDevagar() { // NÃO COMPILA - o método deve ser  
    obrigatoriamente público  
        return "Correndo Devagar";  
    }  
    private default String correrExtremo() { // NÃO COMPILA - o método deve ser  
    obrigatoriamente público  
        return "Correndo ao Extremo";  
    }  
  
    private static double calculeVelocidade(int d, int t) { // NÃO COMPILA - o m  
    étodo deve ser obrigatoriamente público  
        return d / t;  
    }  
}
```

6. Diferente dos outros método de uma interface, os métodos **static** e **default** **não são abstract**, e também não podem ser. Afinal, eles possuem implementação. Apenas métodos sem implementação são **abstract**.

```
interface Corredor {  
    default String correr() { // COMPILA - método default não é abstract  
        return "Correndo";  
    }  
  
    abstract default String correrRapido() { // NÃO COMPILA - método default não  
    pode ser declarado abstract  
        return "Correndo Rápido";  
    }  
  
    String correrDevagar(); // COMPILA - método comum, é abstract por padrão, mesmo  
    que de forma implícita  
  
    abstract String correrExtremo(); // COMPILA - método comum, declarado abstract de  
    forma explícita  
  
    abstract static double calculeVelocidade(int d, int t) { // NÃO COMPILA - método  
    static não pode ser declarado abstract  
        return d / t;  
    }  
}
```

7. Se uma classe implementa duas interfaces que possuem métodos `default` repetidos, ela obrigatoriamente deve implementar o seu próprio.

```
interface Corredor {  
    default String correr() {  
        return "Correndo";  
    }  
}  
  
interface Piloto {  
    default String correr() {  
        return "Piloto Correndo";  
    }  
}  
  
static class Pessoa implements Corredor, Piloto { // NÃO COMPILA - implementa duas  
    interfaces com métodos repetidos e não sobrescreve  
  
}  
  
static class Gigante implements Corredor, Piloto { // COMPILA - implementa duas  
    interfaces com métodos repetidos, mas sobrescreve e cria sua própria implementação  
    @Override  
    public String correr() {  
        return "Gigante Correndo";  
    }  
}
```

8. Ao implementar múltiplas interfaces, é possível acessar a implementação `default` de uma delas.

```
interface Corredor {  
    default String correr() {  
        return "Correndo";  
    }  
}  
  
interface Piloto {  
    default String correr() {  
        return "Piloto Correndo";  
    }  
}  
  
static class Pessoa implements Corredor, Piloto { // COMPILA - mantém a  
implementação do Corredor no método correr()  
    @Override  
    public String correr() {  
        return Corredor.super.correr();  
    }  
  
    public static void main(String[] args) {  
        System.out.println(new Pessoa().correr());  
    }  
}
```

Saída no console

Correndo

9. Quando uma interface herda de outra interface métodos **default**, estes podem ser mantidos, transformados em **abstract** ou redefinidos.

```
interface Corredor {  
    default String correr() {  
        return "Correndo";  
    }  
    default String correrRapido() {  
        return "Correndo Rápido";  
    }  
    default String correrDevagar() {  
        return "Correndo Devagar";  
    }  
}  
  
interface Piloto extends Corredor {  
    String correrRapido();  
  
    default String correrDevagar() {  
        return "Piloto Correndo Devagar";  
    }  
}
```

Nesse exemplo, a interface `Piloto` herda de `Corredor` e mostra 3 cenários distintos:

- Mantém o método `correr()` inalterado;
- Altera o método `correrRapido()` para que seja `abstract`, fazendo com que qualquer classe que implemente a interface `Piloto` tenha que implementar esse método;
- Altera o método `correrDevagar()` para que tenha sua própria implementação

Referências

- Designing an Interface

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 48). Wiley. Edição do Kindle.

- [Static and Default Methods in Interfaces in Java](#).
- [Default Methods](#). Java Documentation.

Localization

Localização (Locale)

Objetivo

Describe the advantages of localizing an application and developing code that defines, reads, and sets the locale with a Locale object.

-

Descrever as vantagens de localizar uma aplicação e desenvolver código que defina, leia e aplique a localidade em um objeto Locale.

É esperado que o candidato saiba compreender e analisar aspectos de Internacionalização e Localização, incluindo o uso da classe **Locale**.

Alguns aspectos de uma aplicação podem depender do país e da linguagem. Por exemplo:

- Formatos de escrita de data
 - O dia 6 de Agosto de 2019 seria representado no Brasil por **06/08/2019**, porém nos EUA seria **08/06/2019**.
- Formatos de escrita de valores monetários
 - Dez Reais no Brasil seriam representados por **R\$ 10**, enquanto na França Dez Euros seriam **10 €**.
- Formatos de separação de casas decimais ou milhares
 - No Brasil utiliza-se vírgula para separar decimais e pontos para milhares: **100.245,03**. Nos EUA, utiliza-se o inverso: **100,245.03**.

Por isso, para que sua aplicação lide corretamente com esses cenários, é necessário compreender dois aspectos: **Internacionalização (Internationalization)** e **Localização (Localization)**.

A **Internacionalização**, também chamada de **i18n**, é o ato de projetar sua aplicação para que seja possível facilmente adaptá-la para utilizar novos formatos e idiomas.

A **Localização**, também chamada de **l10n**, é o ato de adaptar sua aplicação para de fato utilizar um novo formato específico.

Antes de continuar, entenda a execução do método **main** no exemplo a seguir e o que é apresentado no console após sua execução.

```
public static void main(String[] args) throws IOException {
    System.out.println(" - Constantes - ");
    System.out.println(Locale.CANADA);
    System.out.println(Locale.UK);

    System.out.println(" - Construtor - ");
    System.out.println(new Locale("pt", "BR"));
    System.out.println(new Locale("pt", "PT"));
    System.out.println(new Locale("ca", "ES", "VALENCIA"));

    System.out.println(" - Language Tags - ");
    System.out.println(Locale.forLanguageTag("en-CA"));
    System.out.println(Locale.forLanguageTag("pt-BR"));
    System.out.println(Locale.forLanguageTag("pt-PT"));
    System.out.println(Locale.forLanguageTag("ca-ES"));
    System.out.println(Locale.forLanguageTag("gsw-u-sd-chzh"));

    System.out.println(" - Builder - ");
    Locale locale1 = new Locale.Builder()
        .setLanguage("pt")
        .setRegion("BR")
        .build();
    System.out.println(locale1);

    Locale locale2 = new Locale.Builder()
        .setLanguage("az")
        .setRegion("AZ")
        .setScript("Latn")
        .build();
    System.out.println(locale2);

    Locale locale3 = new Locale.Builder()
        .setLanguage("bs")
        .setRegion("BA")
        .setVariant("POSIX")
        .setScript("Latn")
        .setExtension('c', 'cc')
        .build();
    System.out.println(locale3);

}
```

Saída no console

```
- Constantes -
en_CA
en_GB
- Construtor -
pt_BR
pt_PT
ca_ES_VALENCIA
- Language Tags -
en_CA
pt_BR
pt_PT
ca_ES
gsw_#u-sd-chzh
- Builder -
pt_BR
az_AZ_#Latn
bs_BA_POSIX_#Latn_c-cc
```

No Java, a classe `Locale` no pacote `java.util` nos ajuda a lidar com esses aspectos.

1. Geralmente, um `Locale` é representado por um idioma e um país.

- `pt_BR` - Português do Brasil
- `en_US` - Inglês dos EUA
- `it_CH` - Italiano da Suíça
- `fr_BE` - Francês da Bélgica

`src/org/j6toj8/localization/locale/Locale_LocaleLanguageCountry.java`

```
new Locale("pt", "BR"); // Português do Brasil
new Locale("en", "US"); // Inglês dos EUA
new Locale("it", "CH"); // Italiano da Suíça
new Locale("fr", "BE"); // Francês da Bélgica
```

2. Um `Locale` pode ter ainda uma Variante, um Script e Extensões.

src/org/j6toj8/localization/locale/Locale_VarScriptExtension.java

```
public static void main(String[] args) throws IOException {
    Locale locale2 = new Locale.Builder() // az_AZ_#Latn
        .setLanguage("az")
        .setRegion("AZ")
        .setScript("Latn")
        .build();

    Locale locale3 = new Locale.Builder() // bs_BA_POSIX_#Latn_c-cc
        .setLanguage("bs")
        .setRegion("BA")
        .setVariant("POSIX")
        .setScript("Latn")
        .setExtension('c', "cc")
        .build();
}
```



No exame da certificação, a Oracle geralmente só utiliza idioma e país.

3. É possível construir um Locale com o **Builder**, com construtores, ou por uma *Language Tag*.

src/org/j6toj8/localization/locale/LocaleLocaleInstantiation.java

```
public static void main(String[] args) throws IOException {
    new Locale("pt", "BR"); // pt-BR com Construtor

    Locale.forLanguageTag("pt-BR"); // pt-BR com LanguageTag

    Locale localePtBR = new Locale.Builder() // pt-BR com Builder
        .setLanguage("pt")
        .setRegion("BR")
        .build();
}
```

A diferença entre eles é:

- Com os construtores é possível passar apenas o idioma, a região (país) e uma variante.
- Com language tags é possível passar uma **String** no padrão [IETF BCP 47](#).
- Com o builder é possível criar da forma mais específica possível: idioma, região, variante, script e extensões.

4. O **Locale** aceita letras maiúsculas e minúsculas escritas de forma incorreta.

src/org/j6toj8/localization/locale/Locale_LocaleCase.java

```
public static void main(String[] args) throws IOException {
    System.out.println(new Locale("PT", "br"));

    System.out.println(Locale.forLanguageTag("PT-br"));

    Locale localePtBR = new Locale.Builder()
        .setLanguage("PT")
        .setRegion("br")
        .build();

    System.out.println(localePtBR);
}
```

Saída no console

```
pt_BR
pt_BR
pt_BR
```

Nesse exemplo, escrevemos de forma **incorrecta**:

- O idioma deveria ser minúsculo (pt), mas está maiúsculo (PT).
- A região deveria estar maiúscula (BR), está minúscula (br).

Mesmo assim, o **Locale** é criado corretamente. Veja que isso é um código **ruim**. O ideal é sempre escrever respeitando maiúsculas e minúsculas.

5. Existem algumas constantes na classe **Locale** para as localizações mais populares.

src/org/j6toj8/localization/locale/Locale_LocaleCommons.java

```
public static void main(String[] args) throws IOException {
    System.out.println(Locale.CANADA);
    System.out.println(Locale.CANADA_FRENCH);
    System.out.println(Locale.CHINA);
    System.out.println(Locale.CHINESE);
    System.out.println(Locale.ENGLISH);
    System.out.println(Locale.ITALY);
    System.out.println(Locale.SIMPLIFIED_CHINESE);
    System.out.println(Locale.TRADITIONAL_CHINESE);
}
```

Saída no console

```
en_CA  
fr_CA  
zh_CN  
zh  
en  
it_IT  
zh_CN  
zh_TW
```

6. É possível recuperar o **Locale** padrão ou alterá-lo programaticamente.

src/org/j6toj8/localization/locale/Locale_LocaleDefault.java

```
public static void main(String[] args) throws IOException {  
    System.out.println(Locale.getDefault()); // o padrão inicial muda de acordo com  
    // seu dispositivo  
    Locale.setDefault(Locale.KOREA); // altera o Locale default  
    System.out.println(Locale.getDefault()); // ko_KR  
}
```

Saída no console

```
pt_BR  
ko_KR
```

7. É possível verificar os **Locale** disponíveis, pois eles variam de acordo com a JVM sendo executada.

src/org/j6toj8/localization/locale/Locale_LocaleAvailable.java

```
public static void main(String[] args) throws IOException {  
    Locale[] availableLocales = Locale.getAvailableLocales();  
    // imprime os 10 primeiros Locales disponíveis  
    for (int i = 0; i < 10; i++) {  
        System.out.println(availableLocales[i]);  
    }  
}
```

Saída no console

```
nn  
ar_J0  
bg  
kea  
nds  
zu  
am_ET  
fr_DZ  
ti_ET
```

8. Um **Locale** também pode ser representado somente pelo idioma.

src/org/j6toj8/localization/locale/Locale_LocaleLanguageOnly.java

```
public static void main(String[] args) throws IOException {  
    System.out.println(new Locale("pt")); // português  
    System.out.println(new Locale("en")); // inglês  
    System.out.println(new Locale("es")); // espanhol  
    System.out.println(new Locale("fr")); // francês  
}
```

Saída no console

```
pt  
en  
es  
fr
```

Referências

- Adding Internationalization and Localization

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 255). Wiley. Edição do Kindle.

- [Internationalization and Localization in Java 8](#).
- [Internationalization](#). Java Documentation.
- [i18n vs l10n — what's the diff?](#)
- [Internationalization: Understanding Locale in the Java Platform](#).

Pacote de Recursos (Resource Bundle)

Objetivo

- Build a resource bundle for a locale and call a resource bundle from an application.
- Construa um resource bundle (conjunto de recursos) para um Locale e invoque um resource bundle a partir de uma aplicação.

É esperado que o candidato saiba compreender e analisar o uso de *resource bundles* e sua relação com a classe `Locale`.

Ao codificar uma aplicação com internacionalização é comum o uso de *resource bundles*, ou "conjunto de recursos" em português. São arquivos, geralmente `.properties` ou classes Java, que armazenam *Strings*. Cada arquivo contém *Strings* em idiomas, ou *Locales*, diferentes.

Antes de continuar, entenda a execução do método `main` no exemplo a seguir e o que é apresentado no console após sua execução.

`src/org/j6toj8/localization/resourcebundle/ResourceBundle_Complete.java`

```
public static void main(String[] args) {  
  
    Locale.setDefault(new Locale("en", "US"));  
  
    System.out.println("\n -- Locale padrão (en_US) -- ");  
    ResourceBundle bundleDefault = ResourceBundle.getBundle("Text");  
    Set<String> keySetDefault = bundleDefault.keySet();  
    for (String string : keySetDefault) {  
        System.out.println(string + " - " + bundleDefault.getString(string));  
    }  
  
    System.out.println("\n -- Locale es_ES -- ");  
    ResourceBundle bundleEsEs = ResourceBundle.getBundle("Text", new Locale("es",  
    "ES"));  
    Set<String> keySetEsEs = bundleEsEs.keySet();  
    for (String string : keySetEsEs) {  
        System.out.println(string + " - " + bundleEsEs.getString(string));  
    }  
  
    System.out.println("\n -- Locale pt_BR -- ");  
    ResourceBundle bundlePtBr = ResourceBundle.getBundle("Text", new Locale("pt",  
    "BR"));  
    Set<String> keySetPtBr = bundlePtBr.keySet();  
    for (String string : keySetPtBr) {  
        System.out.println(string + " - " + bundlePtBr.getString(string));  
    }  
}
```

../../../../resources/Text.properties

```
phone=phone
tripod tripod
pen:pen
keyboard=keyboard
glass=glass
sticker sticker
paper:papel
rubber rubber
```

../../../../resources/Text_pt.properties

```
paper = papel
```

../../../../resources/Text_pt_BR.properties

```
#arquivo do locale pt_BR
pen=caneta
```

../../../../resources/Text_es.properties

```
keyboard=tec\
lado
```

../../../../resources/Text_es_ES.properties

```
#arquivo do locale es_ES
glass=\tvaso
```

Saída no console

```
-- Locale padrão (en_US) --
tripod - tripod
keyboard - keyboard
glass - glass
paper - paper
phone - phone
rubber - rubber
sticker - sticker
pen - pen

-- Locale es_ES --
tripod - tripod
keyboard - teclado
glass - vaso
paper - paper
phone - phone
rubber - rubber
sticker - sticker
pen - pen

-- Locale pt_BR --
tripod - tripod
keyboard - keyboard
glass - glass
paper - papel
phone - phone
rubber - rubber
pen - caneta
sticker - sticker
```

1. O nome do **Locale** é o sufixo do nome do arquivo, e o *resource bundle* padrão não tem sufixo.
Exemplos:

- **Text.properties** → Locale padrão
- **Text_pt_BR.properties** → Locale pt_BR
- **Text_pt.properties** → Locale pt
- **Text_es_ES.properties** → Locale es_ES
- **Text_es.properties** → Locale es

2. O arquivo **.properties** pode ser expresso com 3 separadores diferentes: **=** (igual), **:** (dois pontos) ou um espaço em branco.

../../../../resources/Text.properties

```
phone=phone
tripod tripod
pen:pen
keyboard=keyboard
glass=glass
sticker sticker
paper:paper
rubber rubber
```

O mais comum é utilizar o `=` para separar as propriedades, mas as 3 funcionam da mesma forma.

3. Em arquivos `.properties`, linhas que começam com `#` ou `!` são comentários.

../../../../resources/Text_pt_BR.properties

```
#arquivo do locale pt_BR
pen=caneta
```

../../../../resources/Text_es_ES.properties

```
!arquivo do locale es_ES
glass=\tvaso
```

4. Em arquivos `.properties`, apenas espaços no final da linha são considerados.

../../../../resources/Text_pt.properties

```
paper = papel
```

Neste exemplo, não é possível ver, mas existem 3 espaços no final da linha. O resultado é o mesmo que escrever `paper=papel`.

5. Em arquivos `.properties`, se você terminar a linha com uma contrabarra, pode quebrar a linha.

../../../../resources/Text_es.properties

```
keyboard=tec\
lado
```

Neste exemplo, seria o mesmo que escrever em uma única linha: `keyboard=teclado`.

6. Em arquivos `.properties`, você também pode usar os caracteres java como `\t` e `\n`.

.../..../resources/Text_es_ES.properties

```
!arquivo do locale es_ES
glass=\tvaso
```

Neste exemplo, existe uma tabulação antes da palavra **vaso**. Você pode perceber no primeiro exemplo deste capítulo que a palavra **vaso** foi impressa no console com uma tabulação à esquerda.

7. Você pode recuperar todas as chaves e valores do *resource bundle* programmaticamente.

src/org/j6toj8/localization/resourcebundle/ResourceBundle_KeysProgrammatically.java

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

    Locale.setDefault(new Locale("en", "US")); // Coloca o Locale en_US como padrão

    ResourceBundle bundle = ResourceBundle.getBundle("Text", new Locale("pt", "BR"));
    // Recupera o bundle 'Text' para o Locale pt_BR
    Set<String> keySet = bundle.keySet(); // Pega um Set com todas as chaves
    for (String key : keySet) {
        System.out.println(key + " - " + bundle.getString(key)); // Imprime "<chave> - <valor>"
    }

}
```

Saída no console

```
tripod - tripod
keyboard - keyboard
glass - glass
paper - papel
phone - phone
rubber - rubber
pen - caneta
sticker - sticker
```

8. O *resource bundle* também pode ser uma classe Java.

resource/Text_fr_CA.java

```
import java.util.ListResourceBundle;

public class Text_fr_CA extends ListResourceBundle {

    @Override
    protected Object[][] getContents() {
        return new Object[][] {
            {"pen", "stylo"},
            {"glass", "verre"},
            {"keyboard", "clavier"}
        };
    }
}
```

src/org/j6toj8/localization/resourcebundle/ResourceBundle_JavaBundle.java

```
 ResourceBundle bundle = ResourceBundle.getBundle("Text", new Locale("fr", "CA"));
System.out.println(bundle.getString("glass"));
```

Saída no console

```
verre
```

9. Ao utilizar classes Java, uma vantagem é poder armazenar valores que não sejam apenas **String**.

resource/Text_fr_FR.java

```
import java.math.BigDecimal;
import java.util.ListResourceBundle;

public class Text_fr_FR extends ListResourceBundle {

    @Override
    protected Object[][] getContents() {
        return new Object[][] {
            {"ten", new BigDecimal("10")}
        };
    }
}
```

10. A nomenclatura do arquivo é a mesma para classes Java e arquivos **.properties**, mudando apenas a extensão.

Veja que os arquivos **.properties** se chamam **Text_xx_XX.properties**, e os arquivos **.java** se chamam **Text_xx_XX.java**. Programaticamente, ambos são utilizados da mesma forma.

11. Se houver um arquivo `.properties` e uma classe Java para o mesmo `Locale`, a classe Java é utilizada.

resource/Text_fr_CA.java

```
import java.util.ListResourceBundle;

public class Text_fr_CA extends ListResourceBundle {

    @Override
    protected Object[][] getContents() {
        return new Object[][] {
            {"pen", "stylo"},
            {"glass", "verre"},
            {"keyboard", "clavier"}
        };
    }
}
```

resource/Text_fr_CA.properties

```
pen=stylo-in-ignored-properties
glass=stylo-in-ignored-properties
keyboard=stylo-in-ignored-properties
```

src/org/j6toj8/localization/resourcebundle/ResourceBundle_JavaClassTakesPrecedence.java

```
/*
 * Recupera o Bundle do arquivo "Text_fr_CA.java",
 * pois tem precedência sobre o arquivo "Text_fr_CA.properties"
 */
 ResourceBundle bundle = ResourceBundle.getBundle("Text", new Locale("fr", "CA"));
 System.out.println(bundle.getString("pen"));
 System.out.println(bundle.getString("glass"));
 System.out.println(bundle.getString("keyboard"));
```

Saída no console

```
stylo
verre
clavier
```

Veja que os valores apresentados no console são aqueles definidos no arquivo `Text_fr_CA.java`, mostrando que a classe Java tem precedência sobre um arquivo `.properties` para o mesmo `Locale`.

12. Ao buscar por um *resource bundle*, o Java tenta encontrar um arquivo com o `Locale` exato. Se não encontrar, busca na seguinte ordem:

- a. Um arquivo do mesmo idioma, mas sem o país;
- b. Um arquivo do **Locale** padrão;
- c. Um arquivo do **Locale** padrão, mas sem o país;
- d. Um arquivo sem **Locale**, que é o *resource bundle* padrão;
- e. Lança **MissingResourceException** caso não encontre.

Por exemplo, ao executar a aplicação com o Locale padrão **en_US**, e solicitar um Locale **pt_BR**, a ordem de busca do *resource bundle* seria a seguinte:

- i. **Text_pt_BR** → Locale exato
- ii. **Text_pt** → Locale solicitado, sem o país
- iii. **Text_en_US** → Locale padrão
- iv. **Text_en** → Locale padrão, sem o país
- v. **Text** → *Resource Bundle* padrão

src/org/j6toj8/localization/resourcebundle/ResourceBundle_NotExactLocale.java

```
Locale.setDefault(new Locale("pt", "BR")); // pt_BR como Locale padrão
ResourceBundle bundle2 = ResourceBundle.getBundle("Text", new Locale("zh",
"CN"));
System.out.println(bundle2.getLocale()); // Bundle localizado para o Locale
"zh_CH" (Chinês simplificado)
ResourceBundle bundle = ResourceBundle.getBundle("Text", new Locale("it",
"CH"));
System.out.println(bundle.getLocale()); // Bundle localizado para o Locale
"it_CH" (Italiano da Suíça)
```

Saída no console

```
pt_BR
it
```

Veja que o **Locale** padrão é **pt_BR**. Por isso ele foi utilizado ao solicitar um *resource bundle* para **zh_CN**, pois não existe um bundle para esse **Locale**.

Por outro lado, ao solicitar um *resource bundle* para o **Locale it_CH**, ele encontrou o mais próximo, que seria o **Locale it**, mas sem um país específico.

13. Os arquivos mais específicos herdam as chaves e valores de arquivos mais genéricos, caso eles não as tenham.

```
Locale.setDefault(new Locale("en", "US")); // pt_BR como Locale padrão
ResourceBundle bundle = ResourceBundle.getBundle("Text", new Locale("pt", "BR"));
System.out.println("Locale: " + bundle.getLocale()); // Bundle localizado para o
Locale "pt_BR" (Português do Brasil)
System.out.println(bundle.getObject("pen"));
System.out.println(bundle.getObject("paper"));
System.out.println(bundle.getObject("keyboard"));
```

../../../../resources/Text.properties

```
phone=phone
tripod tripod
pen:pen
keyboard=keyboard
glass=glass
sticker sticker
paper:paper
rubber rubber
```

../../../../resources/Text_pt.properties

```
paper = papel
```

../../../../resources/Text_pt_BR.properties

```
#arquivo do locale pt_BR
pen=caneta
```

Saída no console

```
Locale: pt_BR
caneta
papel
keyboard
```

Veja que nesse exemplo foi localizado um *resource bundle* com o **Locale** exato **pt_BR**. Porém, nem todas as chaves foram encontradas nesse arquivo:

- **caneta** foi localizado no arquivo **Text_pt_BR.properties**
- **papel** foi localizado no arquivo **Text_pt.properties**
- **keyboard** foi localizado no arquivo **Text.properties**

Referências

- Using a Resource Bundle

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 258). Wiley. Edição do Kindle.

- [A Guide to the ResourceBundle](#).
- [Class ResourceBundle](#). Java Documentation.
- [About the ResourceBundle Class](#). Java Documentation.

Data e Hora

Objetivo

Create and manage date- and time-based events by using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration, including a combination of date and time in a single object.

-
Crie e gerencie eventos baseados em data e hora utilizando LocalDate, LocalTime, LocalDateTime, Instant, Period, e Duration, incluindo combinação de data e hora em um único objeto.

O Java 8 possui uma forma totalmente nova de lidar com data e hora. Várias classes novas foram introduzidas no pacote `java.time.*`. Vejamos a seguir alguns exemplos no formato ISO-8601.

- `LocalDate` → Data sem fuso horário, como `2007-12-03`.
- `LocalTime` → Hora sem fuso horário, como `10:15:30.125`.
- `LocalDateTime` → Uma data com hora sem fuso horário, como `2007-12-03T10:15:30.125`.
- `ZonedDateTime` → Uma data com hora e com fuso horário, como `2007-12-03T10:15:30.125+01:00 Europe/Paris`.
- `Instant` → Um ponto na linha do tempo, um instante.
- `Period` → Uma quantidade de tempo baseado em data, como "2 anos, 3 meses and 4 dias".
- `Duration` → Uma quantidade de tempo baseado em hora, como "34,5 segundos".

Essas novas classes foram melhor projetadas para lidar com os conceitos de data, hora e tempo. As classes antigas, `java.util.Date` e `java.util.Calendar`, não caem na prova de certificação.

Quase todas essas classes serão apresentadas nessa seção. A única exceção é `ZonedDateTime`, que será apresentada na seção de fusos horários e horário de verão.

Todas as novas classes são *imutáveis* e *thread safe*. Ou seja, não é necessário se preocupar com concorrência.

LocalDate

Um `LocalDate` representa uma data sem fuso horário, como `2007-12-03`.

1. A prova geralmente irá utilizar datas no formato americano: `mes/dia/ano`.
2. É possível criar um `LocalDate` através do método `static` chamado `now`.

`src/org/j6toj8/localization/datetime/localdate/LocalDate_Now.java`

```
System.out.println(LocalDate.now());
```

Saída no console

```
2019-05-20
```

A saída no console irá apresentar a data atual.

3. Também é possível criar um `LocalDate` através do método `static` chamado `of`.
4. Você pode utilizar o `enum Month` ou um `int` para representar o mês.
5. Diferente das APIs antigas do Java, o mês agora começa no número `1`, que é Janeiro.

`src/org/j6toj8/localization/datetime/localdate/LocalDate_Of.java`

```
System.out.println(LocalDate.of(2019, 5, 20));
System.out.println(LocalDate.of(2019, Month.MAY, 20));
```

Saída no console

```
2019-05-20
2019-05-20
```

6. Assim como todas as novas classes de data e hora, não é possível criar uma instância de `LocalDate` utilizando o construtor.

`src/org/j6toj8/localization/datetime/localdate/LocalDate_Constructor.java`

```
LocalDate localDate = new LocalDate(); // NÃO COMPILA! - não possui construtor
                                         padrão
System.out.println(localDate);
```

7. Será lançada a exceção `DateTimeException` ao tentar criar uma data inválida.

`src/org/j6toj8/localization/datetime/localdate/LocalDate_Invalid.java`

```
System.out.println(LocalDate.of(2019, Month.MAY, 33));
```

Saída no console

```
Exception in thread "main" java.time.DateTimeException: Invalid value for
DayOfMonth (valid values 1 - 28/31): 33
    at java.time.temporal.ValueRange.checkValidValue(ValueRange.java:311)
    at java.time.temporal.ChronoField.checkValidValue(ChronoField.java:703)
    at java.time.LocalDate.of(LocalDate.java:248)
    at
org.j6toj8.localization.datetime.localdate.LocalDate_Invalid.main(LocalDate_Invalid
.java:10)
```

8. Existem vários métodos para somar e subtrair de **LocalDate**.

src/org/j6toj8/localization/datetime/localdate/LocalDate_Manipulate.java

```
LocalDate localDate = LocalDate.of(2019, Month.MAY, 20);
System.out.println(localDate);
System.out.println("+2 dias: " + localDate.plusDays(2));
System.out.println("+2 semanas: " + localDate.plusWeeks(2));
System.out.println("+2 meses: " + localDate.plusMonths(2));
System.out.println("+2 anos: " + localDate.plusYears(2));
System.out.println("+2 anos: " + localDate.plusYears(2));
System.out.println("+2 décadas: " + localDate.plus(2, ChronoUnit.DECADES));
System.out.println("-2 dias: " + localDate.minusDays(2));
System.out.println("-2 semanas: " + localDate.minusWeeks(2));
System.out.println("-2 meses: " + localDate.minusMonths(2));
System.out.println("-2 anos: " + localDate.minusYears(2));
System.out.println("-2 décadas: " + localDate.minus(2, ChronoUnit.DECADES));
```

Saída no console

```
2019-05-20
+2 dias: 2019-05-22
+2 semanas: 2019-06-03
+2 meses: 2019-07-20
+2 anos: 2021-05-20
+2 anos: 2021-05-20
+2 décadas: 2039-05-20
-2 dias: 2019-05-18
-2 semanas: 2019-05-06
-2 meses: 2019-03-20
-2 anos: 2017-05-20
-2 décadas: 1999-05-20
```

9. **LocalDate** é imutável, então é necessário armazenar o retorno de uma alteração em uma variável.

src/org/j6toj8/localization/datetime/localdate/LocalDate_Immutability.java

```
LocalDate localDate = LocalDate.of(2019, Month.MAY, 20);
System.out.println(localDate);
localDate.plusDays(1); // chamada perdida - a nova data não foi armazenada em uma
variável
System.out.println(localDate);
localDate = localDate.plusDays(1); // chamada útil - data armazenada na variável
System.out.println(localDate);
```

Saída no console

```
2019-05-20
2019-05-20
2019-05-21
```

10. É comum utilizar o encadeamento de chamadas com esses métodos.

src/org/j6toj8/localization/datetime/localdate/LocalDate_Chaining.java

```
LocalDate localDate = LocalDate.of(2019, Month.MAY, 20);
System.out.println(localDate);
System.out.println(localDate.plusDays(1).plusMonths(1).plusYears(1));
```

Saída no console

```
2019-05-20
2020-06-21
```

11. Ao manipular a data, o **LocalDate** irá manipular os meses e anos conforme necessário.

src/org/j6toj8/localization/datetime/localdate/LocalDate_AdjustDifferentUnit.java

```
LocalDate localDate = LocalDate.of(2019, Month.NOVEMBER, 30);
System.out.println(localDate);
System.out.println(localDate.plusDays(1)); // + 1 dia, vira o mês
System.out.println(localDate.plusDays(32)); // + 32 dias, vira o ano
System.out.println(localDate.plusMonths(2)); // + 2 meses, vira o ano
```

Saída no console

```
2019-11-30
2019-12-01
2020-01-01
2020-01-30
```

LocalTime

Um `LocalTime` representa uma hora sem fuso horário e sem data, como `10:15:30.125`.

1. A hora é representada no formato `hora:minuto:segundo.nano`.
2. É possível criar um `LocalTime` através do método `static` chamado `now`.

src/org/j6toj8/localization/datetime/localtime/LocalTime_Now.java

```
System.out.println(LocalTime.now());
```

Saída no console

```
09:15:23.197
```

A saída no console irá apresentar a hora atual.

3. Também é possível criar um `LocalTime` através do método `static` chamado `of`.

src/org/j6toj8/localization/datetime/localtime/LocalTime_Of.java

```
System.out.println(LocalTime.of(9, 20, 1, 135000000));
System.out.println(LocalTime.of(9, 20, 1, 135));
System.out.println(LocalTime.of(9, 20, 1));
System.out.println(LocalTime.of(9, 20));
```

Saída no console

```
09:20:01.135
09:20:01.000000135
09:20:01
09:20
```

4. Assim como todas as novas classes de data e hora, não é possível criar uma instância de `LocalTime` utilizando o construtor.

src/org/j6toj8/localization/datetime/localtime/LocalTime_Constructor.java

```
LocalTime localTime = new LocalTime(); // NÃO COMPILA! - não possui construtor
padrão
System.out.println(localTime);
```

5. Será lançada a exceção `DateTimeException` ao tentar criar uma hora inválida.

src/org/j6toj8/localization/datetime/localtime/LocalTime_Invalid.java

```
System.out.println(LocalTime.of(24, 2, 3)); // lança exceção: a hora deve estar entre 0 e 23
```

Saída no console

```
Exception in thread "main" java.time.DateTimeException: Invalid value for HourOfDay (valid values 0 - 23): 24
    at java.time.temporal.ValueRange.checkValidValue(ValueRange.java:311)
    at java.time.temporal.ChronoField.checkValidValue(ChronoField.java:703)
    at java.time.LocalTime.of(LocalTime.java:317)
    at
org.j6toj8.localization.datetime.localtime.LocalTime_Invalid.main(LocalTime_Invalid.java:9)
```

6. Existem vários métodos para somar e subtrair de `LocalTime`.

src/org/j6toj8/localization/datetime/localtime/LocalTime_Manipulate.java

```
LocalTime localTime = LocalTime.of(9, 26, 12);
System.out.println(localTime);
System.out.println("+2 horas: " + localTime.plusHours(2));
System.out.println("+2 minutos: " + localTime.plusMinutes(2));
System.out.println("+2 segundos: " + localTime.plusSeconds(2));
System.out.println("+2 nanosegundos: " + localTime.plusNanos(2));
System.out.println("+2 microsegundos: " + localTime.plus(2, ChronoUnit.MICROS));
System.out.println("+2 milissegundos: " + localTime.plus(2, ChronoUnit.MILLIS));
System.out.println("-2 horas: " + localTime.minusHours(2));
System.out.println("-2 minutos: " + localTime.minusMinutes(2));
System.out.println("-2 segundos: " + localTime.minusSeconds(2));
System.out.println("-2 nanosegundos: " + localTime.minusNanos(2));
System.out.println("-2 microsegundos: " + localTime.minus(2, ChronoUnit.MICROS));
System.out.println("-2 milissegundos: " + localTime.minus(2, ChronoUnit.MILLIS));
```

Saída no console

```
09:26:12
+2 horas: 11:26:12
+2 minutos: 09:28:12
+2 segundos: 09:26:14
+2 nanosegundos: 09:26:12.000000002
+2 microsegundos: 09:26:12.000002
+2 milissegundos: 09:26:12.002
-2 horas: 07:26:12
-2 minutos: 09:24:12
-2 segundos: 09:26:10
-2 nanosegundos: 09:26:11.999999998
-2 microsegundos: 09:26:11.999998
-2 milissegundos: 09:26:11.998
```

7. `LocalTime` é imutável, então é necessário armazenar o retorno de uma alteração em uma variável.

src/org/j6toj8/localization/datetime/localtime/LocalTime_Immutability.java

```
LocalTime localTime = LocalTime.of(9, 31, 5);
System.out.println(localTime);
localTime.plusHours(1); // chamada perdida - a nova hora não foi armazenada em uma
variável
System.out.println(localTime);
localTime = localTime.plusHours(1); // chamada útil - hora armazenada na variável
System.out.println(localTime);
```

Saída no console

```
09:31:05
09:31:05
10:31:05
```

8. É comum utilizar o encadeamento de chamadas com esses métodos.

src/org/j6toj8/localization/datetime/localtime/LocalTime_Chaining.java

```
LocalTime localTime = LocalTime.of(9, 32, 5);
System.out.println(localTime);
System.out.println(localTime.plusHours(1).plusMinutes(1).plusSeconds(1).plusNanos(1
000000));
```

Saída no console

```
09:32:05
10:33:06.001
```

9. Ao manipular a hora, o `LocalTime` irá manipular as horas, minutos e segundos conforme necessário.

`src/org/j6toj8/localization/datetime/localtime/LocalTime_AdjustDifferentUnit.java`

```
LocalTime localTime = LocalTime.of(9, 59, 59);
System.out.println(localTime);
System.out.println(localTime.plusSeconds(2)); // + 2 segundos, vira o minuto
System.out.println(localTime.plusSeconds(62)); // + 62 segundos, vira a hora
System.out.println(localTime.plusMinutes(2)); // + 2 minutos, vira a hora
System.out.println(localTime.minusNanos(100000000)); // - 1 segundo (em nanos),
vira o minuto
```

Saída no console

```
09:59:59
10:00:01
10:01:01
10:01:59
09:59:58
```

LocalDateTime

Um `LocalDateTime` representa uma data com hora, mas sem fuso horário, como `2007-12-03T10:15:30.125`.

As regras para o `LocalDateTime` são basicamente a junção daquelas para `LocalDate` e `LocalTime`.

1. O `LocalDateTime` é apresentado no formato `ano-mes-diaThora:minuto:segundo.nano`.
2. É possível criar um `LocalDateTime` através do método `static` chamado `now`.

`src/org/j6toj8/localization/datetime/localdatetime/LocalDateTime_Now.java`

```
System.out.println(LocalDateTime.now());
```

Saída no console

```
2019-05-24T10:13:58.370
```

A saída no console irá apresentar a data e hora atual.

3. Também é possível criar um `LocalDateTime` através do método `static` chamado `of`.

src/org/j6toj8/localization/datetime/localdatetime/LocalDateTime.Of.java

```
System.out.println(LocalDateTime.of(LocalDate.of(2019, 5, 20), LocalTime.of(9, 20, 12)));
System.out.println(LocalDateTime.of(2019, 5, 20, 9, 20));
System.out.println(LocalDateTime.of(2019, Month.MAY, 20, 9, 20));
System.out.println(LocalDateTime.of(2019, 5, 20, 9, 20, 12));
System.out.println(LocalDateTime.of(2019, Month.MAY, 20, 9, 20, 12));
System.out.println(LocalDateTime.of(2019, 5, 20, 9, 20, 12, 135));
System.out.println(LocalDateTime.of(2019, Month.MAY, 20, 9, 20, 12, 135));
```

Saída no console

```
2019-05-20T09:20:12
2019-05-20T09:20
2019-05-20T09:20
2019-05-20T09:20:12
2019-05-20T09:20:12
2019-05-20T09:20:12.000000135
2019-05-20T09:20:12.000000135
```

4. Assim como todas as novas classes de data e hora, não é possível criar uma instância de **LocalDateTime** utilizando o construtor.

src/org/j6toj8/localization/datetime/localdatetime/LocalDateTime_Constructor.java

```
LocalDateTime localDateTime = new LocalDateTime(); // NÃO COMPILA! - não possui
construtor padrão
System.out.println(localDateTime);
```

5. Será lançada a exceção **DateTimeException** ao tentar criar uma data ou hora inválida.

src/org/j6toj8/localization/datetime/localdatetime/LocalDateTime_Invalid.java

```
System.out.println(LocalDateTime.of(2019, 4, 31, 9, 20)); // lança exceção: não
existe 31 de abril
```

Saída no console

```
Exception in thread "main" java.time.DateTimeException: Invalid date 'APRIL 31'
    at java.time.LocalDate.create(LocalDate.java:431)
    at java.time.LocalDate.of(LocalDate.java:269)
    at java.time.LocalDateTime.of(LocalDateTime.java:311)
    at
org.j6toj8.localization.datetime.localdatetime.LocalDateTime_Invalid.main(LocalDate
Time_Invalid.java:10)
```

6. Existem vários métodos para somar e subtrair de **LocalDateTime**. São basicamente todos os

disponíveis para `LocalDate` e `LocalTime`.

`src/org/j6toj8/localization/datetime/localdatetime/LocalDateTime_Manipulate.java`

```
LocalDateTime localDateTime = LocalDateTime.of(2019, 5, 20, 9, 20, 12);
System.out.println(localDateTime);
System.out.println("+2 horas: " + localDateTime.plusHours(2));
System.out.println("+2 minutos: " + localDateTime.plusMinutes(2));
System.out.println("+2 segundos: " + localDateTime.plusSeconds(2));
System.out.println("+2 nanosegundos: " + localDateTime.plusNanos(2));
System.out.println("+2 microsegundos: " + localDateTime.plus(2, ChronoUnit.
MICROS));
System.out.println("+2 milissegundos: " + localDateTime.plus(2, ChronoUnit.
MILLIS));
System.out.println("-2 horas: " + localDateTime.minusHours(2));
System.out.println("-2 minutos: " + localDateTime.minusMinutes(2));
System.out.println("-2 segundos: " + localDateTime.minusSeconds(2));
System.out.println("-2 nanosegundos: " + localDateTime.minusNanos(2));
System.out.println("-2 microsegundos: " + localDateTime.minus(2, ChronoUnit.
MICROS));
System.out.println("-2 milissegundos: " + localDateTime.minus(2, ChronoUnit.
MILLIS));
System.out.println("+2 dias: " + localDateTime.plusDays(2));
System.out.println("+2 semanas: " + localDateTime.plusWeeks(2));
System.out.println("+2 meses: " + localDateTime.plusMonths(2));
System.out.println("+2 anos: " + localDateTime.plusYears(2));
System.out.println("+2 anos: " + localDateTime.plusYears(2));
System.out.println("+2 décadas: " + localDateTime.plus(2, ChronoUnit.DECADES));
System.out.println("-2 dias: " + localDateTime.minusDays(2));
System.out.println("-2 semanas: " + localDateTime.minusWeeks(2));
System.out.println("-2 meses: " + localDateTime.minusMonths(2));
System.out.println("-2 anos: " + localDateTime.minusYears(2));
System.out.println("-2 décadas: " + localDateTime.minus(2, ChronoUnit.DECADES));
```

Saída no console

```
2019-05-20T09:20:12
+2 horas: 2019-05-20T11:20:12
+2 minutos: 2019-05-20T09:22:12
+2 segundos: 2019-05-20T09:20:14
+2 nanosegundos: 2019-05-20T09:20:12.000000002
+2 microsegundos: 2019-05-20T09:20:12.000002
+2 milissegundos: 2019-05-20T09:20:12.002
-2 horas: 2019-05-20T07:20:12
-2 minutos: 2019-05-20T09:18:12
-2 segundos: 2019-05-20T09:20:10
-2 nanosegundos: 2019-05-20T09:20:11.999999998
-2 microsegundos: 2019-05-20T09:20:11.999998
-2 milissegundos: 2019-05-20T09:20:11.998
+2 dias: 2019-05-22T09:20:12
+2 semanas: 2019-06-03T09:20:12
+2 meses: 2019-07-20T09:20:12
+2 anos: 2021-05-20T09:20:12
+2 anos: 2021-05-20T09:20:12
+2 décadas: 2039-05-20T09:20:12
-2 dias: 2019-05-18T09:20:12
-2 semanas: 2019-05-06T09:20:12
-2 meses: 2019-03-20T09:20:12
-2 anos: 2017-05-20T09:20:12
-2 décadas: 1999-05-20T09:20:12
```

7. `LocalDateTime` é imutável, então é necessário armazenar o retorno de uma alteração em uma variável.

`src/org/j6toj8/localization/datetime/localdatetime/LocalDateTime_Immutability.java`

```
LocalDateTime localDateTime = LocalDateTime.of(2019, 5, 20, 9, 20);
System.out.println(localDateTime);
localDateTime.plusHours(1); // chamada perdida - a nova data/hora não foi
armazenada em uma variável
System.out.println(localDateTime);
localDateTime = localDateTime.plusHours(1); // chamada útil - data/hora armazenada
na variável
System.out.println(localDateTime);
```

Saída no console

```
2019-05-20T09:20
2019-05-20T09:20
2019-05-20T10:20
```

8. É comum utilizar o encadeamento de chamadas com esses métodos.

src/org/j6toj8/localization/datetime/localdatetime/LocalDateTime_Chaining.java

```
LocalDateTime localDateTime = LocalDateTime.of(2019, 5, 20, 9, 20);
System.out.println(localDateTime);
System.out.println(localDateTime.plusDays(1).plusHours(1).plusYears(1));
```

Saída no console

```
2019-05-20T09:20
2020-05-21T10:20
```

9. Ao manipular a data ou hora, o `LocalDateTime` irá manipular os outros campos conforme necessário.

src/org/j6toj8/localization/datetime/localdatetime/LocalDateTime_AdjustDifferentUnit.java

```
LocalDateTime localDateTime = LocalDateTime.of(2019, 12, 31, 23, 59, 59);
System.out.println(localDateTime);
System.out.println(localDateTime.plusSeconds(2)); // + 2 segundos, vira o ano
```

Saída no console

```
2019-12-31T23:59:59
2020-01-01T00:00:01
```

Instant

Um `Instant` representa um momento na linha do tempo, no fuso horário do `UTC` (às vezes chamado de `GMT`), como `2007-12-03T10:15:30.125Z`.

O `Instant` difere de todos os tipos apresentados anteriormente, pois possui um fuso horário (`UTC`) e representa o mesmo momento na linha do tempo em qualquer lugar do mundo.

1. O `Instant` é apresentado no formato `ano-mes-diaThora:minuto:segundo.nanoZ`.
2. É possível criar um `Instant` através do método `static` chamado `now`.

src/org/j6toj8/localization/datetime/instant/Instant_Now.java

```
System.out.println(Instant.now());
```

Saída no console

```
2019-05-27T21:13:10.450Z
```

A saída no console irá apresentar a data e hora atual no fuso horário `UTC`.

3. Também é possível criar um `Instant` através dos métodos `static` chamados `ofEpochMilli` ou

`ofEpochSecond`. O parâmetro é a quantidade de milissegundos, ou segundos, desde 1970-01-01T00:00:00Z.

src/org/j6toj8/localization/datetime/instant/Instant_Of.java

```
System.out.println(Instant.ofEpochMilli(1000000000000L));
System.out.println(Instant.ofEpochSecond(1000000000));
System.out.println(Instant.ofEpochSecond(1000000000, 123000000)); // com
nanossegundos
```

Saída no console

```
2001-09-09T01:46:40Z
2001-09-09T01:46:40Z
2001-09-09T01:46:40.123Z
```

4. Assim como todas as novas classes de data e hora, não é possível criar uma instância de `Instant` utilizando o construtor.

src/org/j6toj8/localization/datetime/instant/Instant_Constructor.java

```
Instant instant = new Instant(); // NÃO COMPILA! - não possui construtor padrão
System.out.println(instant);
```

5. Existem vários métodos para somar e subtrair de `Instant`. Porém, não suporta operações com unidades maiores do que dias.

src/org/j6toj8/localization/datetime/instant/Instant_Manipulate.java

```
Instant instant = Instant.ofEpochMilli(1000000000000L);
System.out.println(instant);
System.out.println("+2 segundos: " + instant.plusSeconds(2));
System.out.println("+2 nanosegundos: " + instant.plusNanos(2));
System.out.println("+2 microsegundos: " + instant.plus(2, ChronoUnit.MICROS));
System.out.println("+2 milissegundos: " + instant.plus(2, ChronoUnit.MILLIS));
System.out.println("-2 segundos: " + instant.minusSeconds(2));
System.out.println("-2 nanosegundos: " + instant.minusNanos(2));
System.out.println("-2 microsegundos: " + instant.minus(2, ChronoUnit.MICROS));
System.out.println("-2 milissegundos: " + instant.minus(2, ChronoUnit.MILLIS));
System.out.println("+2 dias: " + instant.plus(2, ChronoUnit.DAYS));
System.out.println("+2 semanas: " + instant.plus(2, ChronoUnit.WEEKS)); // erro de
execução
```

Saída no console

```
2001-09-09T01:46:40Z
+2 segundos: 2001-09-09T01:46:42Z
+2 nanosegundos: 2001-09-09T01:46:40.000000002Z
+2 microssegundos: 2001-09-09T01:46:40.000002Z
+2 milissegundos: 2001-09-09T01:46:40.002Z
-2 segundos: 2001-09-09T01:46:38Z
-2 nanosegundos: 2001-09-09T01:46:39.999999998Z
-2 microssegundos: 2001-09-09T01:46:39.999998Z
-2 milissegundos: 2001-09-09T01:46:39.998Z
+2 dias: 2001-09-11T01:46:40Z
Exception in thread "main" java.time.temporal.UnsupportedTemporalTypeException:
Unsupported unit: Weeks
    at java.time.Instant.plus(Instant.java:862)
    at
org.j6toj8.localization.datetime.instant.Instant_Manipulate.main(Instant_Manipulate
.java:21)
```

6. **Instant** é imutável, então é necessário armazenar o retorno de uma alteração em uma variável.
7. Assim como as outras classes de data e hora, é comum utilizar o encadeamento de chamadas com esses métodos.
8. Assim como as outras classes de data e hora, ao manipular o **Instant** irá manipular os outros campos conforme necessário.

src/org/j6toj8/localization/datetime/instant/Instant_Immutability.java

```
Instant instant = Instant.ofEpochMilli(1000000000000L);
System.out.println(instant);
instant.plusSeconds(60); // chamada perdida - a nova data/hora não foi armazenada
em uma variável
System.out.println(instant);
instant = instant.plusSeconds(60); // chamada útil - data/hora armazenada na variável
System.out.println(instant);
```

Saída no console

```
2001-09-09T01:46:40Z
2001-09-09T01:46:40Z
2001-09-09T01:47:40Z
```

9. Você pode transformar um **LocalDateTime** em um **Instant**, e vice-versa, caso informe um fuso horário, .

src/org/j6toj8/localization/datetime/instant/Instant_Convert.java

```
LocalDateTime localDateTime = LocalDateTime.of(2019, 5, 27, 13, 1, 1);
System.out.println("LocalDateTime: " + localDateTime);
System.out.println(localDateTime.toInstant(ZoneOffset.UTC));
System.out.println(localDateTime.toInstant(ZoneOffset.of("-3")));

Instant instant = Instant.ofEpochSecond(1558962061L); // mesma data do
localDateTime
System.out.println("\nInstant: " + instant);
System.out.println(LocalDateTime.ofInstant(instant, ZoneOffset.UTC));
System.out.println(LocalDateTime.ofInstant(instant, ZoneOffset.of("-3")));
```

Saída no console

```
LocalDateTime: 2019-05-27T13:01:01
2019-05-27T13:01:01Z
2019-05-27T16:01:01Z
```

```
Instant: 2019-05-27T13:01:01Z
2019-05-27T13:01:01
2019-05-27T10:01:01
```

Period

Um **Period** é uma quantidade de tempo baseado em datas, como anos, meses ou dias. Por exemplo: **2 anos, 3 meses e 4 dias.**

1. O **Period** é apresentado no formato **PxYxMxD**, onde:
 - **P** representa que é um Period.
 - **xY** é a quantidade de anos. 5 anos seria **5Y**.
 - **xM** é a quantidade de meses. 3 meses seria **3M**.
 - **xD** é a quantidade de dias. 2 dias seria **2D**.
2. Ao imprimir um **Period**, apenas os campos com valor são apresentados. Campos zerados são omitidos.
3. É possível criar um **Period** através dos métodos **static** chamados **of***.

src/org/j6toj8/localization/datetime/period/Period_Of.java

```
System.out.println(Period.ofDays(2)); // período de 2 dias
System.out.println(Period.ofMonths(2)); // período de 2 meses
System.out.println(Period.ofWeeks(2)); // período de 2 semanas
System.out.println(Period.ofYears(2)); // período de 2 anos
System.out.println(Period.of(2, 1, 3)); // 2 anos, 1 mês e 3 dias
```

Saída no console

```
P2D  
P2M  
P14D  
P2Y  
P2Y1M3D
```

Perceba que o período de 2 semanas é apresentado como **14D**, pois **Period** não armazena semanas. O método **ofWeeks** é apenas um facilitador, onde cada semana é o equivalente a 7 dias.

4. Também é possível criar um **Period** a partir de duas **LocalDate**.

src/org/j6toj8/localization/datetime/period/Period_Between.java

```
LocalDate nascimento = LocalDate.of(1990, 8, 6);  
LocalDate hoje = LocalDate.now();  
Period idade = Period.between(nascimento, hoje);  
System.out.println(idade);
```

Saída no console

```
P28Y9M22D
```

No exemplo anterior podemos ver que uma pessoa nascida no dia **6 de agosto de 1990** possuía **28 anos, 9 meses e 22 dias** de idade no dia em que esse código foi executado: **28/05/2019**.

5. **Period** também armazena uma quantidade de dias maior que um mês, e de meses maior que um ano.

src/org/j6toj8/localization/datetime/period/Period_BiggerValues.java

```
System.out.println(Period.of(0, 60, 2)); // 60 meses e 2 dias  
System.out.println(Period.of(0, 30, 50)); // 30 meses e 50 dias  
System.out.println(Period.of(5, 200, 1000)); // 5 anos, 200 meses e 1000 dias
```

Saída no console

```
P60M2D  
P30M50D  
P5Y200M1000D
```

Perceba que **Period** não converteu meses em anos, ou dias em meses.

Curiosidades!

É possível normalizar um `Period` que não está com o intervalo padrão através do método `normalized`. Mas serão considerados apenas os anos e meses, deixando os dias intactos.

src/org/j6toj8/localization/datetime/period/Period_Curiosities.java



```
System.out.println(Period.of(0, 60, 2)); // P60M2D  
System.out.println(Period.of(0, 60, 2).normalized()); // P5Y2D  
  
System.out.println(Period.of(0, 30, 50)); // P30M50D  
System.out.println(Period.of(0, 30, 50).normalized()); // P2Y6M50D
```

Também é possível trabalhar com números negativos e normaliza-los.

src/org/j6toj8/localization/datetime/period/Period_Curiosities.java

```
System.out.println(Period.of(1, -25, 0)); // P1Y-25M  
System.out.println(Period.of(1, -25, 0).normalized()); // P-1Y-1M
```

6. É possível somar um `Period` a um `Instant`, `LocalDate` ou `LocalDateTime`.

src/org/j6toj8/localization/datetime/period/Period_Compatibility.java

```
Period period = Period.ofDays(10);  
System.out.println("Period: " + period);  
  
Instant instant = Instant.ofEpochSecond(1558962061L);  
System.out.println("\nInstant: " + instant);  
System.out.println("Instant + Period: " + instant.plus(period));  
  
LocalDate localDate = LocalDate.of(2018, 5, 27);  
System.out.println("\nLocalDate: " + localDate);  
System.out.println("LocalDate + Period: " + localDate.plus(period));  
  
LocalDateTime localDateTime = LocalDateTime.of(2018, 05, 27, 13, 1, 1);  
System.out.println("\nLocalDateTime: " + localDateTime);  
System.out.println("LocalDateTime + Period: " + localDateTime.plus(period));
```

Saída no console

```
Period: P10D
```

```
Instant: 2019-05-27T13:01:01Z
```

```
Instant + Period: 2019-06-06T13:01:01Z
```

```
LocalDate: 2018-05-27
```

```
LocalDate + Period: 2018-06-06
```

```
LocalDateTime: 2018-05-27T13:01:01
```

```
LocalDateTime + Period: 2018-06-06T13:01:01
```

7. **Não** é possível somar um **Period** a um **LocalTime**, pois o primeiro representa tempo baseado em datas, e o segundo armazena apenas horários.

src/org/j6toj8/localization/datetime/period/Period_LocalTime.java

```
Period period = Period.ofDays(13);
System.out.println("Period: " + period);

LocalTime localTime = LocalTime.of(13, 1, 1);
System.out.println("LocalTime: " + localTime);
System.out.println("LocalTime + Period: " + localTime.plus(period));
```

Saída no console

```
Period: P13D
```

```
LocalTime: 13:01:01
```

```
Exception in thread "main" java.time.temporal.UnsupportedTemporalTypeException:
```

```
Unsupported unit: Days
```

```
at java.time.LocalTime.plus(LocalTime.java:1055)
at java.time.LocalTime.plus(LocalTime.java:125)
at java.time.Period.addTo(Period.java:906)
at java.time.LocalTime.plus(LocalTime.java:988)
at
```

```
org.j6toj8.localization.datetime.period.Period_LocalTime.main(Period_LocalTime.java
:15)
```

Veja que ocorreu exceção ao tentar somar um **Period** a um **LocalTime**.

8. **Não** é possível somar um **Period** que contém meses ou anos a um **Instant**. Como vimos na seção anterior sobre **Instant**, ele não suporta operações com valores maiores que dias.

```
Period period = Period.ofMonths(1);
System.out.println("Period: " + period);

Instant instant = Instant.ofEpochSecond(1558962061L);
System.out.println("Instant: " + instant);
System.out.println("Instant + Period: " + instant.plus(period));
```

Saída no console

```
Period: P1M
Instant: 2019-05-27T13:01:01Z
Exception in thread "main" java.time.temporal.UnsupportedTemporalTypeException:
Unsupported unit: Months
    at java.time.Instant.plus(Instant.java:862)
    at java.time.Instant.plus(Instant.java:207)
    at java.time.Period.addTo(Period.java:902)
    at java.time.Instant.plus(Instant.java:788)
    at
org.j6toj8.localization.datetime.period.Period_Instant.main(Period_Instant.java:15)
```

Duration

Uma **Duration** é uma quantidade de tempo baseado em hora, como segundos, minutos ou horas. Por exemplo: **2 horas, 3 minutos e 4 segundos.**

1. A **Duration** é apresentada no formato **PTxHxMxS**, onde:
 - **PT** representa que é um **Duration**.
 - **xH** representa a quantidade de horas.
 - **xM** representa a quantidade de minutos.
 - **xS** representa a quantidade de segundos.
2. Assim como um **Period**, ao imprimir uma **Duration** apenas os campos com valor são apresentados. Campos zerados são omitidos.
3. É possível criar uma **Duration** através dos métodos **static** chamados **of***.

src/org/j6toj8/localization/datetime/duration/Duration_Of.java

```
System.out.println(Duration.ofNanos(2)); // 2 nanossegundos
System.out.println(Duration.ofMillis(2)); // 2 milissegundos
System.out.println(Duration.ofSeconds(2)); // 2 segundos
System.out.println(Duration.ofMinutes(2)); // 2 minutos
System.out.println(Duration.ofHours(2)); // 2 horas
System.out.println(Duration.ofDays(2)); // 2 dias (48 horas)
System.out.println(Duration.ofSeconds(2, 200)); // 2,000002 segundos (2 segundos e
200 nanossegundos)
System.out.println(Duration.of(2, ChronoUnit.MICROS)); // 2 microssegundos
```

Saída no console

```
PT0.00000002S
PT0.002S
PT2S
PT2M
PT2H
PT48H
PT2.0000002S
PT0.00002S
```

Perceba que **2 dias** é armazenado como 48 horas, pois **Duration** não armazena dias.

4. Também é possível criar uma **Duration** a partir de dois **LocalTime**.

src/org/j6toj8/localization/datetime/duration/Duration_Between.java

```
LocalTime meiaNoite = LocalTime.of(0, 0, 0);
LocalTime meioDia = LocalTime.of(12, 0, 0);

System.out.println(Duration.between(meiaNoite, meioDia));

System.out.println(Duration.between(meioDia, meiaNoite));

System.out.println(Duration.between(meioDia, meioDia));
```

Saída no console

```
PT12H
PT-12H
PT0S
```

5. **Duration** converte automaticamente segundos em minutos e minutos em horas.

src/org/j6toj8/localization/datetime/duration/Duration_BiggerValues.java

```
System.out.println(Duration.ofMinutes(120)); // 120 minutos
System.out.println(Duration.ofMinutes(121)); // 2 horas e 1 minuto
System.out.println(Duration.ofMinutes(119)); // 1 hora e 59 minutos
System.out.println(Duration.ofSeconds(10000)); // 2 horas, 46 minutos e 40 segundos
```

Saída no console

```
PT2H
PT2H1M
PT1H59M
PT2H46M40S
```

6. É possível somar uma **Duration** a um **Instant**, **LocalTime** ou **LocalDateTime**.

src/org/j6toj8/localization/datetime/duration/Duration_Compatibility.java

```
Duration period = Duration.ofHours(2);
System.out.println("Duration: " + period);

Instant instant = Instant.ofEpochSecond(1558962061L);
System.out.println("\nInstant: " + instant);
System.out.println("Instant + Duration: " + instant.plus(period));

LocalTime localTime = LocalTime.of(12, 5, 2);
System.out.println("\nLocalTime: " + localTime);
System.out.println("LocalTime + Duration: " + localTime.plus(period));

LocalDateTime localDateTime = LocalDateTime.of(2018, 05, 27, 13, 1, 1);
System.out.println("\nLocalDateTime: " + localDateTime);
System.out.println("LocalDateTime + Duration: " + localDateTime.plus(period));
```

Saída no console

```
Duration: PT2H

Instant: 2019-05-27T13:01:01Z
Instant + Duration: 2019-05-27T15:01:01Z

LocalTime: 12:05:02
LocalTime + Duration: 14:05:02

LocalDateTime: 2018-05-27T13:01:01
LocalDateTime + Duration: 2018-05-27T15:01:01
```

7. **Não** é possível somar uma **Duration** a uma **LocalDate**, pois o primeiro representa tempo baseado em hora, enquanto o outro armazena apenas datas.

```
Duration duration = Duration.ofHours(2);
System.out.println("Duration: " + duration);

LocalDate localDate = LocalDate.of(1990, 8, 6);
System.out.println("LocalTime: " + localDate);
System.out.println("LocalTime + Period: " + localDate.plus(duration));
```

Saída no console

```
Duration: PT2H
LocalTime: 1990-08-06
Exception in thread "main" java.time.temporal.UnsupportedTemporalTypeException:
Unsupported unit: Seconds
    at java.time.LocalDate.plus(LocalDate.java:1247)
    at java.time.LocalDate.plus(LocalDate.java:137)
    at java.time.Duration.addTo(Duration.java:1070)
    at java.time.LocalDate.plus(LocalDate.java:1149)
    at
org.j6toj8.localization.datetime.duration.Duration_LocalDate.main(Duration_LocalDat
e.java:15)
```

Veja que ocorreu exceção ao tentar somar uma **Duration** a uma **LocalDate**.

Referências

- Working with Dates and Times
Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 234). Wiley. Edição do Kindle.
- [Introduction to the Java 8 Date/Time API](#).
- [Trail: Date Time: Table of Contents](#). The Java™ Tutorials.
- [What's the difference between Instant and LocalDateTime?](#)

Formatação de Números e Datas

Objetivo

Format dates, numbers, and currency values for localization with the `NumberFormat` and `DateFormat` classes, including number and date format patterns.

-
Formatar datas, números e valores monetários para localização utilizando as classes `NumberFormat` e `DateFormat`, incluindo padrões de formato de número e data.

Ainda dentro do assunto de Localização e Internacionalização, é comum a necessidade de apresentar datas, números e valores monetários em diferentes formatos.

O exame de certificação comprehende cinco classes principais de formatação:

- **NumberFormat** → Formatação geral de números, valores monetários, percentuais e números inteiros com arredondamento, possivelmente baseados em **Locale**.
- **DecimalFormat** → Formatação de números quando há necessidade de definições mais específicas ou personalizadas do formato.
- **DateFormat** → Formatação de data e hora antes do Java 8 utilizando formatos predefinidos.
- **SimpleDateFormat** → Formatação de data e hora antes do Java 8 utilizando formatos mais específicos ou personalizados.
- **DateTimeFormatter** → Formatação de data e hora após o Java 8.

NumberFormat

1. É possível obter uma instância de **NumberFormat** a partir de vários métodos estáticos, dependendo da necessidade.

src/org/j6toj8/localization/formats/numberformat/NumberFormat_Instance.java

```
// sem Locale
NumberFormat instance1 = NumberFormat.getInstance();
NumberFormat instance2 = NumberFormat.getNumberInstance(); // igual a getInstance()
NumberFormat instance3 = NumberFormat.getCurrencyInstance();
NumberFormat instance4 = NumberFormat.getPercentInstance();

// com Locale
NumberFormat instance5 = NumberFormat.getInstance(new Locale("pt", "BR"));
NumberFormat instance6 = NumberFormat.getNumberInstance(new Locale("pt", "BR"));
NumberFormat instance7 = NumberFormat.getCurrencyInstance(new Locale("pt", "BR"));
NumberFormat instance8 = NumberFormat.getPercentInstance(new Locale("pt", "BR));
```

Lembre-se que, se não for informado o **Locale**, será utilizado o padrão. O ideal é sempre informar o **Locale**.

2. O **NumberFormat** pode ser utilizado para transformar números em Strings.

src/org/j6toj8/localization/formats/numberformat/NumberFormat_NumberToString.java

```
NumberFormat numberFormatPtBR = NumberFormat.getInstance(new Locale("pt", "BR"));
NumberFormat numberFormatEnUS = NumberFormat.getInstance(new Locale("en", "US"));
NumberFormat numberFormatFrFR = NumberFormat.getInstance(new Locale("fr", "FR"));

double d = 1000.05;

System.out.println("pt_BR: " + numberFormatPtBR.format(d));
System.out.println("en_US: " + numberFormatEnUS.format(d));
System.out.println("fr_FR: " + numberFormatFrFR.format(d));
```

Saída no console

```
pt_BR: 1.000,05
en_US: 1,000.05
fr_FR: 1 000,05
```

Perceba que a representação do número muda de acordo com o **Locale**.

3. O **NumberFormat** por ser utilizado para transformar Strings em números.

src/org/j6toj8/localization/formats/numberformat/NumberFormat_StringToNumber.java

```
NumberFormat numberFormatPtBR = NumberFormat.getInstance(new Locale("pt", "BR"));
NumberFormat numberFormatEnUS = NumberFormat.getInstance(new Locale("en", "US"));
NumberFormat numberFormatFrFR = NumberFormat.getInstance(new Locale("fr", "FR"));

String s = "1000,05";

try {
    Number parsePtBR = numberFormatPtBR.parse(s);
    Number parseEnUS = numberFormatEnUS.parse(s);
    Number parseFrFR = numberFormatFrFR.parse(s);

    System.out.println("pt_BR: " + parsePtBR);
    System.out.println("en_US: " + parseEnUS);
    System.out.println("fr_FR: " + parseFrFR);
} catch (ParseException e) {
    // trate a exceção no parse
}
```

Saída no console

```
pt_BR: 1000.05
en_US: 100005
fr_FR: 1000.05
```

Perceba que dependendo do **Locale** estamos representando um número diferente, e isso muda o

resultado do parse.

4. O `NumberFormat` pode ser utilizado para transformar Strings em valores monetários, e vice-versa.

`src/org/j6toj8/localization/formats/numberformat/NumberFormat_Currency.java`

```
NumberFormat currencyFormatPtBR = NumberFormat.getCurrencyInstance(new Locale("pt", "BR"));
NumberFormat currencyFormatEnUS = NumberFormat.getCurrencyInstance(new Locale("en", "US"));
NumberFormat currencyFormatFrFR = NumberFormat.getCurrencyInstance(new Locale("fr", "FR"));

// Valor monetário para String
double d = 1000.05;

System.out.println("pt_BR: " + currencyFormatPtBR.format(d));
System.out.println("en_US: " + currencyFormatEnUS.format(d));
System.out.println("fr_FR: " + currencyFormatFrFR.format(d));

// String para valor Monetário
String s = "R$ 1000,05";

try {
    System.out.println("pt_BR: " + currencyFormatPtBR.parse(s));
} catch (ParseException e) {
    System.out.println(e.getMessage());
}

try {
    System.out.println("en_US: " + currencyFormatEnUS.parse(s));
} catch (ParseException e) {
    System.out.println(e.getMessage());
}

try {
    System.out.println("fr_FR: " + currencyFormatFrFR.parse(s));
} catch (ParseException e) {
    System.out.println(e.getMessage());
}
```

Saída no console

```
pt_BR: R$ 1.000,05
en_US: $1,000.05
fr_FR: 1 000,05 €
pt_BR: 1000.05
Unparseable number: "R$ 1000,05"
Unparseable number: "R$ 1000,05"
```

Perceba que novamente o resultado muda de acordo com o `Locale`. Além disso, não é possível converter a representação da moeda brasileira com um `Locale en_US` ou `fr_FR`.

- O `NumberFormat` pode ser utilizado para transformar Strings em percentuais, e vice-versa.

`src/org/j6toj8/localization/formats/numberformat/NumberFormat_Percent.java`

```
NumberFormat percentFormatPtBR = NumberFormat.getInstance(new Locale("pt",  
"BR"));  
NumberFormat percentFormatEnUS = NumberFormat.getInstance(new Locale("en",  
"US"));  
NumberFormat percentFormatFrFR = NumberFormat.getInstance(new Locale("fr",  
"FR"));  
  
// Percentual para String  
double d = 0.9;  
  
System.out.println("pt_BR: " + percentFormatPtBR.format(d));  
System.out.println("en_US: " + percentFormatEnUS.format(d));  
System.out.println("fr_FR: " + percentFormatFrFR.format(d));  
  
// String para Percentual  
String s = "80%";  
  
try {  
    System.out.println("pt_BR: " + percentFormatPtBR.parse(s));  
    System.out.println("en_US: " + percentFormatEnUS.parse(s));  
    System.out.println("fr_FR: " + percentFormatFrFR.parse(s));  
} catch (ParseException e) {  
    System.out.println(e.getMessage());  
}
```

Saída no console

```
pt_BR: 90%  
en_US: 90%  
fr_FR: 90 %  
pt_BR: 0.8  
en_US: 0.8  
Unparseable number: "80%"
```

Veja que, ao formatar, `100%` é `1`, logo `80%` é `0,8`. Além disso, no `Locale fr_FR` a representação `80%` não é válida.

- O `NumberFormat` pode ficar complicado ao lidar com vírgulas.

```
NumberFormat percentFormatPtBR = NumberFormat.getPercentInstance(new Locale("pt",  
"BR"));  
NumberFormat percentFormatEnUS = NumberFormat.getPercentInstance(new Locale("en",  
"US"));  
  
// String para Percentual  
String s = "80,2%";  
  
try {  
    System.out.println("pt_BR: " + percentFormatPtBR.parse(s));  
    System.out.println("en_US: " + percentFormatEnUS.parse(s));  
} catch (ParseException e) {  
    // trate a exceção de parse  
}
```

Saída no console

```
pt_BR: 0.802  
en_US: 8.02
```

No `Locale pt_BR` temos o resultado esperado. Porém, no `Locale en_US` o `80,2%` se torna `802%` pois a vírgula não é usada como separador de decimal.

DecimalFormat

Enquanto `NumberFormat` permite utilizar formatos predefinidos, `DecimalFormat` permite uma personalização maior. Um exemplo de formato para o `DecimalFormat` é `###,###.###`.

- `#` → preenche a posição com um número, ou omite se não houver nada.
- `0` → preenche a posição com um número, ou 0 se não houver nada.
- `.` → indica onde é a posição do separador decimal.
- `,` → indica onde é a posição do separador de grupos.

1. É possível obter uma instância de `DecimalFormat` utilizando o construtor.

```
double d = 12345.67;

// omite todas as posições vazias, utiliza separador a cada 3 casas
DecimalFormat instance1 = new DecimalFormat("###,###.###");
System.out.println("###,###.###: " + instance1.format(d));

// omite as posições vazias na parte decimal, utiliza separador a cada 3 casas
DecimalFormat instance2 = new DecimalFormat("000,000.###");
System.out.println("000,000.###: " + instance2.format(d));

// omite as posições vazias na parte inteira, utiliza separador a cada 3 casas
DecimalFormat instance3 = new DecimalFormat("###,###.000");
System.out.println("###,###.000: " + instance3.format(d));

// apresenta todas as posições, utiliza separador a cada 3 casas
DecimalFormat instance4 = new DecimalFormat("000,000.000");
System.out.println("000,000.000: " + instance4.format(d));

// omite todas as posições vazias, não utiliza separador
DecimalFormat instance5 = new DecimalFormat("###.##");
System.out.println("###.##: " + instance5.format(d));

// apresenta todas as posições, não utiliza separador
DecimalFormat instance6 = new DecimalFormat("00000.000");
System.out.println("00000.000: " + instance6.format(d));

// omite todas as posições vazias, não utiliza separador e não apresenta casas
// decimais
DecimalFormat instance7 = new DecimalFormat("###");
System.out.println("###: " + instance7.format(d));
```

Saída no console

```
###,###.###: 12.345,67
000,000.###: 012.345,67
###,###.000: 12.345,670
000,000.000: 012.345,670
###.##: 12345,67
00000.000: 012345,670
###: 12346
```

Estou executando o código onde o **Locale** padrão é **pt_BR**, por isso a saída no console apresenta vírgulas para separar grupos e pontos para separar os decimais.

2. Para utilizar um **Locale** específico é necessário instanciar um **NumberFormat** e fazer um *cast* para **DecimalFormat**.

src/org/j6toj8/localization/formats/decimalformat/DecimalFormat_Locale.java

```
double d = 12345.67;

NumberFormat numberFormatUS = NumberFormat.getNumberInstance(Locale.US);
DecimalFormat decimalFormatUS = (DecimalFormat) numberFormatUS;
decimalFormatUS.applyLocalizedPattern("###,###.###");
System.out.println(decimalFormatUS.format(d));
```

Saída no console

```
12,345.67
```

3. É possível colocar outros símbolos ou palavras no formato do **DecimalFormat**.

src/org/j6toj8/localization/formats/decimalformat/DecimalFormat_Strings.java

```
double d = 12345.67;

DecimalFormat decimalFormat = new DecimalFormat("Número ###,###.### formatado");
System.out.println(decimalFormat.format(d));
```

Saída no console

```
Número 12.345,67 formatado
```

DateFormat

Para formatar Data e Hora antes do Java 8, são utilizadas as classes de **DateFormat** e **SimpleDateFormat**. Essas classes trabalham em geral com a classe **Date** e fazem parte do pacote **java.text**, diferente de **DateTimeFormatter** que é do novo pacote **java.time.format**.

1. Obter instâncias de **DateFormat** é muito parecido com **NumberFormat**.
2. É possível definir o formato das instâncias como **SHORT**, **MEDIUM**, **LONG** ou **FULL**.
3. É possível definir o **Locale** das instâncias.

src/org/j6toj8/localization/formats/dateformat/DateFormat_Instance.java

```
DateFormat dateInstance = DateFormat.getDateInstance();
DateFormat timeInstance = DateFormat.getTimeInstance();
DateFormat dateTimeInstance = DateFormat.getDateInstance();
DateFormat dateTimeLongInstance = DateFormat.getDateInstance(DateFormat.LONG,
DateFormat.LONG);
DateFormat dateTimeUSInstance = DateFormat.getDateInstance(DateFormat.LONG,
DateFormat.LONG, Locale.US);

Date date = new Date(1000000000000L); // data em quantidade de milissegundos desde
01/01/1970

System.out.println(dateInstance.format(date));
System.out.println(timeInstance.format(date));
System.out.println(dateTimeInstance.format(date));
System.out.println(dateTimeLongInstance.format(date));
System.out.println(dateTimeUSInstance.format(date));
```

Saída no console

```
08/09/2001
22:46:40
08/09/2001 22:46:40
8 de Setembro de 2001 22h46min40s BRT
September 8, 2001 10:46:40 PM BRT
```

4. Também é possível transformar **String** em **Date** utilizando o método **parse**.

src/org/j6toj8/localization/formats/dateformat/DateFormat_Parse.java

```
DateFormat dateInstance = DateFormat.getDateInstance(DateFormat.SHORT);
DateFormat timeInstance = DateFormat.getTimeInstance(DateFormat.SHORT);
DateFormat dateTimeInstance = DateFormat.getDateInstance(DateFormat.SHORT,
DateFormat.SHORT);

String date = "08/09/2001";
String time = "22:46:40";
String dateTime = "08/09/2001 22:46:40";

try {
    System.out.println(dateInstance.parse(date));
    System.out.println(timeInstance.parse(time));
    System.out.println(dateTimeInstance.parse(dateTime));
    System.out.println(dateTimeInstance.parse(date)); // exceção, pois date não tem
hora
} catch (ParseException e) {
    System.out.println(e.getMessage());
}
```

Saída no console

```
Sat Sep 08 00:00:00 BRT 2001
Thu Jan 01 22:46:00 BRT 1970
Sat Sep 08 22:46:00 BRT 2001
Unparseable date: "08/09/2001"
```

Perceba que ocorreu uma exceção ao tentar converter uma data utilizando um formatação que espera Data e Hora.

SimpleDateFormat

A classe `SimpleDateFormat` permite criar formatos mais personalizados para apresentar Data e Hora, como `dd/MM/yyyy HH:mm:ss`. A seguir, as letras mais importantes utilizadas na formatação para o exame:

- `y` → Ano (2019, 19)
- `M` → Mês (8, 08, Ago, Agosto)
- `d` → Dia (06)
- `h` → Hora em formato AM/PM
- `H` → Hora em formato 24H
- `m` → Minutos
- `s` → Segundos

Em geral (existem exceções), quanto mais letras forem utilizadas, mais extenso é o formato apresentado. Por exemplo:

- `M` → 8
- `MM` → 08
- `MMM` → Ago
- `MMMM` → Agosto

1. É possível criar formatos personalizados para formatar um `Date` utilizando o construtor de `SimpleDateFormat`.

src/org/j6toj8/localization/formats/simpledateformat/SimpleDateFormat_Instance.java

```
SimpleDateFormat simpleDateTime = new SimpleDateFormat("dd MM yy - HH mm ss");
SimpleDateFormat simpleDate = new SimpleDateFormat("dd MM yy");
SimpleDateFormat simpleTime = new SimpleDateFormat("HH mm ss");

Date date = new Date(100000000000L); // data em quantidade de milissegundos desde
// 01/01/1970

System.out.println(simpleDateTime.format(date));
System.out.println(simpleDate.format(date));
System.out.println(simpleTime.format(date));
```

Saída no console

```
08 09 01 - 22 46 40
08 09 01
22 46 40
```

2. Também é possível criar `Date` a partir de `String` utilizando o método `parse`.

src/org/j6toj8/localization/formats/simpledateformat/SimpleDateFormat_Parse.java

```
SimpleDateFormat simpleDateTime = new SimpleDateFormat("dd MM yy - HH mm ss");
SimpleDateFormat simpleDate = new SimpleDateFormat("dd MM yy");
SimpleDateFormat simpleTime = new SimpleDateFormat("HH mm ss");

String dateTime = "08 09 01 - 22 46 40";
String date = "08 09 01";
String time = "22 46 40";

try {
    System.out.println(simpleDateTime.parse(dateTime));
    System.out.println(simpleDate.parse(date));
    System.out.println(simpleTime.parse(time));
    System.out.println(simpleDateTime.parse(time)); // exceção, pois time não tem
    // data
} catch (ParseException e) {
    System.out.println(e.getMessage());
}
```

Saída no console

```
Sat Sep 08 22:46:40 BRT 2001
Sat Sep 08 00:00:00 BRT 2001
Thu Jan 01 22:46:40 BRT 1970
Unparseable date: "22 46 40"
```

Perceba a exceção ao tentar utilizar o formato incorreto.

DateTimeFormatter

O Java 8 traz a classe `DateTimeFormatter` que possui várias formas de formatar e transformar **Data/Hora** em **String**, e vice-versa.

1. É possível obter instâncias predefinidas de `DateTimeFormatter`, que representam formatos **ISO** ou **RFC**.

`src/org/j6toj8/localization/formats/datetimeformatter/DateTimeFormatter_Predefined.java`

```
LocalDateTime localDT = LocalDateTime.of(2019, 8, 6, 11, 40);
System.out.println(localDT.format(DateTimeFormatter.ISO_LOCAL_DATE));
System.out.println(localDT.format(DateTimeFormatter.ISO_LOCAL_TIME));
System.out.println(localDT.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
System.out.println(localDT.format(DateTimeFormatter.ISO_ORDINAL_DATE));
System.out.println(localDT.format(DateTimeFormatter.ISO_WEEK_DATE));
```

Saída no console

```
2019-08-06
11:40:00
2019-08-06T11:40:00
2019-218
2019-W32-2
```

2. É possível obter instâncias predefinidas de `DateTimeFormatter`, que representam formatos localizados, através da classe `FormatStyle`.

`src/org/j6toj8/localization/formats/datetimeformatter/DateTimeFormatter_FormatStyle.java`

```
LocalDateTime localDT = LocalDateTime.of(2019, 8, 6, 11, 40);

DateTimeFormatter dateTimeShortFormatter = DateTimeFormatter.
ofLocalizedDateTime(FormatStyle.SHORT);
DateTimeFormatter dateTimeMediumFormatter = DateTimeFormatter.
ofLocalizedDateTime(FormatStyle.MEDIUM);
DateTimeFormatter dateShortFormatter = DateTimeFormatter.
ofLocalizedDate(FormatStyle.SHORT);
DateTimeFormatter timeShortFormatter = DateTimeFormatter.
ofLocalizedTime(FormatStyle.SHORT);

System.out.println(localDT.format(dateTimeShortFormatter));
System.out.println(localDT.format(dateTimeMediumFormatter));
System.out.println(localDT.format(dateShortFormatter));
System.out.println(localDT.format(timeShortFormatter));
```

Saída no console

```
06/08/19 11:40  
06/08/2019 11:40:00  
06/08/19  
11:40
```

O resultado depende de onde o código está sendo executado. Este código foi executado no **Locale** padrão **pt_BR**.

3. É possível alterar o **Locale** utilizado pelo **DateTimeFormatter**.

src/org/j6toj8/localization/formats/datetimeformatter/DateTimeFormatter_Locale.java

```
LocalDateTime localDT = LocalDateTime.of(2019, 8, 6, 11, 40);  
  
DateTimeFormatter shortFormatter = DateTimeFormatter.  
ofLocalizedDateTime(FormatStyle.SHORT);  
DateTimeFormatter mediumFormatter = DateTimeFormatter.  
ofLocalizedDateTime(FormatStyle.MEDIUM);  
  
shortFormatter = shortFormatter.withLocale(Locale.US);  
mediumFormatter = mediumFormatter.withLocale(Locale.US);  
  
System.out.println(localDT.format(shortFormatter));  
System.out.println(localDT.format(mediumFormatter));
```

Saída no console

```
8/6/19 11:40 AM  
Aug 6, 2019 11:40:00 AM
```

4. É possível obter instâncias personalizadas de **DateTimeFormatter**.

src/org/j6toj8/localization/formats/datetimeformatter/DateTimeFormatter_Custom.java

```
LocalDateTime localDT = LocalDateTime.of(2019, 8, 6, 11, 40);  
  
DateTimeFormatter customFormatter = DateTimeFormatter.ofPattern("dd MM yy - HH mm  
ss");  
  
System.out.println(localDT.format(customFormatter));
```

Saída no console

```
06 08 19 - 11 40 00
```

Para criar um **DateTimeFormatter** personalizado é necessário conhecer o que cada letra do

formato representa. Volte na seção de [SimpleDateFormat](#) caso não se lembre.

5. Não é possível formatar uma Data/Hora caso o objeto fornecido não tenha os campos necessários. Um exemplo seria tentar apresentar a Data e fornecer um [LocalTime](#).

[src/org/j6toj8/localization/formats/datetimeformatter/DateTimeFormatter_Error.java](#)

```
LocalDate localDate = LocalDate.of(2019, 8, 6);
LocalTime localTime = LocalTime.of(11, 40);
LocalDateTime localDT = LocalDateTime.of(localDate, localTime);

System.out.println(localDate.format(DateTimeFormatter.ISO_LOCAL_DATE));
System.out.println(localDT.format(DateTimeFormatter.ISO_LOCAL_DATE));

// lança exceção pois não possui campos de data
System.out.println(localTime.format(DateTimeFormatter.ISO_LOCAL_DATE));
```

Saída no console

```
2019-08-06
2019-08-06
Exception in thread "main" java.time.temporal.UnsupportedTemporalTypeException:
Unsupported field: Year
    at java.time.LocalTime.get0(LocalTime.java:679)
    at java.time.LocalTime.getLong(LocalTime.java:656)
    at
java.time.format.DateTimePrintContext$1.getLong(DateTimePrintContext.java:205)
    at
java.time.format.DateTimePrintContext.getValue(DateTimePrintContext.java:298)
    at
java.time.format.DateTimeFormatterBuilder$NumberPrinterParser.format(DateTimeFormat
terBuilder.java:2551)
    at
java.time.format.DateTimeFormatterBuilder$CompositePrinterParser.format(DateTimeFor
matterBuilder.java:2190)
    at java.time.format.DateTimeFormatter.formatTo(DateTimeFormatter.java:1746)
    at java.time.format.DateTimeFormatter.format(DateTimeFormatter.java:1720)
    at java.time.LocalTime.format(LocalTime.java:1413)
    at
org.j6toj8.localization.formats.datetimeformatter.DateTimeFormatter_Error.main(Date
TimeFormatter_Error.java:18)
```

6. O mesmo erro pode ocorrer ao utilizar um formato personalizado.

```
LocalDate localDate = LocalDate.of(2019, 8, 6);
LocalTime localTime = LocalTime.of(11, 40);
LocalDateTime localDT = LocalDateTime.of(localDate, localTime);

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("HH mm ss");

System.out.println(localDT.format(formatter));
System.out.println(localTime.format(formatter));
System.out.println(localDate.format(formatter)); // lança exceção pois não possui
campos de hora
```

Saída no console

```
11 40 00
11 40 00
Exception in thread "main" java.time.temporal.UnsupportedTemporalTypeException:
Unsupported field: HourOfDay
    at java.time.LocalDate.get0(LocalDate.java:680)
    at java.time.LocalDate.getLong(LocalDate.java:659)
    at
java.time.format.DateTimePrintContext.getValue(DateTimePrintContext.java:298)
    at
java.time.format.DateTimeFormatterBuilder$NumberPrinterParser.format(DateTimeFormat
terBuilder.java:2551)
    at
java.time.format.DateTimeFormatterBuilder$CompositePrinterParser.format(DateTimeFor
matterBuilder.java:2190)
    at java.time.format.DateTimeFormatter.formatTo(DateTimeFormatter.java:1746)
    at java.time.format.DateTimeFormatter.format(DateTimeFormatter.java:1720)
    at java.time.LocalDate.format(LocalDate.java:1691)
    at
org.j6toj8.localization.formats.datetimeformatter.DateTimeFormatter_ErrorCustom.mai
n(DateTimeFormatter_ErrorCustom.java:20)
```

Nesse caso é lançada exceção porque **LocalDate** não possui hora, minuto ou segundo.

7. Também é possível fazer o oposto: converter uma **String** em uma classe de **Data/Hora**. Para isso existem os métodos **parse**.

src/org/j6toj8/localization/formats/datetimeformatter/DateTimeFormatter_Parse.java

```
DateTimeFormatter customFormatter = DateTimeFormatter.ofPattern("dd MM yy - HH mm ss");
DateTimeFormatter isoFormatter = DateTimeFormatter.ISO_LOCAL_DATE_TIME;
DateTimeFormatter shortFormatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);

LocalDateTime parseCustom = LocalDateTime.parse("06 08 19 - 11 40 00",
customFormatter);
LocalDateTime parseIso = LocalDateTime.parse("2019-08-06T11:40:00", isoFormatter);
LocalDateTime parseShort = LocalDateTime.parse("06/08/19 11:40", shortFormatter);

System.out.println(parseCustom);
System.out.println(parseIso);
System.out.println(parseShort);
```

Saída no console

```
2019-08-06T11:40
2019-08-06T11:40
2019-08-06T11:40
```

8. Todos os métodos utilizados para **format** e **parse** também podem ser invocados diretamente na instância do **DateTimeFormatter**.

src/org/j6toj8/localization/formats/datetimeformatter/DateTimeFormatter_Inverted.java

```
LocalDateTime localDT = LocalDateTime.of(2019, 8, 6, 11, 40);
System.out.println(localDT);

String format = DateTimeFormatter.ISO_LOCAL_DATE_TIME.format(localDT);
System.out.println(format);

TemporalAccessor parse = DateTimeFormatter.ISO_LOCAL_DATE_TIME.parse(format);
System.out.println(parse);
System.out.println(LocalDateTime.from(parse));
```

Saída no console

```
2019-08-06T11:40
2019-08-06T11:40:00
{},ISO resolved to 2019-08-06T11:40
2019-08-06T11:40
```

Porém, veja que ao utilizar o método **parse** diretamente no **DateTimeFormatter**, é necessário converter o resultado para um **LocalDateTime**.

Referências

- Adding Internationalization and Localization

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 255). Wiley. Edição do Kindle.

- Formatting and Parsing

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 603). Wiley. Edição do Kindle.

- [Change date format in a Java string.](#) BalusC.
- [Introduction to the Java 8 Date/Time API.](#)
- [Guide to DateTimeFormatter.](#)
- [A Practical Guide to DecimalFormat.](#)
- [A Guide to SimpleDateFormat.](#)
- [Lesson: Formatting](#) The Java™ Tutorials.
- [Parsing and Formatting.](#) The Java™ Tutorials.
- [Formatting Numeric Print Output](#) The Java™ Tutorials.

Fusos Horários

Objetivo

Work with dates and times across time zones and manage changes resulting from daylight savings.

- Trabalhe com datas e horas entre fusos horários e gerencie alterações resultantes de horários de verão.

O exame de certificação espera que você consiga lidar com diferentes fusos horários, além de conseguir entender o que ocorre ao realizar operações com datas e horas que passam por um horário de verão.

Nessa seção será apresentada a classe `ZonedDateTime`, que armazena Data, Hora e um fuso horário. Ou seja, é muito parecida com a classe `LocalDateTime`, porém possui um fuso horário.

A representação de um `ZonedDateTime` é `2007-12-03T10:15:30+01:00 Europe/Paris`.

1. É possível criar um `ZonedDateTime` através do método `static` chamado `now`.

`src/org/j6toj8/localization/timezones/ZonedDateTime_Now.java`

```
System.out.println(ZonedDateTime.now());
```

Saída no console

```
2019-06-09T18:10:08.863-03:00[America/Sao_Paulo]
```

A saída no console irá apresentar a data e hora atual, e o fuso horário de onde o código está sendo executado.

2. Também é possível criar um `ZonedDateTime` através do método `static` chamado `of`.

`src/org/j6toj8/localization/timezones/ZonedDateTime.Of.java`

```
System.out.println(ZonedDateTime.of(2019, 6, 9, 13, 20, 3, 1000, ZoneId.of("America/Sao_Paulo")));

LocalDateTime localDateTime = LocalDateTime.of(2019, 6, 9, 13, 20, 3, 1000);
System.out.println(ZonedDateTime.of(localDateTime, ZoneId.of("America/Sao_Paulo")));

LocalDate localDate = LocalDate.of(2019, 6, 9);
LocalTime localTime = LocalTime.of(13, 20, 3, 1000);
System.out.println(ZonedDateTime.of(localDate, localTime, ZoneId.of("America/Sao_Paulo")));
```

Saída no console

```
2019-06-09T13:20:03.000001-03:00[America/Sao_Paulo]
2019-06-09T13:20:03.000001-03:00[America/Sao_Paulo]
2019-06-09T13:20:03.000001-03:00[America/Sao_Paulo]
```

Perceba que é possível criar um `ZonedDateTime` diretamente, ou através de um `LocalDateTime`, ou com uma junção de um `LocalDate` e um `LocalTime`. Note que em todos os casos é necessário informar o fuso horário.

3. Assim como todas as novas classes de data e hora, não é possível criar uma instância de `ZonedDateTime` utilizando o construtor.

`src/org/j6toj8/localization/timezones/ZonedDateTime_Constructor.java`

```
ZonedDateTime zonedDateTime = new ZonedDateTime(); // NÃO COMPILA! - não possui
construtor público
System.out.println(zonedDateTime);
```

4. Será lançada a exceção `DateTimeException` ao tentar criar uma data inválida.

`src/org/j6toj8/localization/timezones/ZonedDateTime_Invalid.java`

```
System.out.println(ZonedDateTime.of(2019, 4, 31, 9, 20, 3, 1000, ZoneId.of("America/Sao_Paulo"))); // lança exceção: não existe 31 de abril
```

Saída no console

```
Exception in thread "main" java.time.DateTimeException: Invalid date 'APRIL 31'
  at java.time.LocalDate.create(LocalDate.java:431)
  at java.time.LocalDate.of(LocalDate.java:269)
  at java.time.LocalDateTime.of(LocalDateTime.java:361)
  at java.time.ZonedDateTime.of(ZonedDateTime.java:339)
  at
org.j6toj8.localization.timezones.ZonedDateTime_Invalid.main(ZonedDateTime_Invalid.java:10)
```

5. Existem vários métodos para somar e subtrair de `ZonedDateTime`

`src/org/j6toj8/localization/timezones/ZonedDateTime_Manipulate.java`

```
ZonedDateTime zonedDateTime = ZonedDateTime.of(2019, 5, 20, 9, 20, 12, 1000,
ZoneId.of("America/Sao_Paulo"));
System.out.println(zonedDateTime);
System.out.println("+2 horas: " + zonedDateTime.plusHours(2));
System.out.println("+2 minutos: " + zonedDateTime.plusMinutes(2));
System.out.println("+2 segundos: " + zonedDateTime.plusSeconds(2));
System.out.println("+2 nanosegundos: " + zonedDateTime.plusNanos(2));
System.out.println("+2 microsegundos: " + zonedDateTime.plus(2, ChronoUnit.
MICROS));
System.out.println("+2 milissegundos: " + zonedDateTime.plus(2, ChronoUnit.
MILLIS));
System.out.println("-2 horas: " + zonedDateTime.minusHours(2));
System.out.println("-2 minutos: " + zonedDateTime.minusMinutes(2));
System.out.println("-2 segundos: " + zonedDateTime.minusSeconds(2));
System.out.println("-2 nanosegundos: " + zonedDateTime.minusNanos(2));
System.out.println("-2 microsegundos: " + zonedDateTime.minus(2, ChronoUnit.
MICROS));
System.out.println("-2 milissegundos: " + zonedDateTime.minus(2, ChronoUnit.
MILLIS));
System.out.println("+2 dias: " + zonedDateTime.plusDays(2));
System.out.println("+2 semanas: " + zonedDateTime.plusWeeks(2));
System.out.println("+2 meses: " + zonedDateTime.plusMonths(2));
System.out.println("+2 anos: " + zonedDateTime.plusYears(2));
System.out.println("+2 anos: " + zonedDateTime.plusYears(2));
System.out.println("+2 décadas: " + zonedDateTime.plus(2, ChronoUnit.DECADES));
System.out.println("-2 dias: " + zonedDateTime.minusDays(2));
System.out.println("-2 semanas: " + zonedDateTime.minusWeeks(2));
System.out.println("-2 meses: " + zonedDateTime.minusMonths(2));
System.out.println("-2 anos: " + zonedDateTime.minusYears(2));
System.out.println("-2 décadas: " + zonedDateTime.minus(2, ChronoUnit.DECADES));
```

Saída no console

```
2019-05-20T09:20:12.000001-03:00[America/Sao_Paulo]
+2 horas: 2019-05-20T11:20:12.000001-03:00[America/Sao_Paulo]
+2 minutos: 2019-05-20T09:22:12.000001-03:00[America/Sao_Paulo]
+2 segundos: 2019-05-20T09:20:14.000001-03:00[America/Sao_Paulo]
+2 nanosegundos: 2019-05-20T09:20:12.000001002-03:00[America/Sao_Paulo]
+2 microsegundos: 2019-05-20T09:20:12.000003-03:00[America/Sao_Paulo]
+2 milissegundos: 2019-05-20T09:20:12.002001-03:00[America/Sao_Paulo]
-2 horas: 2019-05-20T07:20:12.000001-03:00[America/Sao_Paulo]
-2 minutos: 2019-05-20T09:18:12.000001-03:00[America/Sao_Paulo]
-2 segundos: 2019-05-20T09:20:10.000001-03:00[America/Sao_Paulo]
-2 nanosegundos: 2019-05-20T09:20:12.000000998-03:00[America/Sao_Paulo]
-2 microsegundos: 2019-05-20T09:20:11.999999-03:00[America/Sao_Paulo]
-2 milissegundos: 2019-05-20T09:20:11.998001-03:00[America/Sao_Paulo]
+2 dias: 2019-05-22T09:20:12.000001-03:00[America/Sao_Paulo]
+2 semanas: 2019-06-03T09:20:12.000001-03:00[America/Sao_Paulo]
+2 meses: 2019-07-20T09:20:12.000001-03:00[America/Sao_Paulo]
+2 anos: 2021-05-20T09:20:12.000001-03:00[America/Sao_Paulo]
+2 anos: 2021-05-20T09:20:12.000001-03:00[America/Sao_Paulo]
+2 décadas: 2039-05-20T09:20:12.000001-03:00[America/Sao_Paulo]
-2 dias: 2019-05-18T09:20:12.000001-03:00[America/Sao_Paulo]
-2 semanas: 2019-05-06T09:20:12.000001-03:00[America/Sao_Paulo]
-2 meses: 2019-03-20T09:20:12.000001-03:00[America/Sao_Paulo]
-2 anos: 2017-05-20T09:20:12.000001-03:00[America/Sao_Paulo]
-2 décadas: 1999-05-20T09:20:12.000001-03:00[America/Sao_Paulo]
```

6. `ZonedDateTime` é imutável, então é necessário armazenar o retorno de uma alteração em uma variável.

`src/org/j6toj8/localization/timezones/ZonedDateTime_Immutability.java`

```
ZonedDateTime zonedDateTime = ZonedDateTime.of(2019, 5, 20, 9, 20, 3, 300,
ZoneId.of("America/Sao_Paulo"));
System.out.println(zonedDateTime);
zonedDateTime.plusHours(1); // chamada perdida - a nova data/hora não foi
armazenada em uma variável
System.out.println(zonedDateTime);
zonedDateTime = zonedDateTime.plusHours(1); // chamada útil - data/hora armazenada
na variável
System.out.println(zonedDateTime);
```

Saída no console

```
2019-05-20T09:20:03.00000300-03:00[America/Sao_Paulo]
2019-05-20T09:20:03.00000300-03:00[America/Sao_Paulo]
2019-05-20T10:20:03.00000300-03:00[America/Sao_Paulo]
```

7. É comum utilizar o encadeamento de chamadas com esses métodos.

src/org/j6toj8/localization/timezones/ZonedDateTime_Chaining.java

```
ZonedDateTime zonedDateTime = ZonedDateTime.of(2019, 5, 20, 9, 20, 4, 300,  
ZoneId.of("America/Sao_Paulo"));  
System.out.println(zonedDateTime);  
System.out.println(zonedDateTime.plusDays(1).plusHours(1).plusYears(1));
```

Saída no console

```
2019-05-20T09:20:04.000000300-03:00[America/Sao_Paulo]  
2020-05-21T10:20:04.000000300-03:00[America/Sao_Paulo]
```

8. Ao manipular um **ZonedDateTime**, será levado em conta o horário de verão daquele fuso horário.

src/org/j6toj8/localization/timezones/ZonedDateTime_DaylightSavings.java

```
ZonedDateTime zonedDateTime = ZonedDateTime.of(2018, 11, 3, 23, 30, 0, 0, ZoneId.  
of("America/Sao_Paulo"));  
System.out.println(zonedDateTime);  
System.out.println("+2 horas: " + zonedDateTime.plusHours(2));
```

Saída no console

```
2018-11-03T23:30-03:00[America/Sao_Paulo]  
+2 horas: 2018-11-04T02:30-02:00[America/Sao_Paulo]
```

Neste exemplo o fuso horário que era **-03:00** virou **-02:00**, pois esse foi o dia em que teve início o horário de verão no Brasil. Outro detalhe é que por conta do horário de verão, ao somar 2 horas às **23:30** resultou em **02:30** do dia seguinte. Se não houvesse horário de verão, o resultado seria **01:30**.

9. É possível recuperar todos os fusos horários disponíveis através da classe **ZoneId**.

src/org/j6toj8/localization/timezones/ZonedDateTime_Zones.java

```
Set<String> availableZoneIds = ZoneId.getAvailableZoneIds();  
for (String zoneId : availableZoneIds) {  
    System.out.println(zoneId);  
}
```

Saída no console (parcial)

```
Asia/Aden
America/Cuiaba
Etc/GMT+9
Etc/GMT+8
Africa/Nairobi
America/Marigot
Asia/Aqtau
Pacific/Kwajalein
America/El_Salvador
Asia/Pontianak
Africa/Cairo
Pacific/Pago_Pago
Africa/Mbabane
Asia/Kuching
Pacific/Honolulu
Pacific/Rarotonga
America/Guatemala
...
```

A lista do console irá apresentar todos os `ZoneId` disponíveis. O exemplo acima contempla apenas parte dos `ZoneId`.

Além disso, existem muitos `ZoneId` duplicados, pois representam o mesmo fuso horário, como por exemplo `America/Sao_Paulo` e `Brazil/East`.

Referências

- Working with Dates and Times
Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 234). Wiley. Edição do Kindle.
- [Introduction to the Java 8 Date/Time API.](#)
- [Trail: Date Time: Table of Contents.](#) The Java™ Tutorials.
- [What's the difference between Instant and LocalDateTime?](#)

Lambda

Interfaces Funcionais (Functional Interfaces)

Objetivo

Define and write functional interfaces and describe the interfaces of the `java.util.function` package.

-

Definir e escrever interfaces funcionais e descrever as interfaces do pacote `java.util.function`.

Interfaces funcionais são um novo tipo de Interface do Java. Nesta seção serão apresentados os conceitos e na seção de [Expressões Lambda](#) será possível ver como utilizá-las.

1. Interfaces Funcionais são aquelas que possuem apenas um método abstrato, chamado de "método funcional".
2. É recomendada a utilização da anotação `@FunctionalInterface`, mas não obrigatório.

`src/org/j6toj8/lambda/functionalinterfaces/FunctionalInterfaces_Basic.java`

```
@FunctionalInterface // a anotação não é obrigatória
interface Executavel { // interface funcional
    void execute(); // método funcional
}
```

A anotação `@FunctionalInterface` garante, em tempo de compilação, que esta interface é funcional. Também indica para outros desenvolvedores que ela foi criada com o intuito de ser utilizada em expressões lambda, e por isso não se deve criar outros métodos abstratos dentro dela.

3. Métodos adicionais que sejam `default` ou `static` não fazem com que a interface deixe de ser funcional.

```
@FunctionalInterface
interface Executavel { // interface funcional
    void execute(); // método funcional

    // métodos adicionais static são permitidos
    static void execute(Executavel... executaveis) {
        for (Executavel executavel : executaveis) {
            executavel.execute();
        }
    }

    // métodos adicionais default são permitidos
    default void executeDuasVezes() {
        execute();
        execute();
    }
}
```

Lembre-se que os métodos `static` em interfaces podem ser chamados diretamente, como `Executavel.execute(...)`. Dessa forma, não há interferência no fato da interface ser funcional.

Por outro lado, os métodos `default` só podem ser chamados caso você possua uma instância da interface, porém eles já possuem uma implementação padrão.

Em caso de dúvidas sobre `static` ou `default` em interfaces, volte na seção de "Métodos `static` e `default` em Interfaces".

4. Sobrescrever na interface um método público de `java.lang.Object` também não faz com que ela deixe de ser funcional.

```
@FunctionalInterface
interface Executavel { // interface funcional
    void execute(); // método funcional

    // sobrescrevendo métodos de Object
    @Override
    String toString();
    @Override
    boolean equals(Object obj);
    @Override
    int hashCode();
}
```

5. Uma interface que estende outra sem acrescentar métodos abstratos também pode ser funcional.

src/org/j6toj8/lambda/functionalinterfaces/FunctionalInterfaces_Extends.java

```
@FunctionalInterface  
interface Executavel { // interface funcional  
    void execute(); // método funcional  
}  
  
@FunctionalInterface  
interface Aplicacao extends Executavel {  
    // também é uma interface funcional  
}
```

6. Se uma interface estende outra que é funcional, porém acrescenta novos métodos abstratos, ela não é funcional.

src/org/j6toj8/lambda/functionalinterfaces/FunctionalInterfaces_ExtendsNewMethod.java

```
@FunctionalInterface  
interface Executavel { // interface funcional  
    void execute(); // método funcional  
}  
  
interface Aplicacao extends Executavel {  
    // NÃO é uma interface funcional, pois possui 2 métodos abstratos: execute  
    // (herdado) e inicie.  
    void inicie();  
}
```

7. Utilizar a anotação `@FunctionalInterface` em interfaces que possuem mais de um método abstrato causa um erro de compilação.

src/org/j6toj8/lambda/functionalinterfaces/FunctionalInterfaces_InterfaceCompilationError.java

```
@FunctionalInterface  
interface Executavel { // interface funcional  
    void execute(); // método funcional  
}  
  
@FunctionalInterface  
interface Aplicacao extends Executavel { // NÃO COMPILA!  
    // não pode ser anotada como funcional, pois possui 2 métodos abstratos  
    void inicie();  
}
```

8. Utilizar a anotação `@FunctionalInterface` em qualquer tipo que não seja uma interface causa um erro de compilação.

src/org/j6toj8/lambda/functionalinterfaces/FunctionalInterfaces_ClassCompilationError.java

```
@FunctionalInterface
interface Executavel { // interface funcional
    void execute(); // método funcional
}

@FunctionalInterface
class Piloto { // NÃO COMPILA!
    // apenas interfaces podem ser anotadas com @FunctionalInterface
}
```

9. Os métodos funcionais podem ter qualquer tipo de retorno.

src/org/j6toj8/lambda/functionalinterfaces/FunctionalInterfaces_ReturnType.java

```
@FunctionalInterface
interface Executavel { // interface funcional
    String execute(); // método funcional com retorno
}
```

10. Interfaces funcionais são feitas com o intuito de serem utilizadas em expressões lambda, mas o código também irá compilar normalmente caso uma classe a implemente.

src/org/j6toj8/lambda/functionalinterfaces/FunctionalInterfaces_Implement.java

```
@FunctionalInterface
interface Executavel { // interface funcional
    String execute(); // método funcional
}

class Pessoa implements Executavel {
    // COMPILA!
    // interfaces funcionais, como Corredor, não foram feitas para serem
    // implementadas dessa forma
    // porém é possível e o código compila normalmente
    @Override
    public String execute() {
        return "Executando";
    }
}
```

Esse é apenas um exemplo para você saber que essa implementação não gera erro de compilação, mas **não** utilize interfaces funcionais dessa forma. Na seção de Expressões Lambda você verá como as interfaces funcionais devem ser utilizadas na prática.

Interfaces Funcionais do pacote `java.util.function`

As interfaces descritas aqui estão disponíveis no pacote `java.util.function`. Nesta seção serão apresentadas apenas suas definições, pois há posteriormente uma seção específica para tratar dos exemplos de cada uma.

Existem outras interfaces nesse pacote além das listadas aqui, porém são apenas específicas para lidar com tipos primitivos, seguindo as mesmas definições.

- `Supplier<T>`: Representa um fornecedor de resultados.

Um `Supplier` literalmente apenas fornece dados ou resultados para alguém. Um gerador de números sequenciais, por exemplo, pode ser um `Supplier`.

- `Consumer<T>`: Representa uma operação que aceita uma única entrada e não possui retorno.
- `BiConsumer<T, U>`: Representa uma operação que aceita duas entradas e não possui retorno.

Os `Consumer` são praticamente o inverso dos `Supplier`, pois eles apenas recebem dados ou informações e os tratam de alguma forma.

- `Function<T, R>`: Representa uma função que aceita um argumento e produz um retorno.
- `BiFunction<T, U, R>`: Representa uma função que aceita dois argumentos e produz um retorno.

As `Function` são mais parecidas com as funções que já conhecemos. Elas recebem dados e produzem um retorno.

- `Predicate<T>`: Representa uma proposição (função de valor booleano) de um argumento.
- `BiPredicate<T, U>`: Representa uma proposição (função de valor booleano) de dois argumentos.

Os `Predicate` são parecidos com as `Function`, porém sempre retornam um resultado booleano, por isso são utilizados para "testes" de verdadeiro ou falso.

- `UnaryOperator<T>`: Representa uma operação em um único operador que produz um resultado do mesmo tipo dele.
- `BinaryOperator<T>`: Representa uma operação em dois operadores que produz um resultado do mesmo tipo deles.

Os `Operator` são especializações de `Function`, pois apesar de também sempre recebem e produzirem resultados, as entradas e saídas são sempre do mesmo tipo.

Referências

- Introducing Functional Programming

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 52). Wiley. Edição do Kindle.

- [Functional Interfaces in Java 8](#).
- [Annotation Type FunctionalInterface](#). Java Platform SE 8.
- [Package java.util.function](#). Java Platform SE 8.
- [Lambda Expressions](#). The Java™ Tutorials.

Expressões Lambda (Lambda Expression)

Objetivo

Describe a lambda expression; refactor the code that uses an anonymous inner class to use a lambda expression; describe type inference and target typing.

-

Descrever uma expressão lambda; refatorar código que usa classes anônimas internas para usar expressões lambda; descrever a inferência de tipos e tipos esperados.

Expressões Lambda são parecidas com classes anônimas, porém só possuem a implementação dos métodos. Por isso, são como "métodos anônimos", que podem ser passados via variáveis.

É essencial o entendimento de funções lambda, e das variações em sua sintaxe, para compreender as próximas seções de Interfaces Funcionais Pré-Construídas e de Referências a métodos.

A expressão lambda possui 3 partes:

1. Uma lista de parâmetros, separados por vírgula
 - Algumas vezes possui parênteses
 - Algumas vezes possui o tipo das variáveis
2. O "arrow token", que sempre é escrito como `->`
3. Um corpo, que pode ou não estar entre chaves
 - Pode possuir mais de uma linha
 - Algumas vezes possui um `return`
 - Algumas vezes possui ponto e vírgula

Exemplos de expressões lambda:

- `i → System.out.println(i)`
- `(Integer i) → System.out.println(i)`
- `(Integer i) → { System.out.println(i); }`

- `(Integer i) → { return i + 1; }`
- `(i, j, k) → { return i + j + k; }`
- `(i, j, k) → System.out.println(i + j + k)`
- `() → System.out.println("nada")`

1. Expressões lambda podem ser entendidas como uma forma diferente de declarar classes anônimas.

src/org/j6toj8/lambda/lambdaexpression/LambdaExpression_AnonymousClass.java

```
// com classe anônima
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Executando.");
    }
}).run();

// com expressão lambda
new Thread(() -> System.out.println("Executando.")).run();
```

Veja que no exemplo acima é instanciada uma `Thread` com uma instância anônima de `Runnable`. Na segunda parte, é feito a mesma coisa de forma muito mais simples utilizando uma expressão lambda.

2. Expressões lambda são sempre utilizadas para criar instâncias de interfaces funcionais, ou seja, interfaces que possuem apenas um único método abstrato.

```
@FunctionalInterface
interface Executavel { // interface funcional
    String execute(); // método funcional
}

private static void executeEApresenteMensagem(Executavel executavel) {
    String mensagem = executavel.execute();
    System.out.println(mensagem);
}

public static void main(String[] args) {

    // com classe anônima
    executeEApresenteMensagem(new Executavel() {
        @Override
        public String execute() {
            return "executei com classe anônima";
        }
    });

    // com expressão lambda
    executeEApresenteMensagem(() -> { return "executei com expressão lambda"; });
}
```

Saída no console

```
executei com classe anônima
executei com expressão lambda
```

Veja que no exemplo acima o mesmo método `executeEApresenteMensagem` é invocado duas vezes. Na primeira vez é passada uma nova classe anônima. Na segunda vez é passado uma expressão lambda.

Veja também que seria impossível criar uma expressão lambda caso a interface não fosse funcional, ou seja, tivesse mais de um método abstrato. O compilador não saberia identificar que o método `execute`, da interface `Executavel`, está sendo implementado dentro da expressão lambda.

3. Existem muitos métodos já disponíveis no Java 8 que se beneficiam da sintaxe de expressões lambda, como o `forEach` de listas.

```
List<Integer> lista = Arrays.asList(1, 2, 3, 4, 5);
lista.forEach((numero) -> { System.out.println(numero); });
```

Saída no console

```
1  
2  
3  
4  
5
```

Veja que o novo método `forEach` executa a expressão lambda passada como parâmetro para cada item da lista, imprimindo todos no console. A expressão lambda recebe como parâmetro um número, que é o número da lista.

Neste caso, a interface funcional que está sendo implementada pela expressão lambda é chamada `Consumer`. Ela será explicada em detalhes em uma seção posterior, juntamente com outras interfaces funcionais padrões do Java 8. Nesta seção é importante apenas entender o que são as expressões lambda e como é sua sintaxe.

4. Declarações de expressões lambda podem ser completas ou simplificadas.

src/org/j6toj8/lambda/lambdaexpression/LambdaExpression_SimpleComplete.java

```
// expressão lambda completa  
UnaryOperator<Double> elevarAoQuadrado1 = (Double x) -> { return Math.pow(x, 2); };  
// expressão lambda simplificada  
UnaryOperator<Double> elevarAoQuadrado2 = (x) -> Math.pow(x, 2);
```

As duas funções lambda acima são idênticas, porém uma é mais explícita do que a outra.

5. Os parênteses só podem ser omitidos caso não haja a declaração do **tipo**, e haja apenas **um único** argumento.
6. Expressões lambda que NÃO possuem argumentos também precisam dos parênteses.

src/org/j6toj8/lambda/lambdaexpression/LambdaExpression_Parenthesis.java

```
// NÃO COMPILA - parênteses são obrigatórios ao declarar o tipo da variável da expressão lambda
UnaryOperator<Double> elevarAoQuadrado = Double x -> Math.pow(x, 2);

// NÃO COMPILA - é obrigatório utilizar parênteses quando há mais de uma variável na expressão lambda
BinaryOperator<Double> elevarAoX = y, x -> Math.pow(y, x);

// NÃO COMPILA - é obrigatório utilizar parênteses quando não há variáveis
Supplier<Double> elevar2aoQuadrado = -> Math.pow(2, 2);

// COMPILA - parênteses vazios quando não há variáveis
Supplier<Double> elevar2aoQuadrado = () -> Math.pow(2, 2);

// COMPILA - sem parênteses quando há apenas uma variável e não escrevemos seu tipo
UnaryOperator<Double> elevarAoQuadrado2 = x -> Math.pow(x, 2);
```

7. É obrigatória a utilização de chaves, ponto e vírgula e **return** (caso a função retorne algum valor) em expressões lambda com mais de uma linha.

src/org/j6toj8/lambda/lambdaexpression/LambdaExpression_Block.java

```
UnaryOperator<Double> elevarAoQuadrado = (Double x) -> {
    double pow = Math.pow(x, 2);
    return pow;
};

Consumer<Double> imprime = (Double x) -> {
    double pow = Math.pow(x, 2);
    System.out.println(pow);
};
```

8. Ao tornar explícito o tipo de um dos argumentos, é obrigatório informar de todos.

src/org/j6toj8/lambda/lambdaexpression/LambdaExpression_VarType.java

```
// NÃO COMPILA - caso o tipo de um dos parâmetros for informado, é necessário informar de todos eles
UnaryOperator<Double> elevarAoX = (Double y, x) -> Math.pow(y, x);

// COMPILA - todos os parâmetros com tipos informados
UnaryOperator<Double> elevarAoX2 = (Double y, Double x) -> Math.pow(y, x);
```

9. Não é permitido declarar variáveis com o mesmo nome dentro da expressão lambda.

src/org/j6toj8/lambda/lambdaexpression/LambdaExpression_Shadowing.java

```
public static void main(String[] args) {
    Double x = 2.0; // variável 'x' no método

    // NÃO COMPILA - a variável com nome 'x' já existe e não pode ser declarada nas
    // variáveis da expressão lambda
    BinaryOperator<Double> elevarAoX = (Double y, Double x) -> Math.pow(y, x);

    // NÃO COMPILA - a variável com nome 'x' já existe e não pode ser declarada no
    // corpo da expressão lambda
    UnaryOperator<Double> elevarAoQuadrado = (Double y) -> {
        Double x = 2.0;
        return Math.pow(y, x);
    };
}
```

10. É permitido acessar variáveis externas dentro da expressão lambda, mas somente variáveis finais ou variáveis que não são alteradas.

src/org/j6toj8/lambda/lambdaexpression/LambdaExpression_AccessExternalVar.java

```
public static void main(String[] args) {
    final Double x1 = 2.0;
    Double x2 = 2.0;
    Double x3 = 2.0;

    // COMPILA - variável externa 'x1' é final e pode ser utilizada na expressão
    // lambda
    UnaryOperator<Double> elevarAoX1 = (Double y) -> Math.pow(y, x1);

    // COMPILA - variável externa 'x2' não é final, mas nunca é alterada, então pode
    // ser utilizada dentro da expressão lambda
    UnaryOperator<Double> elevarAoX2 = (Double y) -> Math.pow(y, x2);

    // NÃO COMPILA - variável externa 'x3' é alterada dentro desse método, então não
    // pode ser utilizada dentro da expressão lambda
    UnaryOperator<Double> elevarAoX3 = (Double y) -> Math.pow(y, x3);

    x3 = 3.0; // alteração da variável x3 não permite que ela seja utilizada em
    // expressões lambda
}
```

Perceba que o compilador identifica que a variável **x3** é alterada no final do método, e por isso, não permite que ela seja utilizada na expressão lambda.

11. Em situações de ambiguidade, o compilador tenta descobrir o tipo da expressão lambda utilizando o contexto.

```
@FunctionalInterface
interface Executavel {
    void execute(); // método funcional sem argumentos
}

@FunctionalInterface
interface Application {
    String run(); // método funcional também sem argumentos
}

static class Executor {
    // esse método pode receber uma expressão lambda sem argumentos
    void rode(Executavel executavel) {
        executavel.execute();
    }

    // esse método também pode receber uma expressão lambda sem argumentos
    void rode(Application application) {
        application.run();
    }
}

public static void main(String[] args) {
    Executor executor = new Executor();
    executor.rode(() -> { return "executando"; }); // irá chamar o execute que recebe
    // uma Application
    executor.rode(() -> { System.out.println("executando"); }); // irá chamar o
    // execute que recebe um Executavel
}
```

No exemplo anterior, apenas o método `run` da interface funcional `Application` retorna uma `String`, enquanto o método `execute` da interface funcional `Executavel` não retorna nada (`void`). Isso é diferença suficiente para o compilador saber qual método chamar cada vez que `rode` é invocado.

Na primeira chamada ao método `rode`, como a expressão lambda passada retorna uma `String`, o compilador entende que essa expressão lambda é uma implementação da interface `Application`, pois o método `run` também retorna uma `String`.

Na segunda chamada ao método `rode`, como a expressão lambda não retorna nada (apenas imprime a `String "executando"`), o compilador entende que essa expressão lambda é uma implementação da interface `Executavel`, pois o método `execute` também não retorna nada.

12. Se não for possível descobrir o tipo da expressão lambda, ocorre erro de compilação.

```
@FunctionalInterface
interface Corredor {
    void corra();
}

@FunctionalInterface
interface Piloto {
    void corra();
}

static class Executor {
    void execute(Corredor corredor) {
        corredor.corra();
    }

    String execute(Piloto piloto) {
        piloto.corra();
        return "correndo";
    }
}

public static void main(String[] args) {
    Executor executor = new Executor();
    // NÃO COMPILA - não é possível determinar o tipo da expressão lambda abaixo
    executor.execute(() -> System.out.println("execute"));
}
```

No exemplo anterior, como as duas interfaces funcionais possuem métodos com retorno `void`, o compilador não sabe qual das duas está sendo instanciada na expressão lambda, e ocorre erro de compilação. A expressão lambda, nesse exemplo, poderia ser tanto do tipo `Piloto` quanto `Corredor`, e não há como o compilador descobrir qual o desenvolvedor de fato quis utilizar.

Referências

- Implementing Functional Interfaces with Lambdas

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 55). Wiley. Edição do Kindle.

- Using Variables in Lambdas

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 172). Wiley. Edição do Kindle.

- [Lambda Expressions and Functional Interfaces: Tips and Best Practices](#).
- [Lambda Expressions](#). The Java™ Tutorials.

Interfaces Funcionais Pré-Construídas (Built-in Interfaces)

Objetivo

Develop code that uses the built-in interfaces included in the `java.util.function` package, such as `Function`, `Consumer`, `Supplier`, `UnaryOperator`, `Predicate`, and `Optional` APIs, including the primitive and binary variations of the interfaces

- Desenvolver código que usa as interfaces embutidas no pacote `java.util.function`, como `Function`, `Consumer`, `Supplier`, `UnaryOperator`, `Predicate` e a API de `Optional`, incluindo as variações de tipos primitivos e binários das interfaces

Interfaces Funcionais

O Java 8 possui algumas Interfaces Funcionais já criadas. Elas provavelmente serão suficientes para a maioria dos cenários onde é usual utilizar expressões lambda, de tal forma que não deve ser muito comum você precisar criar uma nova.

É fundamental entender esses exemplos para dominar a utilização de expressões lambda, e para entender a próxima seção sobre referências a métodos.

Supplier

1. `Supplier` é uma interface funcional que não recebe nenhum parâmetro de entrada, mas retorna um valor. Sua definição na JDK é a seguinte:

```
java.util.function.Supplier<T>
```

```
@FunctionalInterface  
public interface Supplier<T> {  
    T get();  
}
```

2. Uma implementação possível para um `Supplier` é um gerador da data atual:

```
src/org/j6toj8/lambda/builtininterfaces/BuiltInInterfaces_SupplierExample.java
```

```
public static void main(String[] args) {  
    Supplier<LocalDate> supplier = () -> LocalDate.now();  
    System.out.println(supplier.get()); // imprimirá a data atual  
}
```

Saída no console

```
2019-07-08
```

A saída no console irá imprimir a data atual em que este código foi escrito.

Perceba que a expressão lambda está simplificada, sem chaves {} ou return. Caso tenha dúvidas com relação a isso, consulte novamente a seção sobre expressões lambda.

3. Um **Supplier** pode ser utilizado para fornecer uma função custosa em termos de processamento, para que seja chamada apenas se for necessário:

src/org/j6toj8/lambda/builtininterfaces/BuiltInInterfaces_SupplierUseCase.java

```
public static String valideIdade(int idade, Supplier<String> supplier) {  
    if (idade < 18) {  
        return "Menor de idade!";  
    } else {  
        return "Maior de idade! Validação realizada às " + supplier.get();  
    }  
}  
  
public static void main(String[] args) {  
    Supplier<String> supplier = () -> LocalDate.now().atTime(LocalTime.now()).  
format(DateTimeFormatter.ISO_DATE_TIME);  
    System.out.println(valideIdade(17, supplier));  
    System.out.println(valideIdade(18, supplier));  
}
```

Saída no console

```
Menor de idade!  
Maior de idade! Validação realizada às 2019-07-09T00:21:35.488
```

Perceba que neste caso o supplier só precisou ser chamado na segunda vez, evitando uma execução desnecessária da expressão lambda.

4. Existem interfaces **Supplier** para lidar tipos primitivos: **BooleanSupplier**, **IntSupplier**, **LongSupplier** e **DoubleSupplier**.

src/org/j6toj8/lambda/builtininterfaces/BuiltInInterfaces_SupplierPrimitive.java

```
public static void main(String[] args) {  
    IntSupplier randomIntSupplier = () -> new Random().nextInt();  
    System.out.println(randomIntSupplier.getAsInt()); // getAsInt retorna um int  
    primitivo  
}
```

O resultado na console será imprimir o **int** primitivo gerado aleatoriamente.

Consumer e BiConsumer

1. **Consumer** é uma interface funcional que recebe um parâmetro de entrada, e não retorna

nenhum valor. Sua definição na JDK é a seguinte:

```
java.util.function.Supplier<T>
```

```
@FunctionalInterface  
public interface Consumer<T> {  
    void accept(T t);  
}
```

2. **BiConsumer** é uma interface funcional que recebe dois parâmetros de entrada, e não retorna nenhum valor. Sua definição na JDK é a seguinte:

```
java.util.function.Consumer<T>
```

```
@FunctionalInterface  
public interface BiConsumer<T, U> {  
    void accept(T t, U u);  
}
```

3. Implementações possíveis para **Consumer** ou **BiConsumer** são funções que imprimem informações no console:

src/org/j6toj8/lambda/builtininterfaces/BuiltInInterfaces_ConsumerExample.java

```
public static void main(String[] args) {  
    Consumer<Object> impressor = x -> System.out.println(x);  
    impressor.accept(LocalDate.now()); // imprimirá a data atual  
  
    BiConsumer<Object, Object> impressor2 = (x, y) -> { System.out.println(x);  
        System.out.println(y); };  
    impressor2.accept(LocalDate.now(), LocalTime.now()); // imprimirá a data atual e  
    depois a hora atual  
}
```

Saída no console

```
2019-07-08  
2019-07-08  
22:37:39.229
```

4. Existem interfaces **Consumer** para lidar tipos primitivos: **DoubleConsumer**, **IntConsumer**, **LongConsumer**, **ObjDoubleConsumer**, **ObjIntConsumer** e **ObjLongConsumer**.

```
public static void main(String[] args) {
    IntConsumer impressor = x -> System.out.println(x);
    impressor.accept(5); // imprimirá '5'

    ObjIntConsumer<Object> impressor2 = (x, y) -> { System.out.println(x); System.
    out.println(y); };
    impressor2.accept(LocalDate.now(), 5); // imprimirá a data atual e depois '5'
}
```

Predicate e BiPredicate

1. **Predicate** é uma interface funcional que recebe um parâmetro de entrada e retorna um valor booleano. Sua definição na JDK é a seguinte:

java.util.function.Predicate<T>

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

2. **BiPredicate** é uma interface funcional que recebe dois parâmetros de entrada e retorna um valor booleano. Sua definição na JDK é a seguinte:

java.util.function.BiPredicate<T>

```
@FunctionalInterface
public interface BiPredicate<T, U> {
    boolean test(T t, U u);
}
```

3. Implementações possíveis para **Predicate** ou **BiPredicate** são funções que verificam se o valor de entrada é igual ao valor sorteado:

src/org/j6toj8/lambda/builtininterfaces/BuiltInInterfaces_PredicateExample.java

```
public static void main(String[] args) {
    Predicate<Integer> dado = x -> x.equals(new Random().nextInt(7));
    System.out.println(dado.test(1)); // testa se o número gerado é igual a 1

    BiPredicate<Integer, Integer> dadoDuplo = (x, y) -> x.equals(new Random().
nextInt(7)) || y.equals(new Random().nextInt(7));
    System.out.println(dadoDuplo.test(1, 2)); // testa se o primeiro número gerado é
igual a 1
                                            // ou se o segundo número gerado é
igual a 2
}
```

A saída no console é aleatória, pois depende do valor sorteado. Um valor possível seria `false` e `true`.

4. Existem interfaces `Predicate` para lidar tipos primitivos: `DoublePredicate`, `IntPredicate` e `LongPredicate`.

src/org/j6toj8/lambda/builtininterfaces/BuiltInInterfaces_PredicatePrimitive.java

```
public static void main(String[] args) {
    IntPredicate dado = x -> x == new Random().nextInt(7);
    System.out.println(dado.test(1)); // testa se o número gerado é igual a 1
}
```

Function e BiFunction

1. `Function` é uma interface funcional que recebe um parâmetro de entrada e retorna um valor. Sua definição na JDK é a seguinte:

java.util.function.Function<T, R>

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

2. `BiFunction` é uma interface funcional que recebe dois parâmetros de entrada e retorna um valor. Sua definição na JDK é a seguinte:

java.util.function.BiFunction<T>

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

3. Implementações possíveis para `Function` ou `BiFunction` são funções que multiplicam os valores fornecidos:

`src/org/j6toj8/lambda/builtininterfaces/BuiltInInterfaces_FunctionExample.java`

```
public static void main(String[] args) {
    Function<Integer, Double> duplica = x -> x * 2.5;
    System.out.println(duplica.apply(3)); // multiplica 3 * 2.5

    BiFunction<Integer, Integer, Double> multiplicaEDuplica = (x, y) -> x * y * 2.5;
    System.out.println(multiplicaEDuplica.apply(3, 4)); // multiplica 3 * 4 * 2.5
}
```

Saída no console

```
7.5
30.0
```

4. Existem várias interfaces `Function` para lidar tipos primitivos: `DoubleFunction`, `DoubleToIntFunction`, `DoubleToLongFunction`, `IntFunction`, `IntToDoubleFunction`, `IntToLongFunction`, `LongFunction`, `LongToDoubleFunction`, `LongToIntFunction`, `ToDoubleBiFunction`, `ToDoubleFunction`, `ToIntBiFunction`, `ToIntFunction`, `ToLongBiFunction`, `ToLongFunction`.

`src/org/j6toj8/lambda/builtininterfaces/BuiltInInterfaces_FunctionPrimitive.java`

```
public static void main(String[] args) {
    IntToDoubleFunction duplica = x -> x * 2.5;
    System.out.println(duplica.applyAsDouble(3)); // multiplica 3 * 2.5

    ToDoubleBiFunction<Integer, Integer> multiplicaEDuplica = (x, y) -> x * y * 2.5;
    System.out.println(multiplicaEDuplica.applyAsDouble(3, 4)); // multiplica 3 * 4 *
2.5
}
```

Saída no console

```
7.5
30.0
```

UnaryOperator e BinaryOperator

1. `UnaryOperator` é uma interface funcional que recebe um parâmetro de entrada e retorna um valor do mesmo tipo da entrada. Sua definição na JDK é a seguinte:

```
java.util.function.Function<T, R>
```

```
@FunctionalInterface  
public interface UnaryOperator<T> extends Function<T, T> {  
}
```

Perceba que não existe método abstrato declarado, pois ela apenas estende a interface `Function` já existente.

2. `BinaryOperator` é uma interface funcional que recebe dois parâmetros de entrada do mesmo tipo, e retorna um valor do mesmo tipo das entradas. Sua definição na JDK é a seguinte:

```
java.util.function.BiFunction<T>
```

```
@FunctionalInterface  
public interface BinaryOperator<T> extends BiFunction<T, T, T> {  
}
```

Perceba que não existe método abstrato declarado, pois ela apenas estende a interface `BiFunction` já existente.

3. Implementações possíveis para `UnaryOperator` ou `BinaryOperator` são funções que soma um número fixo ou soma um número ao outro:

src/org/j6toj8/lambda/builtininterfaces/BuiltInInterfaces_OperatorExample.java

```
public static void main(String[] args) {  
    UnaryOperator<Integer> somaDois = x -> x + 2;  
    System.out.println(somaDois.apply(7)); // soma 7 + 2  
  
    BinaryOperator<Integer> somaNumeros = (x, y) -> x + y;  
    System.out.println(somaNumeros.apply(1, 5)); // soma 1 + 5  
}
```

Saída no console

```
9  
6
```

4. Existem interfaces `Operator` para lidar tipos primitivos: `DoubleBinaryOperator`, `DoubleUnaryOperator`, `IntBinaryOperator`, `IntUnaryOperator`, `LongBinaryOperator`, `LongUnaryOperator`.

```
public static void main(String[] args) {  
    IntUnaryOperator somaDois = x -> x + 2;  
    System.out.println(somaDois.applyAsInt(7)); // soma 7 + 2  
  
    IntBinaryOperator somaNumeros = (x, y) -> x + y;  
    System.out.println(somaNumeros.applyAsInt(1, 5)); // soma 1 + 5  
}
```

Saída no console

```
9  
6
```

Optional

O Java 8 possui um tipo específico para representar valores que podem não ter sido informados, que é a classe `Optional`. A partir do Java 8, ela geralmente é uma opção melhor do que retornar ou armazenar `null`, pois seus métodos auxiliam em várias situações.

1. É possível criar uma instância de `Optional` com valor através do método `of`.
2. É possível criar uma instância de `Optional` sem valor através do método `empty`.
3. É possível checar se uma instância de `Optional` possui um valor através do método `isPresent`.
4. É possível recuperar o valor de uma instância de `Optional` através do método `get`.

```
// Exemplo de método sem Optional
private static String recuperarNomeMes(int mes) {
    if (mes == 1) {
        return "Janeiro";
    } else {
        return null;
    }
}

// Exemplo de método retornando Optional
private static Optional<String> recuperarNomeMesOptional(int mes) {
    if (mes == 1) {
        return Optional.of("Janeiro"); // cria Optional com valor
    } else {
        return Optional.empty(); // cria Optional vazio
    }
}

public static void main(String[] args) {
    String nomeMes1 = recuperarNomeMes(1);
    if (nomeMes1 != null) { // valida se o valor existe através da tradicional
        comparação '!= null'
        System.out.println(nomeMes1);
    }

    Optional<String> nomeMes2 = recuperarNomeMesOptional(1);
    if (nomeMes2.isPresent()) { // valida se o Optional possui um valor preenchido
        System.out.println(nomeMes2.get()); // recupera o valor dentro do Optional
    }
}
```

5. Não é possível chamar o método `of` passando `null` como argumento. Para isso existe o método `ofNullable`.

```
public static void main(String[] args) {
    // Exemplo tentando utilizar .of e passando 'null' como argumento
    try {
        Optional.of(null); // Lança NullPointerException nesta linha
    } catch (Exception e) {
        e.printStackTrace();
    }

    // Exemplo utilizando o método correto: .ofNullable
    Optional<String> nullable = Optional.ofNullable(null); // Cria um Optional vazio
    System.out.println(nullable.isPresent()); // Imprime 'false' pois não possui valor
}
```

Saída no console

```
java.lang.NullPointerException
    at java.util.Objects.requireNonNull(Objects.java:203)
    at java.util.Optional.<init>(Optional.java:96)
    at java.util.Optional.of(Optional.java:108)
    at
org.j6toj8.lambda.builtininterfaces.BuiltInInterfaces_OptionalNullable.main(BuiltInInterfaces_OptionalNullable.java:11)
false
```

6. Com o método **ifPresent** é possível executar uma expressão lambda apenas se o **Optional** tiver valor.

```
public static void main(String[] args) {
    Optional<String> optionalVazio = Optional.empty();
    Optional<String> optionalComValor = Optional.of("valor");

    // A linha abaixo não irá imprimir nada, pois o optional está vazio
    optionalVazio.ifPresent(valor -> System.out.println("Vazio: " + valor));
    // A linha abaixo irá imprimir, pois o optional possui valor
    optionalComValor.ifPresent(valor -> System.out.println("Com Valor: " + valor));
}
```

Saída no console

```
Com Valor: valor
```

7. É possível recuperar um valor padrão caso o **Optional** esteja vazio. O método **orElse** retorna um

valor diretamente, e o `orElseGet` retorna através de uma expressão lambda.

`src/org/j6toj8/lambda/builtininterfaces/BuiltInInterfaces_OptionalOrElse.java`

```
public static void main(String[] args) {
    // Exemplo com Optional vazio
    Optional<String> optionalVazio = Optional.empty();

    // as duas variáveis abaixo terão a String "valor padrao", pois o optional está
    // vazio
    String orElse = optionalVazio.orElse("valor padrao"); // obtém a String
    // diretamente
    String orElseGet = optionalVazio.orElseGet(() -> { return "valor padrao"; }); // obtém a String através da expressão lambda

    System.out.println(orElse);
    System.out.println(orElseGet);

    // Exemplo com Optional com valor
    Optional<String> optionalComValor = Optional.of("valor");

    // as duas variáveis abaixo irão utilizar o valor presente no optional, pois ele
    // já está preenchido
    String orElse2 = optionalComValor.orElse("valor padrao");
    String orElseGet2 = optionalComValor.orElseGet(() -> { return "valor padrao"; });

    System.out.println(orElse2);
    System.out.println(orElseGet2);
}
```

Saída no console

```
valor padrao
valor padrao
valor
valor
```



Observe que esse é um ótimo caso para lembrar de um benefício das expressões lambda. Na utilização de `orElseGet` a expressão lambda só é executada caso o `Optional` esteja vazio. No caso do exemplo, como é apenas o retorno de uma `String`, não faz diferença. Porém, se fosse uma operação mais pesada, você só iria de fato executá-la se o `Optional` estivesse vazio. Caso houvesse valor, a expressão lambda nem seria executada, evitando o custo de processamento.

8. Também é possível lançar uma exceção caso um valor não esteja presente no `Optional` utilizando o método `orElseThrow`.

src/org/j6toj8/lambda/builtininterfaces/BuiltInInterfaces_OptionalOrElseThrow.java

```
public static void main(String[] args) {
    Optional<String> optionalVazio = Optional.empty();
    Optional<String> optionalComValor = Optional.of("valor");

    // Nesse caso será impresso o valor presente em Optional, pois ele está
    // preenchido
    String orElseThrow1 = optionalComValor.orElseThrow(() -> new RuntimeException());
    System.out.println(orElseThrow1);

    // Nesse caso será lançada exceção, pois o Optional não está preenchido
    String orElseThrow2 = optionalVazio.orElseThrow(() -> new RuntimeException());
}
```

Saída no console

```
valor
Exception in thread "main" java.lang.RuntimeException
    at
org.j6toj8.lambda.builtininterfaces.BuiltInInterfaces_OptionalOrElseThrow.lambda$1(
BuiltInInterfaces_OptionalOrElseThrow.java:17)
    at java.util.Optional.orElseThrow(Optional.java:290)
    at
org.j6toj8.lambda.builtininterfaces.BuiltInInterfaces_OptionalOrElseThrow.main(Buil
tInInterfaces_OptionalOrElseThrow.java:17)
```

9. Será lançada uma exceção ao chamar o método `get` em um `Optional` vazio.

src/org/j6toj8/lambda/builtininterfaces/BuiltInInterfaces_OptionalGetEmpty.java

```
Optional<String> optionalComValor = Optional.of("valor");
System.out.println(optionalComValor.get()); // recupera o valor corretamente

Optional<String> optionalVazio = Optional.empty();
System.out.println(optionalVazio.get()); // lança exceção
```

Saída no console

```
valor
Exception in thread "main" java.util.NoSuchElementException: No value present
    at java.util.Optional.get(Optional.java:135)
    at
org.j6toj8.lambda.builtininterfaces.BuiltInInterfaces_OptionalGetEmpty.main(BuiltIn
Interfaces_OptionalGetEmpty.java:13)
```

10. Existem algumas classes para lidar com valor opcionais de variáveis primitivas, já que elas não podem ser utilizadas com `generics: OptionalInt, OptionalDouble, OptionalLong`.

```
OptionalInt optionalComValor = OptionalInt.of(5);
OptionalInt optionalVazio = OptionalInt.empty();

if (optionalComValor.isPresent()) {
    System.out.println(optionalComValor.getAsInt());
}
if (optionalVazio.isPresent()) {
    System.out.println(optionalVazio.getAsInt());
}
```

Saída no console

```
5
```

Referências

- Working with Built-In Functional Interfaces
Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 173). Wiley. Edição do Kindle.
- [Functional Interfaces in Java 8](#).
- [Guide To Java 8 Optional](#).
- [Package java.util.function](#). Java Platform SE 8.
- [Class Optional<T>](#). Java Platform SE 8.
- [Lambda Built-in Functional Interfaces](#).

Referências a Métodos (Method Reference)

Objetivo

Develop code that uses a method reference, including refactoring a lambda expression to a method reference.

- Desenvolver código que utiliza uma referência a método, incluindo a refatoração de uma expressão lambda para uma referência a método.

A sintaxe de referência a um método é uma novidade do Java 8. Com ela é possível fazer referência a métodos específicos, em quatro ocasiões diferentes:

- Referências a métodos estáticos → `String::valueOf`
- Referências a métodos de um objeto → `instanciaDeString::isEmpty`

- Referências a métodos de um tipo de objeto (de uma classe, interface, etc) → `String::isEmpty`
- Referências a construtores → `String::new`

É essencial lembrar das Interfaces Funcionais, das variações de sintaxe de Expressões Lambda e das definições de Interfaces Funcionais Pré-Construídas. Caso julgue necessário, reveja as seções deste capítulo.

É possível pensar nas referências a métodos como uma outra forma de escrever uma expressão lambda, caso a única coisa que sua expressão lambda faça seja chamar um outro método.

A seguir serão apresentadas as ocasiões em que são utilizadas as referências a métodos.

1. Chamadas a métodos estáticos em expressões lambda podem virar uma referência ao método.

`src/org/j6toj8/lambda/methodreference/MethodReference_Static.java`

```
// representação com expressão lambda
Function<Integer, String> converteIntEmStr1 = x -> String.valueOf(x);
// representação com referência ao método
Function<Integer, String> converteIntEmStr2 = String::valueOf;

System.out.println(converteIntEmStr1.apply(5));
System.out.println(converteIntEmStr2.apply(5));
```

Saída no console

```
5
5
```

Nesse caso a única coisa que a expressão lambda faz é receber um argumento `x` e repassar para o método `valueOf` de `String`. Para simplificar isso, o Java 8 permite que você escreva essa mesma função lambda como foi apresentado na linha seguinte: `String::valueOf`.

Só é possível representar a primeira expressão lambda na forma de um *method reference* porque:

- A implementação de `String.valueOf` satisfaz a interface funcional `Function` (recebe um argumento e retorna um valor).
- O argumento de entrada da expressão lambda `x` é exatamente o mesmo passado para o método `String.valueOf(x)`.
- A expressão lambda é simples: somente possui uma chamada a um método.

2. Chamadas a métodos de uma instância específica também podem ser representados como uma referência a um método.

```
static class Conversor {
    public String converte(Integer x, Integer y) {
        return String.valueOf(x) + " - " + String.valueOf(y);
    }
}

public static void main(String[] args) {
    Conversor conversor = new Conversor(); // instância da classe Conversor

    // representação com expressão lambda
    BiFunction<Integer, Integer, String> converte1 = (x, y) -> conversor.converte(x, y);
    // representação com referência ao método da instância
    BiFunction<Integer, Integer, String> converte2 = conversor::converte;

    // os resultados serão iguais
    System.out.println(converte1.apply(5, 8));
    System.out.println(converte2.apply(5, 8));
}
```

Saída no console

```
5 - 8
5 - 8
```

Só é possível representar a primeira expressão lambda na forma de um *method reference* porque:

- A implementação de `Conversor.converte(Integer, Integer)` satisfaz a interface funcional `BiFunction` (recebe dois argumentos e retorna um valor).
 - Os argumentos de entrada da expressão lambda `x` e `y` são exatamente os mesmos passados para o método `Conversor.converte(Integer, Integer)`.
 - A expressão lambda é simples: somente possui uma chamada a um método.
3. Chamadas a métodos de uma classe, sem especificar a instância específica, também podem ser representados como uma referência a um método.

```
// representação com expressão lambda
Function<Integer, Double> intParaDouble1 = x -> x.doubleValue();
// representação com referência ao método doubleValue
Function<Integer, Double> intParaDouble2 = Integer::doubleValue;

// os resultados serão iguais
System.out.println(intParaDouble1.apply(8));
System.out.println(intParaDouble2.apply(8));
```

Saída no console

```
8.0
8.0
```

Nesse exemplo, a referência está sendo feita ao método `doubleValue` do tipo `Integer`. Só é possível representar a primeira expressão lambda na forma de um *method reference* porque:

- Nossa expressão lambda satisfaz a interface funcional `Function` (recebe um argumento `x` e retorna um valor `double`).
 - A expressão lambda recebe um argumento `x` do tipo `Integer`, que possui o método `doubleValue` que não recebe parâmetros.
 - A expressão lambda é simples: somente possui uma chamada a um método.
4. Também é possível utilizar a referência ao método de um tipo, como no exemplo anterior, mesmo que o método receba parâmetros.

```
// representação com expressão lambda
BiFunction<Integer, Integer, Integer> comparador1 = (x, y) -> x.compareTo(y);
// representação com referência ao método compareTo do tipo Integer (que recebe um
// parâmetro)
BiFunction<Integer, Integer, Integer> comparador2 = Integer::compareTo;

// os resultados serão iguais
System.out.println(comparador1.apply(1, 2));
System.out.println(comparador2.apply(1, 2));
```

Saída no console

```
-1
-1
```

Nesse exemplo o compilador "descobre" ainda mais coisas que nos exemplos anteriores. Ao escrever apenas a referência ao método, o compilador entende que a variável `x`, que vem primeiro, será a instância de `Integer` onde o método `compareTo` será chamado. E que `y` é a

instância de `Integer` que será passada como argumento para o método `compareTo`.

5. Chamadas a um construtor também podem ser representadas como uma referência a um método.

`src/org/j6toj8/lambda/methodreference/MethodReference_Constructor.java`

```
// representação com expressão lambda
Function<String, Integer> stringParaInteger1 = s -> new Integer(s);
// representação com referência ao construtor
Function<String, Integer> stringParaInteger2 = Integer::new;

// os resultados serão iguais
System.out.println(stringParaInteger1.apply("1"));
System.out.println(stringParaInteger2.apply("1"));
```

Saída no console

```
1
1
```

Esse exemplo é muito parecido com o anterior, com a única diferença sendo que o método referenciado é um construtor. Perceba que o construtor também recebe um parâmetro e, assim como no exemplo anterior, o compilador entende que o argumento da função lambda deve ser passado para o construtor que foi chamado.

6. Expressões lambda complexas não podem ser convertidas em referência a método, como a expressão abaixo:

`src/org/j6toj8/lambda/methodreference/MethodReference_Complex.java`

```
Function<String, Integer> stringParaInteger1 = s -> new Integer(s + "2");
System.out.println(stringParaInteger1.apply("1"));
```

Como nesse caso temos uma outra `String + "2"` sendo acrescentada no construtor, não há como representar isso com uma simples referência ao construtor.

7. É possível utilizar *method reference* também com suas próprias classes. Veja no exemplo a seguir os tipos criados pelo nosso código e as expressões lambda equivalentes com e sem referência a métodos.

```
static class SuperHeroi {  
    private Pessoa pessoa;  
  
    public SuperHeroi(Pessoa pessoa) {  
        this.pessoa = pessoa;  
    }  
  
    public static SuperHeroi crieSuperHeroi(Pessoa pessoa) {  
        return new SuperHeroi(pessoa);  
    }  
  
}  
  
static class Pessoa {  
  
    public SuperHeroi vireSuperHeroi() {  
        return new SuperHeroi(this);  
    }  
  
}  
  
public static void main(String[] args) {  
    // expressões lambda equivalentes utilizando chamada a método estático  
    Function<Pessoa, SuperHeroi> transformaEmHeroiStatic1 = p ->  
    SuperHeroi.crieSuperHeroi(p);  
    Function<Pessoa, SuperHeroi> transformaEmHeroiStatic2 =  
    SuperHeroi::crieSuperHeroi;  
  
    // expressões lambda equivalentes utilizando construtor  
    Function<Pessoa, SuperHeroi> transformaEmHeroiConstrutor1 = p -> new  
    SuperHeroi(p);  
    Function<Pessoa, SuperHeroi> transformaEmHeroiConstrutor2 = SuperHeroi::new;  
  
    // expressões lambda equivalentes utilizando chamada de método comum, mas  
    // referenciando o método da classe  
    Function<Pessoa, SuperHeroi> transformaEmHeroiClasse1 = p -> p.vireSuperHeroi();  
    Function<Pessoa, SuperHeroi> transformaEmHeroiClasse2 = Pessoa::vireSuperHeroi;  
  
    // expressões lambda equivalentes utilizando chamada de método comum, mas  
    // referenciando o método do objeto  
    Pessoa pessoa = new Pessoa();  
    Supplier<SuperHeroi> transformaEmHeroiInstancia1 = () -> pessoa.vireSuperHeroi();  
    Supplier<SuperHeroi> transformaEmHeroiInstancia2 = pessoa::vireSuperHeroi;  
}
```

Perceba a diferença entre as expressões lambda:

- Uma parte implementa a interface functional **Function**, pois recebem um argumento e

retornam um valor.

- A última implementa a interface funcional **Supplier**, pois não recebe argumento, mas retorna um valor.

Em caso de dúvidas, consulte novamente os tipos de interfaces funcionais nas outras seções deste capítulo.

8. A variedade de formas para representar uma mesma expressão lambda pode ser grande, então cuidado para não se confundir.

src/org/j6toj8/lambda/methodreference/MethodReference_Variaty.java

```
// ATENÇÃO! Todas essas expressões lambda são equivalentes!

Function<Integer, String> lambda1 = (Integer x) -> { return String.valueOf(x); };
Function<Integer, String> lambda2 = (x) -> { return String.valueOf(x); };
Function<Integer, String> lambda3 = x -> { return String.valueOf(x); };

Function<Integer, String> lambda4 = (Integer x) -> String.valueOf(x);
Function<Integer, String> lambda5 = (x) -> String.valueOf(x);
Function<Integer, String> lambda6 = x -> String.valueOf(x);

Function<Integer, String> lambda7 = String::valueOf;
```

Você já viu todas as formas de criar uma expressão lambda, desde a mais completa até a mais simples. Tenha certeza que conhece todas essas variações para o exame de certificação.

Referências

- Using Method References

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 152). Wiley. Edição do Kindle.

- [Method References in Java.](#)
- [Method References](#). The Java™ Tutorials.

Java Streams

Utilizando Streams

Objetivo

Describe the Stream interface and pipelines; create a stream by using the `Arrays.stream()` and `IntStream.range()` methods; identify the lambda operations that are lazy.

-

Descrever a interface e pipelines de Stream; criar um stream utilizando os métodos `Arrays.stream()` e `IntStream.range()`; identificar quais operações lambda executam sob demanda (`_lazy_`).

Uma das maiores novidades do Java 8 são os *Streams*. Um *Stream* é basicamente um fluxo de dados. Os dados podem ser *Strings*, números, ou qualquer outro objeto. Esses dados passam por uma série de operações, e o conjunto dessas operações é chamado de *pipeline*. Essas operações são quase sempre representadas por expressões lambda, então é muito importante ter dominado todo o capítulo sobre **lambda**, pois todos aqueles conceitos serão utilizados agora para formar um *Stream*.

A partir dos exemplos a seguir, essa explicação ficará mais clara.

Criando um Stream

Geralmente, um *Stream* é criado a partir de um conjunto de dados, como uma lista ou outro tipo de coleção. O objetivo da certificação deixa explícito que é necessário conhecer os métodos `Arrays.stream()` e `IntStream.range()`. Mas, além dessas, serão apresentadas também algumas outras formas comuns de criar um *Stream*.

1. É possível criar um Stream a partir de um `Array` utilizando o método `Arrays.stream()`.

`src/org/j6toj8/streams/usingstreams/Stream_ArraysStream.java`

```
// Criação de um array comum de Strings
String[] array = new String[] { "A", "B", "C" };
// Criação de um Stream a partir do array e, para
// cada elemento, o método println é chamado.
Arrays.stream(array).forEach(System.out::println);
```

Saída no console

A
B
C

2. É possível criar um Stream a partir de uma faixa de números utilizando o método `IntStream.range()`.

src/org/j6toj8/streams/usingstreams/Stream_IntRangeStream.java

```
IntStream.range(0, 4).forEach(System.out::println);
```

Saída no console

```
0  
1  
2  
3
```

Perceba que o primeiro argumento (número 0) é inclusivo, enquanto o segundo argumento (número 4) é exclusivo. Por isso a saída no console apresenta apenas os números 0 a 3.

3. É possível criar um *Stream* a partir de uma lista.

src/org/j6toj8/streams/usingstreams/Streams_ListStream.java

```
List<String> list = Arrays.asList("A", "B", "C");  
list.stream().forEach(System.out::println);
```

Saída no console

```
A  
B  
C
```

4. É possível criar um *Stream* a partir de elementos específicos utilizando o método *Stream.of*.

src/org/j6toj8/streams/usingstreams/Streams_Of.java

```
Stream.of("A", 'B', 1, 2L, 3.0F, 4.0D).forEach(System.out::println);
```

Saída no console

```
A  
B  
1  
2  
3.0  
4.0
```

Nesse caso foi criado um *Stream* que contém: *String*, *Character*, *Integer*, *Long*, *Float* e *Double*.

Operações em Streams

As operações feitas em um *Stream* irão formar seu *pipeline*. As operações que podem ser realizadas

em um Stream são divididas em **operações intermediárias** e **operações finais**. O *Stream* pode conter inúmeras operações intermediárias, porém somente uma final. Nos exemplos anteriores a única operação utilizada foi o `forEach`, que é uma operação final. A seguir serão apresentadas outras operações.

Operações intermediárias

1. É possível ignorar elementos de um stream com a operação `skip`.

src/org/j6toj8/streams/usingstreams/Stream_Skip.java

```
IntStream.range(0, 4) // stream de 0 a 3  
    .skip(2) // ignora 2 elementos  
    .forEach(System.out::println); // imprime os elementos
```

Saída no console

```
2  
3
```

Perceba que nesse caso os elementos 0 e 1 foram ignorados, pois são os dois primeiros elementos do *Stream*. Isso ocorreu pela existência da operação `skip`.

2. É possível limitar a quantidade de elementos que serão processados utilizando a operação `limit`.

src/org/j6toj8/streams/usingstreams/Streams_Limit.java

```
IntStream.range(0, 4) // stream de 0 a 3  
    .limit(2) // limita a 2 elementos  
    .forEach(System.out::println); // imprime os elementos
```

Saída no console

```
0  
1
```

Nesse caso apenas os 2 primeiros elementos foram impressos no console, pois a operação `limit` limitou a quantidade de elementos a serem processados.

3. É possível filtrar elementos do *Stream* utilizando a operação `filter`.

src/org/j6toj8/streams/usingstreams/Streams_Filter.java

```
IntStream.range(0, 4) // stream de 0 a 3  
    .filter(e -> e % 2 == 0) // limita a números pares (resto da divisão por 2 é 0)  
    .forEach(System.out::println); // imprime os elementos
```

Saída no console

```
0  
2
```

Nesse caso apenas os elementos pares foram impressos, pois a operação `filter` limitou àqueles que possuem resto da divisão por 2 igual a 0.

4. É possível filtrar elementos repetidos do *Stream* utilizando a operação `distinct`.

src/org/j6toj8/streams/usingstreams/Streams_Distinct.java

```
// Criação de um array comum de Strings  
String[] array = new String[] { "A", "B", "C", "A", "B", "F" };  
  
Arrays.stream(array)  
    .distinct() // ignora elementos repetidos  
    .forEach(System.out::println);
```

Saída no console

```
A  
B  
C  
F
```

Perceba que nesse caso os elementos repetidos do stream ("A" e "B") foram ignorados, sendo apresentados apenas uma vez.

A operação `distinct` utiliza os métodos `equals` e `hashCode`, então tenha certeza de que eles estão implementados corretamente caso esteja utilizando um tipo de objeto criado por você. No exemplo foram utilizados objetos do tipo `String`, que já possuem essa implementação por padrão.

5. É possível aplicar uma transformação nos elementos do Stream utilizando a operação `map`.

src/org/j6toj8/streams/usingstreams/Streams_Map.java

```
IntStream.range(0, 4) // stream de 0 a 3  
    .map(e -> e * 2) // multiplica os elementos por 2  
    .forEach(System.out::println); // imprime os elementos
```

Saída no console

```
0  
2  
4  
6
```

Perceba que nesse caso os elementos sofreram uma transformação, que foi a multiplicação por 2, antes de serem impressos no console.

6. É possível ordenar os elementos de um Stream utilizando a operação `sorted`.

src/org/j6toj8/streams/usingstreams/Streams_Sorted.java

```
// Criação de um array comum de Strings
String[] array = new String[] { "G", "T", "Y", "A", "B", "C", "A", "B", "F" };

Arrays.stream(array)
    .sorted() // ordena utilizando a ordem natural
    .forEach(System.out::println);
```

Saída no console

```
A
A
B
B
C
F
G
T
Y
```

Nesse caso todos os elementos são ordenados utilizando a ordem natural dos objetos `String`, pois eles já implementam a interface `Comparable`, sendo apresentados em ordem alfabética. Também existe uma versão do método `sort` que recebe como argumento uma implementação de `Comparator`, caso deseje ordenar de outra forma.

7. É possível observar os elementos que passam por um *Stream* utilizando a operação `peek`.

src/org/j6toj8/streams/usingstreams/Streams_Peek.java

```
// Criação de um array comum de Strings
String[] array = new String[] { "G", "T", "Y", "A" };

Arrays.stream(array)
    .peek(e -> System.out.println("Peek: " + e)) // observa o que passou pelo
    Stream
    .forEach(e -> System.out.println("ForEach: " + e));
```

Saída no console

```
Peek: G
ForEach: G
Peek: T
ForEach: T
Peek: Y
ForEach: Y
Peek: A
ForEach: A
```

A operação `peek` funciona apenas para observar o que está passando pelo *Stream*. Pode ser muito útil para realizar *debug* ou *log*. Nesse caso os elementos estão sendo impressos duas vezes no console, pois o método `peek` e o `forEach` estão ambos realizando essa mesma ação. Porém, em aplicações reais, geralmente a operação final **não** será um `forEach`, de tal forma que fará sentido utilizar o `peek`.

8. É possível transformar um *Stream* de vários *Arrays* em um único *Stream* **contínuo** utilizando o método `flatMap`.

src/org/j6toj8streams/usingstreams/Streams_FlatMap.java

```
// Criação de 3 arrays distintos
String[] array1 = new String[] { "A", "B", "C" };
String[] array2 = new String[] { "D", "E", "F" };
String[] array3 = new String[] { "G", "H", "I" };

Stream.of(array1, array2, array3) // criação de um Stream de Arrays
    .flatMap(a -> Arrays.stream(a)) // transforma os dados de cada array em um único
    fluxo de dados
    .forEach(System.out::println); // imprime os elementos
```

Saída no console

```
A
B
C
D
E
F
G
H
I
```

Perceba que nesse caso existem 3 *Arrays* distintos. Então cria-se um *Stream* contendo 3 *Arrays*. O cenário comum seria um que cada elemento do *Stream* fosse um objeto do tipo *Array*. Porém, ao utilizar a operação `flatMap`, é criado um *Stream* para cada um desses *Arrays*, que são unidos e formam um único *Stream* contínuo.

Operações finais

1. É possível executar uma ação final para cada elemento do *Stream* utilizando a operação `forEach`, conforme demonstrado nos exemplos anteriores.
2. É possível recuperar o maior e o menor valor de um *Stream* utilizando as operações finais `max` e `min`. E também é possível recuperar a quantidade de elementos de um *Stream* utilizando a operação final `count`.

`src/org/j6toj8streams/usingstreams/Streams_MaxMinCount.java`

```
Optional<Integer> max = Stream.of(7, 2, 1, 8, 4, 9, 2, 8) // stream de vários Integer
    .max(Comparator.naturalOrder()); // pega o maior número do Stream
System.out.println("Max: " + max.get());

Optional<Integer> min = Stream.of(7, 2, 1, 8, 4, 9, 2, 8) // stream de vários Integer
    .min(Comparator.naturalOrder()); // pega o menor número do Stream
System.out.println("Min: " + min.get());

long count = Stream.of(7, 2, 1, 8, 4, 9, 2, 8) // stream de vários Integer
    .count(); // pega a quantidade de elementos do Stream
System.out.println("Count: " + count);
```

Saída no console

```
Max: 9
Min: 1
Count: 8
```

No caso das operações `max` e `min`, é necessário passar como argumento qual comparador será utilizado. Como os números possuem uma ordem natural, isto é, implementam a interface `Comparable`, é possível utilizar um comparador que usa essa ordem natural, que é o `Comparator.naturalOrder()`. Caso seja um tipo de objeto que não possui uma ordem natural, é necessário passar como argumento uma outra implementação de `Comparator`.

As operações `max` e `min` retornam `Optional` pois, caso o *Stream* esteja vazio, será um `Optional` vazio. Desde o Java 8, com a adição da classe `Optional`, isso tem sido preferido ao invés de retornar `null`, pois facilita a programação funcional. A operação `count` não precisa de um `Optional`, pois mesmo com um *Stream* vazio irá retornar `0`.

3. É possível pegar o primeiro elemento do *Stream* utilizando a operação final `findFirst`, ou um elemento qualquer com `findAny`.

src/org/j6toj8/streams/usingstreams/Streams_FindFirstAny.java

```
Optional<Integer> findFirst = Stream.of(7, 2, 1, 8, 4, 9, 2, 8) // stream de vários Integer
    .findFirst(); // pega o primeiro número do Stream
System.out.println("First: " + findFirst.get());

Optional<Integer> findAny = Stream.of(7, 2, 1, 8, 4, 9, 2, 8) // stream de vários Integer
    .findAny(); // pega qualquer número do Stream
System.out.println("Any: " + findAny.get());
```

Saída no console

```
First: 7
Any: 7
```

Nesse caso, como o *Stream* é sequencial e não paralelo, os dois resultados são iguais. Em *Streams* paralelos, que serão apresentados em uma próxima seção, a operação `findAny` pode trazer resultados diferentes.

Assim como as operações `max` e `min` apresentadas anteriormente, `findAny` e `findFirst` retornam um `Optional` vazio caso o *Stream* esteja vazio.

4. É possível verificar se os elementos do *Stream* atendem a alguma validação utilizando as operações finais `allMatch`, `anyMatch` e `noneMatch`.

src/org/j6toj8/streams/usingstreams/Streams_Match.java

```
boolean anyMatch = Stream.of(7, 2, 1, 8, 4, 9, 2, 8) // stream de vários Integer
    .anyMatch(e -> e > 5); // verifica se algum elemento é maior que 5
System.out.println("anyMatch: " + anyMatch);

boolean allMatch = Stream.of(7, 2, 1, 8, 4, 9, 2, 8) // stream de vários Integer
    .allMatch(e -> e > 5); // verifica se TODOS os elementos são maiores que 5
System.out.println("allMatch: " + allMatch);

boolean noneMatch = Stream.of(7, 2, 1, 8, 4, 9, 2, 8) // stream de vários Integer
    .noneMatch(e -> e > 5); // verifica NENHUM elemento é maior que 5
System.out.println("noneMatch: " + noneMatch);
```

Saída no console

```
anyMatch: true
allMatch: false
noneMatch: false
```

Perceba que na primeira operação é verificado se **qualquer** elemento é maior do que 5. Na

segunda, se **todos** os elementos são maiores do que 5. E na terceira, se **nenhum** elemento é maior do que 5.

5. Não é possível chamar mais de uma operação final no mesmo *Stream*.

src/org/j6toj8/streams/usingstreams/Streams_ReuseStream.java

```
Stream<Integer> stream = Stream.of(7, 2, 1);
stream.forEach(System.out::println); // imprime elementos do Stream
stream.forEach(System.out::println); // LANÇA EXCEÇÃO - o Stream já estava fechado
```

Saída no console

```
7
2
1
Exception in thread "main" java.lang.IllegalStateException: stream has already been
operated upon or closed
at
java.util.stream.AbstractPipeline.sourceStageSpliterator(AbstractPipeline.java:279)
at java.util.stream.ReferencePipeline$Head.forEach(ReferencePipeline.java:580)
at
org.j6toj8.streams.usingstreams.Streams_ReuseStream.main(Streams_ReuseStream.java:1
1)
```

Pipeline

1. É possível criar um *pipeline* com várias operações em um único *Stream*.

src/org/j6toj8/streams/usingstreams/Streams_Pipeline.java

```
IntStream.range(0, 10) // Stream de 0 a 9
    .filter(e -> e % 2 == 0) // mantém apenas números pares
    .skip(2) // ignora os dois primeiros
    .limit(2) // limita a 3 elementos
    .map(e -> e * 2) // multiplia cada elemento por 2
    .forEach(System.out::println); // imprime cada elemento
```

Saída no console

```
8
12
```

Para entender todas as operações realizadas nesse pipeline, é necessário entender passo a passo:

- Foi criado um *Stream* contendo todos os números de 0 a 9
- Foi aplicado um filtro mantendo apenas os números pares: 0, 2, 4, 6 e 8

- c. Foram ignorados os dois primeiros números, mantendo apenas: 4, 6 e 8
 - d. Foi limitado o processamento aos dois primeiros números: 4 e 6
 - e. Foi aplicada uma multiplicação por 2 em cada elemento, resultando em: 8 e 12
 - f. Os dois elementos foram impressos no console.
2. O Stream só será criado de fato depois que alguma operação for executada nele.

src/org/j6toj8/streams/usingstreams/Streams_ChangeBackingList.java

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(1);
list.add(2);
list.add(3);

Stream<Integer> stream = list.stream();

list.add(4);

stream.forEach(System.out::println);
```

Saída no console

```
1
2
3
4
```

Perceba que, mesmo que o *Stream* aparentemente tenha sido criado antes de adicionar o número 4 na lista, ele imprime esse número no console. Isso acontece porque o *Stream* só foi criado de fato quando alguma operação foi feita nele, ou seja, quando o **forEach** foi invocado.

3. É possível encadear a operação final do *Stream* utilizando expressões lambda na classe **Optional**.

src/org/j6toj8/streams/usingstreams/Streams_Optional.java

```
// Exemplo sem encadear a chamada de Optional
Optional<Integer> max = Stream.of(7, 2, 1)
    .max(Comparator.naturalOrder());

max.ifPresent(System.out::println);

// Exemplo encadeando a chamada de Optional
Stream.of(7, 2, 1)
    .max(Comparator.naturalOrder())
    .ifPresent(System.out::println);
```

Perceba que o método **ifPresent** é da classe **Optional**, mesmo que no segundo exemplo possa

parecer que ele faz parte do *Stream*. Em outras palavras, a operação final é `max`, e `ifPresent` é uma chamada em `Optional` e não mais no *Stream*.

Execução sob-demanda (lazy)

As operações intermediárias de um *Stream* só são executadas quando necessário. Ou seja, mesmo que a operação esteja presente no *pipeline*, não é certeza de que ela será executada.

1. Nada será feito se o *Stream* não contiver uma operação final.

`src/org/j6toj8/streams/usingstreams/Streams_LazyNoFinal.java`

```
IntStream.range(0, 4)
    .filter(e -> e % 2 == 0)
    .limit(3)
    .map(e -> e * 2)
    .peek(System.out::println);
```

Nesse caso nada é impresso no console, pois **nenhuma** operação **final** foi aplicada no *Stream*. Ou seja, se não há nada consumindo o resultado desse *Stream*, o Java nem precisa executar o *pipeline* criado.

2. Outras operações intermediárias também não costumam ser executadas se não for necessário.

`src/org/j6toj8/streams/usingstreams/Streams_LazyMap.java`

```
IntStream.range(0, 10)
    .peek(e -> System.out.println("Peek: " + e))
    .limit(3)
    .forEach(e -> System.out.println("ForEach: " + e));
```

Saída no console

```
Peek: 0
ForEach: 0
Peek: 1
ForEach: 1
Peek: 2
ForEach: 2
```

Perceba que, mesmo que a operação `peek` esteja antes da operação `limit`, ela não é executada para todos os elementos do *Stream*, apenas para aqueles que serão realmente utilizados.

Streams primitivos

Existem Streams específicos para alguns tipos primitivos como `double`, `int` e `long`. Eles possuem a vantagem de evitar o *Boxing* e *Unboxing*, fornecendo alguns métodos mais especializados como demonstrado a seguir.

1. É possível criar *Streams* de tipos primitivos com as classes: `DoubleStream`, `IntStream` e `LongStream`.

src/org/j6toj8/streams/usingstreams/primitives/Streams_Primitives.java

```
System.out.println(" DoubleStream");
DoubleStream.of(1.1, 2.2, 3.3).forEach(System.out::print);

System.out.println("\n IntStream");
IntStream.of(1, 2, 3).forEach(System.out::print);
System.out.println();
IntStream.range(1, 4).forEach(System.out::print);

System.out.println("\n LongStream");
LongStream.of(1, 2, 3).forEach(System.out::print);
System.out.println("");
LongStream.range(1, 4).forEach(System.out::print);
```

Saída no console

```
DoubleStream
1.12.23.3
IntStream
123
123
LongStream
123
123
```

2. É possível transformar um *Stream* comum em um *Stream* de primitivos utilizando as operações `mapTo*`.

src/org/j6toj8/streams/usingstreams/primitives/Streams_MapTo.java

```
List<Integer> list = Arrays.asList(1, 2, 3, 4);

System.out.println(" Stream para IntStream");
list.stream() // cria Stream<Integer>
    .mapToInt(Integer::intValue) // transforma em IntStream
    .forEach(System.out::print);

System.out.println("\n Stream para LongStream");
list.stream() // cria Stream<Long>
    .mapToLong(Integer::longValue) // transforma em LongStream
    .forEach(System.out::print);

System.out.println("\n Stream para DoubleStream");
list.stream() // cria Stream<Double>
    .mapToDouble(Integer::doubleValue) // transforma em DoubleStream
    .forEach(System.out::print);
```

Saída no console

```
Stream para IntStream  
1234  
Stream para LongStream  
1234  
Stream para DoubleStream  
1.02.03.04.0
```

3. É possível gerar Streams infinitos com o método `generate`.

src/org/j6toj8streams/usingstreams/primitives/Streams_Generate.java

```
// Cria Stream infinito de números aleatórios  
System.out.println(" IntStream infinito de números aleatórios");  
IntStream.generate(() -> new Random().nextInt())  
.limit(3)  
.forEach(System.out::println);  
  
System.out.println("\n DoubleStream infinito de números aleatórios");  
DoubleStream.generate(Math::random)  
.limit(3)  
.forEach(System.out::println);
```

Saída no console

```
IntStream infinito de números aleatórios  
2111846625  
-1692075394  
122693397  
  
DoubleStream infinito de números aleatórios  
0.913037010633669  
0.23669861350384735  
0.32655918031847697
```

Nesse caso os *Streams* são realmente infinitos. Só foram apresentados 3 números de cada pois existe a operação `limit`, caso contrário a execução do programa também seria sem fim.

4. É possível utilizar a operação `rangeClosed` ao invés de `range`, deixando o código mais legível.

src/org/j6toj8streams/usingstreams/primitives/Streams_RangeClosed.java

```
IntStream.range(1, 4).forEach(System.out::print);  
System.out.println();  
IntStream.rangeClosed(1, 4).forEach(System.out::print);
```

Saída no console

```
123  
1234
```

Perceba que na chamada utilizando `range`, o último número é exclusivo (não faz parte do *Stream*). No `rangeClosed`, tanto o primeiro quanto o último número são inclusivos (fazem parte do *Stream*).

- É possível gerar várias estatísticas de *Streams* utilizando a operação `summaryStatistics`.

```
src/org/j6toj8/streams/usingstreams/primitives/Streams_Statistics.java
```

```
IntSummaryStatistics summaryStatistics = IntStream.range(0, 10).  
    summaryStatistics();  
System.out.println("Quantidade: " + summaryStatistics.getCount());  
System.out.println("Maior: " + summaryStatistics.getMax());  
System.out.println("Menor: " + summaryStatistics.getMin());  
System.out.println("Soma: " + summaryStatistics.getSum());  
System.out.println("Média: " + summaryStatistics.getAverage());
```

Saída no console

```
Quantidade: 10  
Maior: 9  
Menor: 0  
Soma: 45  
Média: 4.5
```

Reduce e Collectors

Reduce

Reduce é uma das principais operações final que podem ser feitas em um *Stream*. *Reduce* é uma operação que transforma os vários valores do *Stream* em um único valor. Várias operações apresentadas anteriormente são um tipo de *Reduce*, como: `max`, `min` e `summaryStatistics`. Porém, nem sempre essas operações são suficientes, e por isso existem os métodos `reduce`. Eles permitem a implementação de operações personalizadas de *Reduce*.

- É possível criar uma operação de *Reduce* personalizada com o método `reduce()` que recebe 1 argumento.

```
src/org/j6toj8/streams/usingstreams/primitives/Streams_Reduce.java
```

```
Optional<Integer> reduce = Stream.of(7, 2, 3, 8)  
.reduce((e1, e2) -> e1 * e2); // reduce que multiplica todos os números  
  
System.out.println(reduce.get());
```

Saída no console

336

Nesse caso está sendo feito um *Reduce* onde o resultado da operação anterior é passado para a próxima execução. Ou seja, primeiro é feita a multiplicação de $7 * 2$, que é 14. Então a função é chamada novamente passando como argumento o resultado anterior (14) e o próximo número do Stream (3). O resultado é 42. Então a função é chamada uma última vez passando o resultado anterior (42) e o próximo número do Stream (8), o que dá o resultado de 336.

2. É possível criar uma operação de *Reduce* informando o valor de identidade.

src/org/j6toj8/streams/usingstreams/primitives/Streams_ReduceIdentity.java

```
Integer reduce = Stream.of(7, 2, 3, 8)
    .reduce(1, (e1, e2) -> e1 * e2);

System.out.println(reduce);
```

Saída no console

336

Nesse caso é possível informar o valor de identidade da função. O conceito de valor ou função de identidade são um pouco mais complexos, mas para a certificação apenas compreenda que ele representa um valor neutro. Ou seja, para a operação de multiplicação, o valor de identidade é 1, pois qualquer valor multiplicado por 1 resulta nele mesmo. Caso fosse uma operação de soma, o valor de identidade seria 0, pois qualquer valor somado a 0 resulta nele mesmo.

Além disso, se o *Stream* estiver vazio, o valor de identidade será retornado. Por isso, diferente do exemplo anterior, não é necessário retornar um *Optional*.

3. É possível criar uma operação de *Reduce* que pode ser executada em várias *Threads* e depois combinada em um único valor.

src/org/j6toj8/streams/usingstreams/primitives/Streams_ReduceCombiner.java

```
Integer reduce = Stream.of(7, 2, 3, 8)
    .reduce(1, (e1, e2) -> e1 * e2, (e1, e2) -> e1 * e2);

System.out.println(reduce);
```

Saída no console

336

Nesse caso é passado um argumento adicional. Ele é a função de combinação. Essa função é utilizada quando o *Stream* é paralelo, ou seja, utiliza mais de uma *thread*. Ela pega o valor

retornado por 2 ou mais *threads* e combina-os em um único valor. Em uma operação de multiplicação, a combinação também é uma multiplicação. Ou seja, caso a primeira *thread* multiplique 7 e 2, resultando em 14, e a segunda multiplique 3 e 8, resultando em 24, a função de combinação só precisa multiplicar 14 por 24 para chegar ao valor de 336. Sendo assim, a função de combinação só faz sentido em um *Stream* paralelo, que será apresentado no próximo capítulo.

Collect

A operação final `collect` também é um tipo de *Reduce*, porém é utilizada para objetos mutáveis. Ou seja, ao invés de utilizar a operação `reduce` com `String`, provavelmente seria mais eficiente utilizar a operação `collect` com a classe `StringBuilder`, para evitar a criação de vários objetos do tipo `String`. Como Java utiliza muitos objetos mutáveis, incluindo listas e mapas, geralmente a operação `collect` será mais eficiente do que a `reduce`.

Por serem muito comuns, existem vários `Collectors` já implementados no Java, disponíveis na classe `Collectors`.

1. É possível utilizar um `Collector` que junta várias `Strings`.

```
src/org/j6toj8/streams/usingstreams/primitives/Streams_CollectorJoining.java
```

```
String collect = Stream.of("A", "B", "C")
    .collect(Collectors.joining());
System.out.println(collect);
```

Saída no console

```
ABC
```

2. É possível utilizar um `Collector` para representar cada elemento como um número e calcular a média entre eles.

```
src/org/j6toj8/streams/usingstreams/primitives/Streams_CollectorAveragingInt.java
```

```
// Calcula a média do tamanho de cada nome
Double collect = Stream.of("Rinaldo", "Rodrigo", "Luiz", "Amélia", "Roseany")
    .collect(Collectors.averagingInt(s -> s.length()));
System.out.println(collect);
```

Saída no console

```
6.2
```

3. É possível utilizar um `Collector` para armazenar os elementos de um *Stream* em uma nova coleção.

```
// Armazena o resultado do Stream em um ArrayList  
Collection<Integer> collect = Stream.of(1, 2, 3, 4)  
    .collect(Collectors.toCollection(ArrayList::new));  
System.out.println("ArrayList: " + collect);  
  
// Armazena o resultado do Stream em um HashSet  
Collection<Integer> collect2 = Stream.of(1, 2, 3, 4)  
    .collect(Collectors.toCollection(HashSet::new));  
System.out.println("HashSet: " + collect2);  
  
// Armazena o resultado do Stream em uma LinkedList  
Collection<Integer> collect3 = Stream.of(1, 2, 3, 4)  
    .collect(Collectors.toCollection(LinkedList::new));  
System.out.println("LinkedList: " + collect3);  
  
// Armazena o resultado do Stream em um TreeSet  
Collection<Integer> collect4 = Stream.of(1, 2, 3, 4)  
    .collect(Collectors.toCollection(TreeSet::new));  
System.out.println("TreeSet: " + collect4);
```

Saída no console

```
ArrayList: [1, 2, 3, 4]  
HashSet: [1, 2, 3, 4]  
LinkedList: [1, 2, 3, 4]  
TreeSet: [1, 2, 3, 4]
```

4. É possível utilizar um **Collector** para armazenar os elementos de um *Stream* em um mapa.

```
// Armazena o resultado do Stream em um Mapa  
// A Chave é o próprio nome (s -> s)  
// O Valor é o tamanho do nome  
Map<String, Integer> collect = Stream.of("Rinaldo", "Rodrigo", "Luiz", "Amélia",  
    "Roseany")  
    .collect(Collectors.toMap(s -> s, s -> s.length()));  
  
System.out.println(collect);
```

Saída no console

```
{Roseany=7, Amélia=6, Rodrigo=7, Rinaldo=7, Luiz=4}
```

5. Também é possível armazenar em um mapa para casos em que a chave for se repetir. O terceiro argumento do método **toMap** define a regra de mesclagem dos valores para chaves iguais.

src/org/j6toj8streams/usingstreams/primitives/Streams_CollectorToMapDuplicateKey.java

```
// Armazena o resultado do Stream em um Mapa
// A Chave é o tamanho do nome
// O Valor são os nomes com aquele tamanho
Map<Object, Object> collect = Stream.of("Rinaldo", "Rodrigo", "Luiz", "Amélia",
    "Roseany")
    .collect(Collectors.toMap(s -> s.length(), s -> s, (s1, s2) -> s1 + "," + s2));

System.out.println(collect);
```

Saída no console

```
{4=Luiz, 6=Amélia, 7=Rinaldo,Rodrigo,Roseany}
```

6. É possível utilizar um **Collector** que cria um mapa agrupando valores que tem a mesma chave em uma lista.

src/org/j6toj8streams/usingstreams/primitives/Streams_CollectorGroupingBy.java

```
// Armazena o resultado do Stream em um Mapa
// A Chave é o tamanho do nome
// O Valor é uma lista com os nomes que tem aquele tamanho
Map<Object, List<String>> collect = Stream.of("Rinaldo", "Rodrigo", "Luiz", "Amélia", "Roseany")
    .collect(Collectors.groupingBy(s -> s.length()));

System.out.println(collect);
```

Saída no console

```
{4=[Luiz], 6=[Amélia], 7=[Rinaldo, Rodrigo, Roseany]}
```

7. Também é possível personalizar a maneira que os valores com chaves iguais serão combinados.

src/org/j6toj8streams/usingstreams/primitives/Streams_CollectorGroupingByDownstream.java

```
// Armazena o resultado do Stream em um Mapa
// A Chave é o tamanho do nome
// O Valor são os nomes que tem aquele tamanho
Map<Object, String> collect = Stream.of("Rinaldo", "Rodrigo", "Luiz", "Amélia",
    "Roseany")
    .collect(Collectors.groupingBy(s -> s.length(), Collectors.joining(",")));

System.out.println(collect);
```

Saída no console

```
{4=Luiz, 6=Amélia, 7=Rinaldo,Rodrigo,Roseany}
```

Perceba que nesse caso os valores foram combinados utilizando outro *Collector*, que agrupou os nomes separando com vírgula.

8. Também é possível definir qual tipo de mapa será utilizado para agrupar.

src/org/j6toj8/streams/usingstreams/primitives/Streams_CollectorGroupingByMapFactory.java

```
// Armazena o resultado do Stream em um Mapa
// A Chave é o tamanho do nome
// O Valor são os nomes que tem aquele tamanho
Map<Object, String> collect = Stream.of("Rinaldo", "Rodrigo", "Luiz", "Amélia",
"Roseany")
.collect(Collectors.groupingBy(s -> s.length(), TreeMap::new, Collectors.joining
(", ")));

System.out.println(collect);
```

Saída no console

```
{4=Luiz, 6=Amélia, 7=Rinaldo,Rodrigo,Roseany}
```

Perceba que o resultado desse exemplo é idêntico ao anterior, porém foi passado um argumento a mais, que é o construtor do mapa que deveria ser utilizado.

9. É possível utilizar um *Collector* que particiona valores em *True* ou *False* a partir de um função do tipo *Predicate*.

src/org/j6toj8/streams/usingstreams/primitives/Streams_CollectorPartitioningBy.java

```
// Armazena o resultado do Stream em um Mapa
// As Chaves são true ou false
// O Valor é uma lista dos valores que atendem ou não a regra de particionamento
Map<Boolean, List<String>> collect = Stream.of("Rinaldo", "Rodrigo", "Luiz", "Amélia", "Roseany")
.collect(Collectors.partitioningBy(s -> s.startsWith("R")));

System.out.println(collect);
```

Saída no console

```
{false=[Luiz, Amélia], true=[Rinaldo, Rodrigo, Roseany]}
```

Perceba que nesse caso a regra de particionamento são os nomes que iniciam-se por *R*.

10. Também é possível personalizar como a combinação dos valores particionados será feita.

```
src/org/j6toj8/streams/usingstreams/primitives/Streams_CollectorPartitioningByDownstream.java

// Armazena o resultado do Stream em um Mapa
// As Chaves são true ou false
// O Valor é uma String que são os nomes que atendem ou não a regra de
particionamento
Map<Boolean, String> collect = Stream.of("Rinaldo", "Rodrigo", "Luiz", "Amélia",
"Roseany")
    .collect(Collectors.partitioningBy(s -> s.startsWith("R")), Collectors.joining(
", "));

System.out.println(collect);
```

Saída no console

```
{false=Luiz,Amélia, true=Rinaldo,Rodrigo,Roseany}
```

Perceba que nesse caso os valores foram combinados utilizando um outro **Collector**, que juntou os valores daquela mesma chave em uma única **String** separados por vírgula.

11. É possível adicionar uma camada a mais de transformação ao utilizar um **Collector**, utilizando o método **mapping**.

```
src/org/j6toj8/streams/usingstreams/primitives/Streams_CollectorMapping.java
```

```
// Armazena o resultado do Stream em um Mapa
// A Chave é o tamanho do nome
// O Valor são os nomes que tem aquele tamanho, convertidos para maiúscula,
separados por vírgula
Map<Integer, String> collect = Stream.of("Rinaldo", "Rodrigo", "Luiz", "Amélia",
"Roseany")
    .collect(Collectors.groupingBy(s -> s.length(), Collectors.mapping(s ->
s.toUpperCase(), Collectors.joining(", "))));

System.out.println(collect);
```

Saída no console

```
{4=LUIZ, 6=AMÉLIA, 7=RINALDO,RODRIGO,ROSEANY}
```

Esse tipo de código, apesar de complexo, pode aparecer no exame de certificação. É recomendado praticar esses exemplos com uma IDE para entender de fato seus comportamentos. Acesse os códigos de exemplo deste livro para facilitar seus estudos.

Referências

- Using Streams

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 185). Wiley. Edição do Kindle.

- [The Java 8 Stream API Tutorial.](#)
- [Lesson: Aggregate Operations.](#) The Java™ Tutorials.
- [Package java.util.stream.](#) Java Platform SE 8.
- [Interface Stream<T>.](#) Java Platform SE 8.

Streams Paralelos

Objetivo

Develop code that uses parallel streams, including decomposition operation and reduction operation in streams

-

Desenvolver código que usa Streams Paralelos, incluindo operação de decomposição e operação de redução em Streams

Streams podem ser sequenciais ou paralelos. Os sequencias foram vistos na seção anterior, enquanto os paralelos serão apresentados nesta seção. Streams paralelos são executados por mais de uma *Thread*, geralmente uma quantidade igual à quantidade de núcleos do processador onde a aplicação está sendo executada. Apesar disso, nem sempre é útil utilizá-los. Seu ganho real é em Streams com grande volumes de dados. Em um Stream pequeno, transformá-lo em paralelo pode até causar uma perda de performance.

Ao utilizar qualquer tipo de *Stream*, é recomendável não executar funções lambdas que causem efeitos colaterais, como mudanças no estado de objetos. Em Streams paralelos essa recomendação é ainda mais importante.

1. É possível transformar qualquer Stream em paralelo utilizando o método `parallel`.

`src/org/j6toj8/streams/parallelstreams/Streams_Parallel.java`

```
String[] array = new String[] { "A", "B", "C" };
Arrays.stream(array)
    .parallel() // stream transformado em paralelo
    .forEach(System.out::println);
```

2. É possível criar Streams paralelos diretamente em Coleções através do método `parallelStream`.

src/org/j6toj8streams/parallelstreams/Streams_ParallelStream.java

```
List<String> list = Arrays.asList("A", "B", "C");
list.parallelStream() // cria um Stream paralelo diretamente
    .forEach(System.out::println);
```

3. Ao utilizar a operação `forEach` em um *Stream* paralelo, a ordem de execução não é garantida.

src/org/j6toj8streams/parallelstreams/Streams_ParallelForEach.java

```
List<String> list = Arrays.asList("A", "B", "C");

System.out.println("Sequencial: ");
list.stream() // cria um Stream sequencial
    .forEach(System.out::println);

System.out.println("Paralelo: ");
list.parallelStream() // cria um Stream paralelo
    .forEach(System.out::println);
```

Saída no console

```
Sequencial:
A
B
C
Paralelo:
B
C
A
```

O *Stream* paralelo poderia ter impresso em qualquer ordem, pois não há nenhuma garantia na ordem em que os elementos serão tratados.

4. A operação `forEachOrdered` garante que a ordem será mantida mesmo em *Streams* paralelos.

src/org/j6toj8streams/parallelstreams/Streams_ParallelForEachOrdered.java

```
List<String> list = Arrays.asList("A", "B", "C");

System.out.println("Sequencial: ");
list.stream() // cria um Stream sequencial
    .forEachOrdered(System.out::println);

System.out.println("Paralelo: ");
list.parallelStream() // cria um Stream paralelo
    .forEachOrdered(System.out::println);
```

Saída no console

Sequencial:

A
B
C

Paralelo:

A
B
C

5. Em coleções com muitos objetos pode haver um ganho considerável de performance.

src/org/j6toj8/streams/parallelstreams/Streams_ParallelPerformance.java

```
long inicio = System.currentTimeMillis();
IntStream.range(0, Integer.MAX_VALUE) // stream sequencial
    .mapToDouble(n -> Math.pow(n, 2))
    .average()
    .ifPresent(n -> System.out.println("Tempo stream sequencial: " +
        (System.currentTimeMillis() - inicio)));

long inicio2 = System.currentTimeMillis();
IntStream.range(0, Integer.MAX_VALUE)
    .parallel() // stream paralelo
    .mapToDouble(n -> Math.pow(n, 2))
    .average()
    .ifPresent(n -> System.out.println("Tempo stream paralelo: " +
        (System.currentTimeMillis() - inicio2)));



```

Saída no console

```
Tempo stream sequencial: 9863
Tempo stream paralelo: 1479
```

Perceba que na máquina onde o código foi executado, a execução em paralelo demorou apenas 15% do tempo da execução sequencial. Esse não é um teste minucioso, mas mostra o potencial de *Streams* paralelos.

6. Operações intermediárias que alteram o estado de objetos podem gerar resultados inesperados ao serem executadas em paralelo.

src/org/j6toj8streams/parallelstreams/Streams_ParallelStatefulOperation.java

```
List<String> synchronizedList = Collections.synchronizedList(new ArrayList<>());
List<String> list = Arrays.asList("A", "B", "C");

System.out.println("Ordem no forEachOrdered: ");
list.parallelStream()
    .map(s -> {synchronizedList.add(s); return s;}) // operação com efeito
    colateral - altera o estado de um objeto
    .forEachOrdered(System.out::println);

System.out.println("Ordem na synchronizedList: ");
for (String s : synchronizedList) {
    System.out.println(s);
}
```

Saída no console

```
Ordem no forEachOrdered:
```

```
A
```

```
B
```

```
C
```

```
Ordem na synchronizedList:
```

```
A
```

```
C
```

```
B
```

Perceba que a ordem foi respeitada na última operação do *Stream*, o `forEachOrdered`, mas não foi respeitada na execução da operação intermediária `map`. Isso ocorre porque essa operação intermediária não precisa seguir a ordem dos itens do stream.

7. Diferente da execução em um *Stream* sequencial, a operação `findAny` traz resultados realmente aleatórios ao ser executada em um *Stream* paralelo.

src/org/j6toj8streams/parallelstreams/Streams_ParallelFindAny.java

```
Optional<Integer> findAny1 = Stream.of(7, 2, 1, 8, 4, 9, 2, 8)
    .findFirst();
System.out.println("findAny Sequencial: " + findAny1.get());

Optional<Integer> findAny2 = Stream.of(7, 2, 1, 8, 4, 9, 2, 8)
    .parallel()
    .findAny();
System.out.println("findAny Paralelo: " + findAny2.get());
```

Saída no console

```
findAny Sequencial: 7  
findAny Paralelo: 9
```

8. Ao realizar uma operação de reduce não há problema caso o acumulador seja associativo.

src/org/j6toj8/streams/parallelstreams/Streams_ParallelReduceAssociative.java

```
Stream.of(7, 2, 3, 8, 2, 1, 4, 5)  
    .reduce((e1, e2) -> e1 * e2)  
    .ifPresent(System.out::println);  
  
Stream.of(7, 2, 3, 8, 2, 1, 4, 5)  
    .parallel()  
    .reduce((e1, e2) -> e1 * e2)  
    .ifPresent(System.out::println);
```

Saída no console

```
13440  
13440
```

Perceba que o resultado com o Stream sequencial é idêntico ao paralelo. Isso ocorre porque a operação de multiplicação é associativa, ou seja, fazer $(2 \times 2) \times (3 \times 3)$ é o mesmo que fazer $(2 \times 2 \times 3) \times 3$, ou até mesmo $2 \times (2 \times 3) \times 3$.

9. Ao realizar uma operação de reduce acumuladores não-associativos irá gerar resultados inesperados.

src/org/j6toj8/streams/parallelstreams/Streams_ParallelReduceNonAssociative.java

```
Stream.of(7, 2, 3, 8, 2, 1, 4, 5)  
    .reduce((e1, e2) -> e1 - e2)  
    .ifPresent(System.out::println);  
  
Stream.of(7, 2, 3, 8, 2, 1, 4, 5)  
    .parallel()  
    .reduce((e1, e2) -> e1 - e2)  
    .ifPresent(System.out::println);
```

Saída no console

```
-18  
8
```

Isso ocorre pois a operação de subtração não é associativa, então o resultado pode variar conforme o Stream for "fatiado" para ser executado em paralelo. Ou seja, fazer $1 - 2 - 3 - 4$

não é o mesmo que fazer (1 - 2) - (3 - 4).

10. Para coletar o resultado de um *Stream* paralelo em um mapa, utilize a operação `toConcurrentMap`.

src/org/j6toj8/streams/parallelstreams/Streams_ParallelToConcurrentMap.java

```
Map<String, Integer> collect1 = Stream.of("Rinaldo", "Rodrigo", "Luiz", "Amélia",
    "Roseany")
    .parallel()
    .collect(Collectors.toMap(s -> s, s -> s.length()));
System.out.println("toMap: " + collect1);

Map<String, Integer> collect2 = Stream.of("Rinaldo", "Rodrigo", "Luiz", "Amélia",
    "Roseany")
    .parallel()
    .collect(Collectors.toConcurrentMap(s -> s, s -> s.length()));
System.out.println("toConcurrentMap: " + collect2);
```

Saída no console

```
toMap: {Roseany=7, Amélia=6, Rodrigo=7, Rinaldo=7, Luiz=4}
toConcurrentMap: {Amélia=6, Roseany=7, Rodrigo=7, Rinaldo=7, Luiz=4}
```

Perceba que o resultados das operações pode ser diferente. Ao utilizar o `Collector toConcurrentMap` em um *Stream* paralelo, as operações podem ser executadas em qualquer ordem e não há necessidade de criar múltiplos *Map's* para serem combinados posteriormente. Em grandes *Streams*, isso pode ocasionar em um ganho de performance.

11. Para coletar o resultado de um Stream paralelo utilize `groupingByConcurrent` ao invés de `groupingBy`.

src/org/j6toj8/streams/parallelstreams/Streams_ParallelGroupingByConcurrent.java

```
Map<Object, List<String>> collect1 = Stream.of("Rinaldo", "Rodrigo", "Luiz", "Amélia",
    "Roseany")
    .parallel()
    .collect(Collectors.groupingBy(s -> s.length()));
System.out.println(collect1);

Map<Object, List<String>> collect2 = Stream.of("Rinaldo", "Rodrigo", "Luiz", "Amélia",
    "Roseany")
    .parallel()
    .collect(Collectors.groupingByConcurrent(s -> s.length()));
System.out.println(collect2);
```

Saída no console

```
{4=[Luiz], 6=[Amélia], 7=[Rinaldo, Rodrigo, Roseany]}\n{4=[Luiz], 6=[Amélia], 7=[Roseany, Rodrigo, Rinaldo]}
```

Pelo mesmo motivo do exemplo anterior, a ordem pode variar ao utilizar o **groupingByConcurrent**, porém pode haver ganho de performance em grandes *Streams* paralelos, pois a ordem não é necessariamente seguida e não há necessidade de criar múltiplos mapas.

Referências

- Working with Parallel Streams

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 366). Wiley. Edição do Kindle.

- [The Java 8 Stream API Tutorial](#).
- [Parallelism](#). The Java™ Tutorials.
- [Package java.util.stream](#). Java Platform SE 8.
- [Interface Stream<T>](#). Java Platform SE 8.

Concurrency

Pacote Concurrent

Objetivo

Use classes from the `java.util.concurrent` package including `CyclicBarrier` and `CopyOnWriteArrayList` with a focus on the advantages over and differences from the traditional `java.util` collections

- Usar classes do pacote `java.util.concurrent`, incluindo `CyclicBarrier` e `CopyOnWriteArrayList`, com foco nas vantagens e diferenças das Coleções tradicionais do pacote `java.util`

O pacote `java.util.concurrent` inclui inúmeras classes para serem utilizadas em aplicações multi-thread. Nesta seção serão apresentadas algumas dessas classes.

Muitas das classes do pacote concurrent são apenas versões das coleções comuns, porém com blocos `synchronized`, garantindo que múltiplas *threads* poderão acessá-las ao mesmo tempo mantendo sua integridade. As classes `ConcurrentHashMap`, `ConcurrentLinkedQueue` e `ConcurrentLinkedDeque` são exemplos disso. Por isso é importante conhecer e lembrar das coleções comuns do Java 6.

Todas as seções deste capítulo podem conter exemplos maiores do que os que foram apresentados até agora, principalmente quando for necessário a criação de múltiplas *Threads*. É importante dedicar um tempo maior para entender cada um desses exemplos.

1. É possível criar uma **Fila** que lança uma exceção após um tempo predefinido utilizando a classe `LinkedBlockingQueue`.

`src/org/j6toj8/concurrency/concurrentpackage/Concurrency_LinkedBlockingQueue.java`

```
BlockingQueue<String> queue = new LinkedBlockingQueue<String>();

try {
    queue.offer("ABC", 1, TimeUnit.SECONDS);
} catch (InterruptedException e) {
    System.out.println("Não conseguiu inserir em menos de 1 segundo.");
}

try {
    queue.poll(1, TimeUnit.SECONDS);
} catch (InterruptedException e) {
    System.out.println("Não conseguiu remover em menos de 1 segundo.");
}
```

2. É possível criar uma **Fila Duplamente Ligada (Deque)** que lança uma exceção após um tempo predefinido utilizando a classe `LinkedBlockingDeque`.

```
BlockingDeque<String> queue = new LinkedBlockingDeque<String>();

try {
    queue.offerFirst("ABC", 1, TimeUnit.SECONDS);
    queue.offerLast("DEF", 1, TimeUnit.SECONDS);
} catch (InterruptedException e) {
    System.out.println("Não conseguiu inserir em menos de 1 segundo.");
}

try {
    queue.pollFirst(1, TimeUnit.SECONDS);
    queue.pollLast(1, TimeUnit.SECONDS);
} catch (InterruptedException e) {
    System.out.println("Não conseguiu remover em menos de 1 segundo.");
}
```

3. É possível criar uma lista que aloca todo o seu conteúdo em um novo *array* sempre que é modificada utilizando a classe `CopyOnWriteArrayList`.

```
List<String> list = new CopyOnWriteArrayList<String>();
list.add("A");
list.add("B");
list.add("C");

for (String string : list) {
    System.out.println(string);
    if (string.equals("A")) {
        list.add("D");
    }
}

System.out.println("Lista final: " + list);
```

Saída no console

```
A
B
C
Lista final: [A, B, C, D]
```

Perceba que foi possível acrescentar um valor na lista durante a execução do `forEach`. Em uma lista tradicional haveria ocorrido uma `ConcurrentModificationException`.

Perceba também que, mesmo alterando a lista dentro do `forEach`, a letra "D" não aparece no console. Isso ocorre pois, quando a letra "D" foi inserida na lista, um novo *array* foi alocado

internamente contendo todos os novos elementos, e a iteração continuou ocorrendo no *array* antigo.

4. É possível criar versões *synchronized* de coleções utilizando métodos utilitários da classe **Collections**.

src/org/j6toj8/concurrency/concurrentpackage/Concurrency_CollectionsSyncronized.java

```
// Concurrent Map, garante o acesso de múltiplas threads
Map<String, String> concurrentHashMap = new ConcurrentHashMap<>();

// Map Comum, NÃO garante o acesso de múltiplas threads
Map<String, String> map = new HashMap<>();

// Syncronized Map, garante o acesso de múltiplas threads
Map<String, String> synchronizedMap = Collections.synchronizedMap(map);
```

5. **Não** é possível adicionar ou remover elementos durante o *forEach* de uma coleção sincronizada que foi criada utilizando o método da classe **Collections**.

src/org/j6toj8/concurrency/concurrentpackage/Concurrency_CollectionsSyncronizedForEach.java

```
Map<Integer, String> map = new HashMap<Integer, String>();
map.put(1, "A");
map.put(2, "B");
map.put(3, "C");
Map<Integer, String> synchronizedMap = Collections.synchronizedMap(map);

for (Entry<Integer, String> entry : synchronizedMap.entrySet()) {
    synchronizedMap.remove(1);
}
```

Saída no console

```
Exception in thread "main" java.util.ConcurrentModificationException
  at java.util.HashMap$HashIterator.nextNode(HashMap.java:1445)
  at java.util.HashMap$EntryIterator.next(HashMap.java:1479)
  at java.util.HashMap$EntryIterator.next(HashMap.java:1477)
  at
org.j6toj8.concurrency.concurrentpackage.Concurrency_CollectionsSyncronizedForEach.
main(Concurrency_CollectionsSyncronizedForEach.java:18)
```

Perceba que, apesar do **Map** ter sido transformado em *synchronized*, não é possível alterá-lo durante uma iteração com *forEach*. Isso é possível apenas nas versões *Concurrent* das coleções.

6. É possível sincronizar a execução de várias *threads* utilizando a classe **CyclicBarrier**.

```
// Classe que será executada por 3 threads
static class Acao implements Runnable {

    CyclicBarrier cyclicBarrier;

    public Acao(CyclicBarrier cyclicBarrier) {
        this.cyclicBarrier = cyclicBarrier;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ": Primeira Parte");

        try {
            cyclicBarrier.await(); // sincronização das threads
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName() + ": Segunda Parte");

        try {
            cyclicBarrier.await(); // sincronização das threads
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName() + ": Terceira Parte");
    }
}

public static void main(String[] args) {
    // Criação de um CyclicBarrier para 3 threads
    CyclicBarrier cyclicBarrier = new CyclicBarrier(3);

    // Criação das threads
    Thread thread1 = new Thread(new Acao(cyclicBarrier));
    Thread thread2 = new Thread(new Acao(cyclicBarrier));
    Thread thread3 = new Thread(new Acao(cyclicBarrier));

    // Início de execução das threads
    thread1.start();
    thread2.start();
    thread3.start();
}
```

Saída no console

```
Thread-2: Primeira Parte
Thread-1: Primeira Parte
Thread-0: Primeira Parte
Thread-1: Segunda Parte
Thread-2: Segunda Parte
Thread-0: Segunda Parte
Thread-0: Terceira Parte
Thread-1: Terceira Parte
Thread-2: Terceira Parte
```

Neste exemplo estão sendo criadas 3 *threads*. Todas executam instâncias da classe `Acao` que recebem a mesma instância da classe `CyclicBarrier`. Toda vez que uma *thread* faz uma chamada ao método `await` da instância de `cyclicBarrier`, ela fica suspensa até que todas as outras *threads* cheguem até aquele mesmo ponto. Por isso todas as *threads* imprimem no console de forma sincronizada. Se não houvesse sincronização, a saída no console seria completamente imprevisível. Abaixo é o exemplo de uma execução sem a chamada ao método `await`:

Saída no console caso não houvesse CyclicBarrier

```
Thread-0: Primeira Parte
Thread-1: Primeira Parte
Thread-1: Segunda Parte
Thread-1: Terceira Parte
Thread-2: Primeira Parte
Thread-0: Segunda Parte
Thread-2: Segunda Parte
Thread-0: Terceira Parte
Thread-2: Terceira Parte
```

- Using Concurrent Collections

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 358). Wiley. Edição do Kindle.

- [Overview of the java.util.concurrent](#).
- [CyclicBarrier in Java](#).
- [Concurrent Collections](#). The Java™ Tutorials.
- [Package java.util.concurrent](#). Java Platform SE 8.

Locks

Objetivo

Use Lock, ReadWriteLock, and ReentrantLock classes in the java.util.concurrent.locks and java.util.concurrent.atomic packages to support lock-free thread-safe programming on single variables

-
Usar classes dos tipos Lock, ReadWriteLock, e ReentrantLock dos pacotes java.util.concurrent.locks e java.util.concurrent.atomic para suportar programação sem bloqueio e multi-thread em variáveis únicas

As classes e interfaces **Lock**, **ReadWriteLock** e **ReentrantLock** permitem obter diferentes tipos de *Locks* (em tradução livre: bloqueios ou travas). Esses *Locks* são utilizados para que um número limitado de *threads* tenham acesso a mesma variável em um determinado momento, ou para que apenas uma delas possa alterar seu valor.

Nesta seção serão apresentados exemplos utilizando essas classes e interfaces. Assim como em outras seções deste capítulo, os exemplos podem ser grandes quando for necessário a criação de *threads*. Dedique um tempo maior para entendê-los completamente.

Reentrant Lock

1. É possível adquirir um *Lock* utilizando a classe **ReentrantLock**.

src/org/j6toj8/concurrency/locks/Locks_ReentrantLock.java

```
Lock lock = new ReentrantLock();
try {
    lock.lock(); // apenas uma thread obtém o lock por vez
    System.out.println("ABC");
} finally {
    lock.unlock(); // desfaz o lock
}
```

Saída no console

```
ABC
```

Perceba que o *lock* é removido dentro de um bloco **finally**. Isso garante que uma *thread* não irá ficar com um *lock* indeterminadamente.

2. Chamar o método *unlock* sem ter obtido um *lock* anteriormente irá lançar uma exceção.

src/org/j6toj8/concurrency/locks/Locks_UnlockWithoutLock.java

```
Lock lock = new ReentrantLock();
try {
    System.out.println("ABC");
} finally {
    lock.unlock(); // lança exceção, pois não há lock
}
```

Saída no console

```
ABC
Exception in thread "main" java.lang.IllegalMonitorStateException
    at
java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:151)
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSynchro
nizer.java:1261)
    at java.util.concurrent.locks.ReentrantLock.unlock(ReentrantLock.java:457)
    at
org.j6toj8.concurrency.locks.Locks_UnlockWithoutLock.main(Locks_UnlockWithoutLock.j
ava:14)
```

3. É possível tentar obter um *lock* imediatamente utilizando o método **tryLock**.

src/org/j6toj8/concurrency/locks/Locks_TryLock.java

```
Lock lock = new ReentrantLock();
boolean temLock = lock.tryLock();

if (temLock) {
    try {
        System.out.println("ABC");
    } finally {
        lock.unlock(); // desfaz o lock
    }
} else {
    System.out.println("DEF");
}
```

Saída no console

```
ABC
```

4. Também é possível tentar obter um *lock* definindo um tempo de espera máximo.

```
Lock lock = new ReentrantLock();

boolean temLock = false;
try {
    // tenta obter o lock por no máximo 1 segundo
    temLock = lock.tryLock(1, TimeUnit.SECONDS);
} catch (InterruptedException e) {
    System.out.println("Não obteve o Lock");
}

if (temLock) {
    try {
        System.out.println("ABC");
    } finally {
        lock.unlock(); // desfaz o lock
    }
} else {
    System.out.println("DEF");
}
```

Saída no console

```
ABC
```

5. Em um cenário com várias *threads*, é possível que apenas uma delas consiga obter um *lock*.

```
static class Acao implements Runnable {  
  
    private Lock lock;  
  
    public Acao(Lock reentrantLock) {  
        this.lock = reentrantLock;  
    }  
  
    @Override  
    public void run() {  
        if (lock.tryLock()) {  
            try {  
                System.out.println(Thread.currentThread().getName() + ": Conseguiu o  
Lock");  
            } finally {  
                lock.unlock();  
            }  
        }  
    }  
}  
  
public static void main(String[] args) {  
    Lock lock = new ReentrantLock();  
  
    // Criação das threads  
    Thread thread1 = new Thread(new Acao(lock));  
    Thread thread2 = new Thread(new Acao(lock));  
    Thread thread3 = new Thread(new Acao(lock));  
  
    // Execução das threads  
    thread1.start();  
    thread2.start();  
    thread3.start();  
}
```

Saída no console

```
Thread-0: Conseguiu o Lock  
Thread-2: Conseguiu o Lock
```

Nesta execução com 3 *threads*, apenas duas conseguiram obter o *lock* imediatamente e imprimir no console. Porém o resultado é imprevisível. Podem existir execuções onde todas obterão o *lock*, e outras em que apenas uma thread conseguirá.

6. Uma *thread* pode obter mais de um *lock* no mesmo objeto **Lock**, mas deve desfazer o *lock* múltiplas vezes também.

```
Lock lock = new ReentrantLock();
try {
    lock.lock();
    lock.lock();
    System.out.println("ABC");
} finally {
    lock.unlock(); // desfaz o primeiro lock
    lock.unlock(); // desfaz o segundo lock
}
```

Saída no console

```
ABC
```

Como a *thread* chamou `lock` duas vezes, caso ela não houvesse chamado `unlock` duas vezes, outra *thread* não seria capaz de obter o `lock`.

7. É possível garantir uma distribuição mais "justa" de *locks* passando `true` como argumento para o `ReentrantLock`.

```
Lock lock = new ReentrantLock(true); // lock "justo"
try {
    lock.lock();
    System.out.println("ABC");
} finally {
    lock.unlock(); // desfaz o lock
}
```

Ao passar o argumento `true`, quando várias *threads* estiverem esperando pelo mesmo `lock`, ele será dado àquela *thread* que está aguardando a mais tempo.

ReentrantReadWriteLock

1. É possível separar *locks* de leitura e escrita utilizando a classe `ReadWriteLock`. *Locks* de leitura podem ser obtidos por múltiplas *threads*, porém *locks* de escrita não.

```
static class Acao implements Runnable {

    private ReadWriteLock lock;

    public Acao(ReadWriteLock reentrantLock) {
        this.lock = reentrantLock;
    }

    @Override
    public void run() {
        Lock readLock = lock.readLock();
        if (readLock.tryLock()) {
            try {
                System.out.println(Thread.currentThread().getName() + ": Consegui o Lock
de leitura");
            } finally {
                readLock.unlock();
            }
        }

        Lock writeLock = lock.writeLock();
        if (writeLock.tryLock()) {
            try {
                System.out.println(Thread.currentThread().getName() + ": Consegui o Lock
de escrita");
            } finally {
                writeLock.unlock();
            }
        }
    }
}

public static void main(String[] args) {
    ReadWriteLock lock = new ReentrantReadWriteLock();

    // Criação das threads
    Thread thread1 = new Thread(new Acao(lock));
    Thread thread2 = new Thread(new Acao(lock));
    Thread thread3 = new Thread(new Acao(lock));

    // Execução das threads
    thread1.start();
    thread2.start();
    thread3.start();
}
```

Saída no console

```
Thread-0: Conseguiu o Lock de leitura
Thread-2: Conseguiu o Lock de leitura
Thread-1: Conseguiu o Lock de leitura
Thread-1: Conseguiu o Lock de escrita
```

Perceba que todas as *threads* conseguiram obter o *lock* de leitura, porém apenas uma conseguiu obter o *lock* de escrita.

2. Se uma *thread* já possuir o *lock* de escrita, outras não conseguirão obter nem mesmo o *lock* de leitura.

```
static class Acao implements Runnable {

    private ReadWriteLock lock;

    public Acao(ReadWriteLock reentrantLock) {
        this.lock = reentrantLock;
    }

    @Override
    public void run() {
        Lock writeLock = lock.writeLock();
        if (writeLock.tryLock()) {
            try {
                System.out.println(Thread.currentThread().getName() + ": Consegui o Lock
de escrita");
            } finally {
                writeLock.unlock();
            }
        }

        Lock readLock = lock.readLock();
        if (readLock.tryLock()) {
            try {
                System.out.println(Thread.currentThread().getName() + ": Consegui o Lock
de leitura");
            } finally {
                readLock.unlock();
            }
        }
    }
}

public static void main(String[] args) {
    ReadWriteLock lock = new ReentrantReadWriteLock();

    // Criação das threads
    Thread thread1 = new Thread(new Acao(lock));
    Thread thread2 = new Thread(new Acao(lock));
    Thread thread3 = new Thread(new Acao(lock));

    // Execução das threads
    thread1.start();
    thread2.start();
    thread3.start();
}
```

Saída no console

```
Thread-0: Conseguiu o Lock de escrita  
Thread-0: Conseguiu o Lock de leitura
```

Perceba que neste exemplo o *lock* de escrita está sendo obtido **antes** do de leitura, de tal forma que apenas a primeira *thread* que foi executada conseguiu obter os dois *locks*.

- Applying Locks

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 607). Wiley. Edição do Kindle.

- [Guide to java.util.concurrent.Locks](#).
- [Package java.util.concurrent.locks](#). Java Platform SE 8.

Executando tarefas com ExecutorService

Objetivo

Use Executor, ExecutorService, Executors, Callable, and Future to execute tasks using thread pools

-
Usar Executor, ExecutorService, Executors, Callable, e Future para executar tarefas usando um conjunto de threads

A classe `ExecutorService` é utilizada para a criação e execução de *threads*. Geralmente é utilizado um conjunto, ou repositório, de *threads*. A classe `Executors` contém métodos utilitários para a criação desses executores. Serão apresentados os principais usos de `ExecutorService`.

1. É possível executar uma tarefa em uma *thread* separada utilizando o método `newSingleThreadExecutor`.

src/org/j6toj8/concurrency/executetasks/Tasks_SingleThread.java

```
ExecutorService executor = null;  
try {  
    executor = Executors.newSingleThreadExecutor(); // executor com uma única thread  
    executor.execute(() -> System.out.println("Thread do Executor: " +  
        Thread.currentThread().getName()));  
    System.out.println("Thread Principal: " + Thread.currentThread().getName());  
} finally {  
    if (executor != null) {  
        executor.shutdown();  
    }  
}
```

Saída no console

```
Thread Principal: main
Thread do Executor: pool-1-thread-1
```

O método `execute` recebe uma instância de `Runnable`, que nestes exemplos serão criadas na forma de uma expressão lambda.

Perceba que a *thread* utilizada pelo executor é diferente da *thread* principal.

Perceba também que é boa prática chamar o método `shutdown` dentro de um bloco `finally`, buscando não deixar *threads* pendentes. Fique atento, pois chamar o método `shutdown` **não** garante que as *threads* foram finalizadas.

Em um programa real, talvez você não queira que a tarefa seja finalizada logo após a sua criação, como nos exemplos aqui apresentados. Logo, pode ser necessário invocar o `shutdown` em algum ponto do sistema, ou quando a aplicação for encerrada. Apenas lembre-se de invocar o método `shutdown` para que sua aplicação possa ser encerrada corretamente, caso contrário o processo poderá ficar travado impedindo que o programa encerre corretamente..

2. É possível executar várias tarefas em uma mesma *thread* separada utilizando o mesmo método `newSingleThreadExecutor`.

`src/org/j6toj8/concurrency/executetasks/Tasks_SingleThreadManyTasks.java`

```
ExecutorService executor = null;
try {
    executor = Executors.newSingleThreadExecutor(); // executor com uma única thread
    executor.execute(() -> System.out.println("Tarefa 1 - Thread do Executor: " +
        Thread.currentThread().getName()));
    executor.execute(() -> System.out.println("Tarefa 2 - Thread do Executor: " +
        Thread.currentThread().getName()));
    executor.execute(() -> System.out.println("Tarefa 3 - Thread do Executor: " +
        Thread.currentThread().getName()));
    System.out.println("Thread Principal: " + Thread.currentThread().getName());
} finally {
    if (executor != null) {
        executor.shutdown();
    }
}
```

Saída no console

```
Tarefa 1 - Thread do Executor: pool-1-thread-1
Tarefa 2 - Thread do Executor: pool-1-thread-1
Thread Principal: main
Tarefa 3 - Thread do Executor: pool-1-thread-1
```

Perceba que todas as tarefas foram executadas pela mesma *thread*, que é diferente e paralela à

thread principal.

3. É possível chamar `shutdownNow` para **tentar** encerrar todas as *threads* imediatamente.

src/org/j6toj8/concurrency/executetasks/Tasks_ShutdownNow.java

```
ExecutorService executor = null;
try {
    executor = Executors.newSingleThreadExecutor();
    executor.execute(() -> System.out.println("Thread do Executor: " +
Thread.currentThread().getName()));
    System.out.println("Thread Principal: " + Thread.currentThread().getName());
} finally {
    if (executor != null) {
        executor.shutdownNow(); // TENTA encerrar todas as threads imediatamente
    }
}
```

Saída no console

```
Thread Principal: main
Thread do Executor: pool-1-thread-1
```

4. É possível utilizar o método `submit` para ter informações sobre a tarefa que está sendo executada.

src/org/j6toj8/concurrency/executetasks/Tasks_SingleThreadSubmit.java

```
ExecutorService executor = null;
try {
    executor = Executors.newSingleThreadExecutor();
    Future<?> tarefa = executor.submit(() -> System.out.println("Thread do Executor:
" + Thread.currentThread().getName()));

    System.out.println("Tarefa já finalizada? " + tarefa.isDone());
    System.out.println("Tarefa já finalizada? " + tarefa.isDone());
    System.out.println("Tarefa já finalizada? " + tarefa.isDone());
    System.out.println("Tarefa já finalizada? " + tarefa.isDone());
} finally {
    if (executor != null) {
        executor.shutdown();
    }
}
```

Saída no console

```
Tarefa já finalizada? false
Thread do Executor: pool-1-thread-1
Tarefa já finalizada? false
Tarefa já finalizada? true
Tarefa já finalizada? true
```

A instância de `Future` representa um valor que será retornado no futuro. Nesse caso é um valor que representa a própria tarefa submetida.

Veja que a tarefa inicialmente ainda não havia sido finalizada, mas nas duas últimas impressões no console já havia sido.

O resultado desse exemplo irá ser diferente para cada execução, tendo em vista a utilização de mais de uma *thread*: a principal e uma paralela.

5. Também é possível ver outras informações sobre a tarefa ou tentar cancelar sua execução.

`src/org/j6toj8/concurrency/executetasks/Tasks_SingleThreadFuture.java`

```
ExecutorService executor = null;
try {
    executor = Executors.newSingleThreadExecutor();
    Future<?> tarefa = executor.submit(() -> System.out.println("Tarefa
executando"));

    // verifica se a tarefa está finalizada
    System.out.println("Tarefa já finalizada? " + tarefa.isDone());

    // tenta cancelar a tarefa
    System.out.println("Tentando cancelar a tarefa. Conseguiu? " + tarefa.cancel(
true));

    // verifica se a tarefa foi cancelada
    System.out.println("Tarefa foi cancelada? " + tarefa.isCancelled());
} finally {
    if (executor != null) {
        executor.shutdown();
    }
}
```

Saída no console

```
Tarefa já finalizada? false
Tentando cancelar a tarefa. Conseguiu? true
Tarefa executando
Tarefa foi cancelada? true
```

Perceba como foi possível cancelar a tarefa mesmo após ela já ter impresso no console.

O resultado desse exemplo também será diferente para cada execução.

6. É possível retornar um valor da tarefa utilizando o método `submit` que recebe uma instância de `Callable`.

src/org/j6toj8/concurrency/executetasks/Tasks_SingleThreadCallable.java

```
ExecutorService executor = null;
try {
    executor = Executors.newSingleThreadExecutor();
    Future<?> retornoDaTarefa = executor.submit(() -> "String que será retornada");

    // O .get() abaixo irá esperar a tarefa finalizar para pegar seu retorno
    System.out.println("Retorno da tarefa: " + retornoDaTarefa.get());
} catch (InterruptedException | ExecutionException e) {
    System.out.println("Execução interrompida.");
} finally {
    if (executor != null) {
        executor.shutdown();
    }
}
```

Saída no console

```
Retorno da tarefa: String que será retornada
```

Nesse caso a instância de `Future` representa o valor retornado pela tarefa, uma `String`.

É necessário o bloco `catch` para capturar as exceções que podem ser lançadas pelo método `get` da classe `Future`.

7. É possível passar uma lista de tarefas para serem executadas utilizando o método `invokeAll`.

```
List<Callable<String>> tarefas = new ArrayList<Callable<String>>();  
tarefas.add(() -> "Tarefa 1 executada na thread " + Thread.currentThread().  
getName());  
tarefas.add(() -> "Tarefa 2 executada na thread " + Thread.currentThread().  
getName());  
tarefas.add(() -> "Tarefa 3 executada na thread " + Thread.currentThread().  
getName());  
  
ExecutorService executor = null;  
try {  
    executor = Executors.newSingleThreadExecutor();  
  
    // invokeAll devolve todos os retornos das tarefas executadas em uma lista  
    List<Future<String>> retornos = executor.invokeAll(tarefas);  
  
    for (Future<String> retorno : retornos) {  
        System.out.println("Retorno da tarefa: " + retorno.get());  
    }  
} catch (InterruptedException | ExecutionException e) {  
    System.out.println("Execução interrompida.");  
} finally {  
    if (executor != null) {  
        executor.shutdown();  
    }  
}
```

Saída no console

```
Retorno da tarefa: Tarefa 1 executando na thread pool-1-thread-1  
Retorno da tarefa: Tarefa 2 executando na thread pool-1-thread-1  
Retorno da tarefa: Tarefa 3 executando na thread pool-1-thread-1
```

8. É possível passar uma lista de tarefas onde apenas uma será concluída utilizando o método `invokeAny`.

```
List<Callable<String>> tarefas = new ArrayList<Callable<String>>();  
tarefas.add(() -> "Tarefa 1 executada na thread " + Thread.currentThread().  
getName());  
tarefas.add(() -> "Tarefa 2 executada na thread " + Thread.currentThread().  
getName());  
tarefas.add(() -> "Tarefa 3 executada na thread " + Thread.currentThread().  
getName());  
  
ExecutorService executor = null;  
try {  
    executor = Executors.newSingleThreadExecutor();  
  
    // invokeAny devolve apenas uma das tarefas que finalizou e interrompe as outras  
    String retorno = executor.invokeAny(tarefas);  
    System.out.println("Retorno da tarefa: " + retorno);  
  
} catch (InterruptedException | ExecutionException e) {  
    System.out.println("Execução interrompida.");  
} finally {  
    if (executor != null) {  
        executor.shutdown();  
    }  
}
```

Saída no console

```
Retorno da tarefa: Tarefa 1 executada na thread pool-1-thread-1
```

As outras tarefas são interrompidas ou, como neste caso só temos uma *thread*, nem são executadas.

Em um exemplo com inúmeras *threads*, é possível que qualquer uma das 3 tarefas finalize primeiro e interrompa as outras duas.

9. **Runnable** não possui retorno nem pode lançar exceção checada, enquanto **Callable** possui retorno e pode lançar exceção checada. É importante saber diferenciar qual versão do método **submit** está sendo chamado.

```
ExecutorService executor = null;
try {
    executor = Executors.newSingleThreadExecutor();

    // tarefa sem retorno, instância de Runnable
    executor.submit(() -> System.out.println("Runnable"));

    // tarefa com retorno, instância de Callable
    executor.submit(() -> "Callable");

    // tarefa que lança uma Exception deve ser Callable, logo deve ter retorno
    executor.submit(() -> {Thread.sleep(1); return "Callable";});

    // tarefa que lança uma Exception, mas não declara retorno
    // NÃO COMPILA pois é interpretada como Runnable
    executor.submit(() -> Thread.sleep(1));

} finally {
    if (executor != null) {
        executor.shutdown();
    }
}
```

Lembre-se que `Thread.sleep()` lança uma exceção checada.

10. É possível esperar as tarefas finalizarem sua execução por um tempo específico com o método `awaitTermination`.

```
ExecutorService executor = null;
try {
    executor = Executors.newSingleThreadExecutor();

    executor.execute(() -> System.out.println("Tarefa 1 - Thread do Executor: " +
Thread.currentThread().getName()));
    executor.execute(() -> System.out.println("Tarefa 2 - Thread do Executor: " +
Thread.currentThread().getName()));
    executor.execute(() -> System.out.println("Tarefa 3 - Thread do Executor: " +
Thread.currentThread().getName()));
} finally {
    if (executor != null) {
        executor.shutdown();
    }
}

if (executor != null) {
    try {
        System.out.println("Tarefas finalizadas? " + executor.isTerminated());
        executor.awaitTermination(1, TimeUnit.SECONDS);
        System.out.println("Tarefas finalizadas? " + executor.isTerminated());
    } catch (InterruptedException e) {
        System.out.println("Erro de interrupção.");
    }
}
```

Saída no console

```
Tarefa 1 - Thread do Executor: pool-1-thread-1
Tarefa 2 - Thread do Executor: pool-1-thread-1
Tarefas finalizadas? false
Tarefa 3 - Thread do Executor: pool-1-thread-1
Tarefas finalizadas? true
```

Caso as tarefas não tenham terminado depois de 1 segundo, a execução teria continuado normalmente. Não há exceção neste caso. A única diferença seria a saída no console, pois nas duas vezes seria impresso **false**.

Tarefas agendadas

1. É possível agendar tarefas com as classes de **ScheduledExecutorService**.

```
ScheduledExecutorService executor = null;
try {
    executor = Executors.newSingleThreadScheduledExecutor(); // executor de
    agendamento com uma única thread
    System.out.println("Agora: " + LocalTime.now()); // imprime a hora atual

    executor.schedule(() -> System.out.println("Execução: " + LocalTime.now()), 3,
    TimeUnit.SECONDS);
} finally {
    if (executor != null) {
        executor.shutdown();
    }
}
```

Saída no console

```
Agora: 16:05:25.978
Execução: 16:05:28.984
```

Perceba que a tarefa foi executada aproximadamente 3 segundos após o agendamento.

O método `schedule` utilizado neste exemplo recebe um *Runnable*, por isso não há retorno na expressão lambda.

Ao agendar uma tarefa, o programa só encerra após a execução, cancelamento ou interrupção da tarefa.

2. É possível agendar uma tarefa com retorno passando uma instância de *Callable*.

```
ScheduledExecutorService executor = null;
try {
    executor = Executors.newSingleThreadScheduledExecutor(); // executor de
    agendamento com uma única thread
    System.out.println("Antes do agendamento: " + LocalTime.now());
    ScheduledFuture<String> retorno = executor.schedule(() -> "Execução: " +
    LocalTime.now(), 3, TimeUnit.SECONDS);
    System.out.println("Depois do agendamento: " + LocalTime.now());

    System.out.println(retorno.get()); // fica parado aqui esperando o retorno
    System.out.println("Depois da execução: " + LocalTime.now());
} catch (InterruptedException | ExecutionException e) {
    System.out.println("Erro ao fazer .get()");
} finally {
    if (executor != null) {
        executor.shutdown();
    }
}
```

Saída no console

```
Antes do agendamento: 16:10:48.898
Depois do agendamento: 16:10:48.903
Execução: 16:10:51.904
Depois da execução: 16:10:51.904
```

3. É possível agendar uma tarefa para ser executada a cada X tempo usando o método `scheduleAtFixedRate`.

```
public static void main(String[] args) {
    ScheduledExecutorService executor = null;
    try {
        executor = Executors.newSingleThreadScheduledExecutor(); // executor de
        agendamento com uma única thread
        System.out.println("Antes do agendamento: " + LocalTime.now()); // imprime a
        hora atual
        executor.scheduleAtFixedRate(() -> System.out.println("Execução: " +
        LocalTime.now()), 3, 1, TimeUnit.SECONDS);
        System.out.println("Após do agendamento: " + LocalTime.now()); // imprime a
        hora atual
        sleep(); // aguarda um tempo para ser possível enxergar as execuções
        System.out.println("Após o sleep de 10 segundos: " + LocalTime.now()); // //
        imprime a hora atual
    } finally {
        if (executor != null) {
            System.out.println("Invocando shutdown no executor.");
            executor.shutdown();
        }
    }
}

private static void sleep() {
    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Saída no console

```
Antes do agendamento: 16:16:54.511
Após do agendamento: 16:16:54.517
Execução: 16:16:57.517
Execução: 16:16:58.517
Execução: 16:16:59.517
Execução: 16:17:00.517
Execução: 16:17:01.517
Execução: 16:17:02.517
Execução: 16:17:03.517
Execução: 16:17:04.517
Após o sleep de 10 segundos: 16:17:04.517
Invocando shutdown no executor.
```

Neste exemplo a execução demora 3 segundos para começar, e é repetida a cada 1 segundo, até o `shutdown` ser chamado. Por isso existe um `sleep` no final do bloco `try`, para garantir que enxergariamos a execução da tarefa por algumas vezes antes de invocar o `shutdown`.

4. É possível agendar uma tarefa para ser executada X tempo após o término da tarefa anterior usando o método `scheduleWithFixedDelay`.

`src/org/j6toj8/concurrency/executetasks/Schedule_SingleThreadFixedDelay.java`

```
public static void main(String[] args) {
    ScheduledExecutorService executor = null;
    try {
        executor = Executors.newSingleThreadScheduledExecutor(); // executor de
        agendamento com uma única thread
        System.out.println("Antes do agendamento: " + LocalTime.now()); // imprime a
        hora atual
        executor.scheduleWithFixedDelay(() -> System.out.println("Execução: " +
        LocalTime.now()), 3, 1, TimeUnit.SECONDS);
        System.out.println("Após do agendamento: " + LocalTime.now()); // imprime a
        hora atual
        sleep(); // aguarda um tempo para ser possível enxergar as execuções
        System.out.println("Após o sleep de 10 segundos: " + LocalTime.now()); // //
        imprime a hora atual
    } finally {
        if (executor != null) {
            System.out.println("Invocando shutdown no executor.");
            executor.shutdown();
        }
    }
}

private static void sleep() {
    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Saída no console

```
Antes do agendamento: 16:20:46.717
Após do agendamento: 16:20:46.727
Execução: 16:20:49.728
Execução: 16:20:50.729
Execução: 16:20:51.729
Execução: 16:20:52.730
Execução: 16:20:53.731
Execução: 16:20:54.732
Execução: 16:20:55.732
Após o sleep de 10 segundos: 16:20:56.728
Invocando shutdown no executor.
```

Este exemplo é muito parecido com o anterior, porém há uma diferença importante: a nova

tarefa só começa a executar 1 segundo depois do **término** da anterior. No exemplo acima a tarefa era executada a **cada** 1 segundo, independente do término da anterior. Perceba as diferenças nos milissegundos.

Múltiplas Threads

Todos os exemplos até agora utilizaram apenas uma *thread* adicional. Em todos eles, seria possível utilizar construtores de **Executors** que fornecem mais de uma *thread*.

1. É possível criar um Executor que cria *threads* conforme o necessário, e as reutiliza quando possível, utilizando o método **newCachedThreadPool**.

src/org/j6toj8/concurrency/executetasks/TasksMulti_CachedThreadPool.java

```
ExecutorService executor = null;
try {
    executor = Executors.newCachedThreadPool(); // executor com cache de threads
    executor.execute(() -> System.out.println("Tarefa 1 - Thread do Executor: " +
        Thread.currentThread().getName()));
    executor.execute(() -> System.out.println("Tarefa 2 - Thread do Executor: " +
        Thread.currentThread().getName()));
    executor.execute(() -> System.out.println("Tarefa 3 - Thread do Executor: " +
        Thread.currentThread().getName()));
    executor.execute(() -> System.out.println("Tarefa 4 - Thread do Executor: " +
        Thread.currentThread().getName()));
    executor.execute(() -> System.out.println("Tarefa 5 - Thread do Executor: " +
        Thread.currentThread().getName()));
} finally {
    if (executor != null) {
        executor.shutdown();
    }
}
```

Saída no console

```
Tarefa 1 - Thread do Executor: pool-1-thread-1
Tarefa 2 - Thread do Executor: pool-1-thread-2
Tarefa 3 - Thread do Executor: pool-1-thread-3
Tarefa 4 - Thread do Executor: pool-1-thread-3
Tarefa 5 - Thread do Executor: pool-1-thread-3
```

Perceba que foram criadas 3 *threads*, e a terceira foi utilizada 3 vezes.

2. É possível criar um Executor com um número delimitado de *threads* utilizando o método **newFixedThreadPool**.

```
ExecutorService executor = null;
try {
    executor = Executors.newFixedThreadPool(2); // executor com duas threads
    executor.execute(() -> System.out.println("Tarefa 1 - Thread do Executor: " +
Thread.currentThread().getName()));
    executor.execute(() -> System.out.println("Tarefa 2 - Thread do Executor: " +
Thread.currentThread().getName()));
    executor.execute(() -> System.out.println("Tarefa 3 - Thread do Executor: " +
Thread.currentThread().getName()));
    executor.execute(() -> System.out.println("Tarefa 4 - Thread do Executor: " +
Thread.currentThread().getName()));
    executor.execute(() -> System.out.println("Tarefa 5 - Thread do Executor: " +
Thread.currentThread().getName()));
} finally {
    if (executor != null) {
        executor.shutdown();
    }
}
```

Saída no console

```
Tarefa 1 - Thread do Executor: pool-1-thread-1
Tarefa 2 - Thread do Executor: pool-1-thread-2
Tarefa 3 - Thread do Executor: pool-1-thread-1
Tarefa 4 - Thread do Executor: pool-1-thread-2
Tarefa 5 - Thread do Executor: pool-1-thread-1
```

3. É possível criar um Executor de agendamento com um número delimitado de *threads* utilizando o método `newScheduledThreadPool`.

```
ScheduledExecutorService executor = null;
try {
    executor = Executors.newScheduledThreadPool(2); // executor de agendamento com
duas threads
    System.out.println("Agora: " + LocalTime.now()); // imprime a hora atual

    executor.schedule(() -> System.out.println("Execução 1: " + Thread.
currentThread().getName() + " - " + LocalTime.now()), 3, TimeUnit.SECONDS);
    executor.schedule(() -> System.out.println("Execução 2: " + Thread.
currentThread().getName() + " - " + LocalTime.now()), 3, TimeUnit.SECONDS);
    executor.schedule(() -> System.out.println("Execução 3: " + Thread.
currentThread().getName() + " - " + LocalTime.now()), 3, TimeUnit.SECONDS);
    executor.schedule(() -> System.out.println("Execução 4: " + Thread.
currentThread().getName() + " - " + LocalTime.now()), 3, TimeUnit.SECONDS);
    executor.schedule(() -> System.out.println("Execução 5: " + Thread.
currentThread().getName() + " - " + LocalTime.now()), 3, TimeUnit.SECONDS);
} finally {
    if (executor != null) {
        executor.shutdown();
    }
}
```

Saída no console

```
Agora: 16:33:36.825
Execução 1: pool-1-thread-1 - 16:33:39.834
Execução 2: pool-1-thread-2 - 16:33:39.836
Execução 3: pool-1-thread-1 - 16:33:39.837
Execução 4: pool-1-thread-1 - 16:33:39.838
Execução 5: pool-1-thread-2 - 16:33:39.838
```

Duas *threads* e *delay* de 3 segundos em todos os agendamentos.

- Creating Threads with the ExecutorService

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 335). Wiley. Edição do Kindle.

- [Introduction to Thread Pools in Java.](#)
- [A Guide to the Java ExecutorService.](#)
- <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html>[Interface ExecutorService.] Java Platform SE 7.

Framework Fork/Join

Objetivo

Use the parallel Fork/Join Framework

-

Usar o Framework Fork/Join de paralelismo

Com o framework de Fork/Join é possível dividir uma tarefa grande em pedaços menores e executá-los em paralelo. A utilização do framework é simples. A criação de tarefas quebradas em várias partes pode ser um pouco mais complexa.

Uma tarefa geralmente segue um comportamento padrão:

- Recebe um valor;
- Avalia se o valor é pequeno o suficiente para executar a tarefa com ele;
 - Caso positivo, executa a tarefa com aquele valor;
 - Caso negativo, quebra em uma ou mais partes e cria novas tarefas menores;
- Segue executando de forma recursiva.

Serão apresentados exemplos utilizando a classe `RecursiveAction`, que não retorna valores, e por isso sua implementação é mais simples. E utilizando a classe `RecursiveTask`, que retorna valores e por isso sua implementação é um pouco mais complexa.

Os exemplos são grandes, pois é necessário implementar toda a classe `RecursiveAction` ou `RecursiveTask` para o exemplo funcionar. Por isso, os passos serão descritos com comentários no código, explicando o funcionamento de cada chamada realizada ao framework.

1. É possível implementar uma `RecursiveAction` que divide uma tarefa grande em partes menores.

`src/org/j6toj8/concurrency/forkjoin/ForkJoin_RecursiveAction.java`

```
// Classe que representa a tarefa que será executada
static class ImpressaoDeStrings extends RecursiveAction {

    private String stringParaImprimir; // String que será impressa

    public ImpressaoDeStrings(String stringParaImprimir) {
        this.stringParaImprimir = stringParaImprimir;
    }

    @Override
    protected void compute() {
        if (stringParaImprimir.length() < 10) {
            // se a String tiver menos de 10 caracteres, será impressa
            System.out.println(Thread.currentThread().getName() + " - " +
                stringParaImprimir);
        } else {
    }}
```

```

        // caso contrário, são criadas duas novas tarefas, uma com a primeira metade
        // da String
        // e outra com a segunda metade da String
        List<ImpressaoDeStrings> novasTarefas = divideEmDuasTarefas();

        // Invoca a execução das duas tarefas criadas
        ForkJoinTask.invokeAll(novasTarefas);
    }
}

private List<ImpressaoDeStrings> divideEmDuasTarefas() {
    // esse método divide a String em duas partes e cria duas novas tarefas
    // cada uma das tarefas recebe uma parte da String

    int tamanhoDaString = stringParaImprimir.length();
    int meioDaString = tamanhoDaString / 2;

    String primeiraMetade = stringParaImprimir.substring(0, meioDaString);
    String segundaMetade = stringParaImprimir.substring(meioDaString);

    List<ImpressaoDeStrings> acoes = new ArrayList<ImpressaoDeStrings>();
    acoes.add(new ImpressaoDeStrings(primeiraMetade));
    acoes.add(new ImpressaoDeStrings(segundaMetade));
    return acoes;
}

public static void main(String[] args) {
    // string que queremos imprimir
    String stringParaImprimir = "ABCDEFGHIJKLMNPQRSTUVWXYZ";

    // tarefa principal que será executada
    ImpressaoDeStrings tarefa = new ImpressaoDeStrings(stringParaImprimir);

    // criação do ForkJoinPool e execução da tarefa
    ForkJoinPool forkJoinPool = new ForkJoinPool();
    forkJoinPool.invoke(tarefa);
}

```

Saída no console

```

ForkJoinPool-1-worker-1 - ABCDEF
ForkJoinPool-1-worker-4 - TUVWXYZ
ForkJoinPool-1-worker-2 - NOPQRS
ForkJoinPool-1-worker-3 - GHIJKLMNOP

```

- Esse exemplo cria uma tarefa do tipo `RecursiveAction` chamada `ImpressaoDeStrings`.
- Essa `RecursiveAction` recebe uma `String` para imprimir no Console.

- c. No método `compute`, a tarefa decide se irá imprimir diretamente no console, ou se irá dividir esse trabalho em duas partes: caso a `String` tenha menos que 10 caracteres, ela imprime diretamente no Console; caso contrário, divide o trabalho em duas novas tarefas.

Perceba que cada impressão no console é realizada em uma *thread* diferente, ficando claro que o trabalho está sendo compartilhado por múltiplas *threads*.

Perceba também que o processamento da `String` não retorna nenhum valor, e por isso foi utilizada a classe `RecursiveAction`.

2. Caso seja necessário retornar um valor, é possível implementar uma `RecursiveTask` que divide uma tarefa grande em partes menores.

`src/org/j6toj8/concurrency/forkjoin/ForkJoin_RecursiveTask.java`

```
// Classe que representa a tarefa que será executada
static class ImpressaoDeStrings extends RecursiveTask<Integer> {

    private String stringParaImprimir; // String que será impressa

    public ImpressaoDeStrings(String stringParaImprimir) {
        this.stringParaImprimir = stringParaImprimir;
    }

    @Override
    protected Integer compute() {
        if (stringParaImprimir.length() < 10) {
            // se a String tiver menos de 10 caracteres, será impressa
            System.out.println(Thread.currentThread().getName() + " - " +
stringParaImprimir);

            // retornamos a quantidade de caracteres impressos
            return stringParaImprimir.length();
        } else {
            // caso contrário, são criadas duas novas tarefas, uma com a primeira metade
            // da String
            // e outra com a segunda metade da String
            List<ImpressaoDeStrings> novasTarefas = divideEmDuasTarefas();
            ImpressaoDeStrings tarefa1 = novasTarefas.get(0);
            ImpressaoDeStrings tarefa2 = novasTarefas.get(1);

            // 'fork' irá enviar a tarefa1 para ser executada em uma nova thread
            ForkJoinTask<Integer> primeiraTarefa = tarefa1.fork();

            // 'compute' irá executar a tarefa2 nessa mesma thread
            Integer resultadoDaTarefa2 = tarefa2.compute();

            // 'join' irá pegar o resultado da tarefa1 que estava sendo executada em
            // outra thread
            Integer resultadoDaTarefa1 = primeiraTarefa.join();
        }
    }
}
```

```

        // retornamos a soma dos resultados, pois é a quantidade de caracteres
        impressos
        return resultadoDaTarefa2 + resultadoDaTarefa1;
    }
}

private List<ImpressaoDeStrings> divideEmDuasTarefas() {
    // esse método divide a String em duas partes e cria duas novas tarefas
    // cada uma das tarefas recebe uma parte da String

    int tamanhoDaString = stringParaImprimir.length();
    int meioDaString = tamanhoDaString / 2;

    String primeiraMetade = stringParaImprimir.substring(0, meioDaString);
    String segundaMetade = stringParaImprimir.substring(meioDaString);

    List<ImpressaoDeStrings> acoes = new ArrayList<ImpressaoDeStrings>();
    acoes.add(new ImpressaoDeStrings(primeiraMetade));
    acoes.add(new ImpressaoDeStrings(segundaMetade));
    return acoes;
}

public static void main(String[] args) {
    // string que queremos imprimir
    String stringParaImprimir = "ABCDEFGHIJKLMNPQRSTUVWXYZ";

    // tarefa principal que será executada
    ImpressaoDeStrings tarefa = new ImpressaoDeStrings(stringParaImprimir);

    // criação do ForkJoinPool e execução da tarefa
    ForkJoinPool forkJoinPool = new ForkJoinPool();
    Integer resultado = forkJoinPool.invoke(tarefa);
    System.out.println("Resultado da execução: " + resultado);
}

```

Saída no console

```

ForkJoinPool-1-worker-3 - NOPQRS
ForkJoinPool-1-worker-2 - GHIJKLM
ForkJoinPool-1-worker-4 - ABCDEF
ForkJoinPool-1-worker-1 - TUVWXYZ
Resultado da execução: 26

```

- Esse exemplo cria uma tarefa do tipo `RecursiveTask` chamada `ImpressaoDeStrings`.
- Essa `RecursiveTask` recebe uma `String` para imprimir no Console.
- No método `compute`, a tarefa decide se irá imprimir diretamente no console, ou se irá dividir esse trabalho em duas partes: caso a `String` tenha menos que 10 caracteres, ela imprime

diretamente no Console; caso contrário, divide o trabalho em duas novas tarefas.

Perceba também que o processamento da String retorna quantos caracteres foram impressos, e por isso foi utilizada a classe **RecursiveTask**.

- Managing Concurrent Processes

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 377). Wiley. Edição do Kindle.

- [Guide to the Fork/Join Framework in Java](#).
- [Fork/Join](#). The Java™ Tutorials.

Java File I/O (NIO.2)

Paths

Objetivo

Operate on file and directory paths by using the Paths class

-

Operar em arquivos e diretórios usando a classe Paths

As classes `Path` e `Paths` são novidades do Java 7.

A classe `Path` representa um arquivo ou um diretório no sistema de arquivos, e a maioria das suas operações não altera diretamente arquivos ou diretórios.

A classe `Paths` contém métodos estáticos para a criação de `Path`.

Para que os exemplos executem independente do sistema, será utilizado o diretório do usuário, que no Java está disponível em uma propriedade da JVM chamada `user.home`.

1. Existem inúmeras formas de obter uma instância de `Path`.

```
// diretório padrão do usuário
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

// path absoluto
Path path1 = Paths.get("/home/rinaldo");
System.out.println("Path 1: " + path1);

// path absoluto dividido em strings
Path path2 = Paths.get("/", "home", "rinaldo");
System.out.println("Path 2: " + path2);

// path absoluto a partir do userHome
Path path3 = Paths.get(userHome);
System.out.println("Path 3: " + path3);

// path absoluto para um arquivo
Path path4 = Paths.get("/home/rinaldo/arquivos/arquivo.txt");
System.out.println("Path 4: " + path4);

// path absoluto para um arquivo a partir do userHome
Path path5 = Paths.get(userHome, "arquivos", "arquivo.txt");
System.out.println("Path 5: " + path5);

// path absoluto em um sistema windows
Path path6 = Paths.get("C:/users/rinaldo");
System.out.println("Path 6: " + path6);

// path absoluto windows divido em strings
Path path7 = Paths.get("C:", "users", "rinaldo");
System.out.println("Path 7: " + path7);

// path relativo
Path path8 = Paths.get("rinaldo");
System.out.println("Path 8: " + path8);

// path a partir de uma URI
Path path9 = Paths.get(URI.create("file:///home/rinaldo/arquivos/arquivo.txt"));
System.out.println("Path 9: " + path9);

// path sem utilizar a classe Paths - produz o mesmo resultado
Path path10 = FileSystems.getDefault().getPath("/home/rinaldo");
System.out.println("Path 10: " + path10);
```

Saída no console

```
User home: /home/rinaldo
Path 1: /home/rinaldo
Path 2: /home/rinaldo
Path 3: /home/rinaldo
Path 4: /home/rinaldo/arquivo.txt
Path 5: /home/rinaldo/arquivo.txt
Path 6: C:/users/rinaldo
Path 7: C:/users/rinaldo
Path 8: rinaldo
Path 9: /home/rinaldo/arquivo.txt
Path 10: /home/rinaldo
```

2. É possível criar uma instância de **Path** apontando para um diretório ou arquivo que não existe.

src/org/j6toj8/fileio/paths/Paths_CreationDoesntExists.java

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

Path path = Paths.get(userHome, "arquivoQueNaoExiste.txt");
System.out.println("Path: " + path);
```

Saída no console

```
User home: /home/rinaldo
Path: /home/rinaldo/arquivoQueNaoExiste.txt
```

3. É possível converter um **Path** para um **File**.

src/org/j6toj8/fileio/paths/Paths_ToFile.java

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

Path path = Paths.get(userHome, "arquivoQueNaoExiste.txt");
System.out.println("Path: " + path);

File file = path.toFile();
System.out.println("File: " + file);
```

Saída no console

```
User home: /home/rinaldo
Path: /home/rinaldo/arquivoQueNaoExiste.txt
File: /home/rinaldo/arquivoQueNaoExiste.txt
```

4. Existem inúmeros método no **Path** para recuperar informações a seu respeito.

src/org/j6toj8/fileio/paths/Paths_Information.java

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome + "\n");

Path path1 = Paths.get(userHome, "arquivos", "arquivo.txt"); // absoluto
System.out.println("Representação em String: " + path1.toString());
System.out.println("Nome do Arquivo: " + path1.getFileName());
System.out.println("Diretório Superior: " + path1.getParent());
System.out.println("Diretório Raiz: " + path1.getRoot());
System.out.println("É absoluto?: " + path1.isAbsolute());

System.out.println();

Path path2 = Paths.get("home", "rinaldo", "arquivos"); // relativo
System.out.println("Representação em String: " + path2.toString());
System.out.println("Nome do Arquivo: " + path2.getFileName());
System.out.println("Diretório Superior: " + path2.getParent());
System.out.println("Diretório Raiz: " + path2.getRoot());
System.out.println("É absoluto?: " + path2.isAbsolute());
```

Saída no console

```
User home: /home/rinaldo

Representação em String: /home/rinaldo/arquivos/arquivo.txt
Nome do Arquivo: arquivo.txt
Diretório Superior: /home/rinaldo/arquivos
Diretório Raiz: /
É absoluto? true

Representação em String: home/rinaldo/arquivos
Nome do Arquivo: arquivos
Diretório Superior: home/rinaldo
Diretório Raiz: null
É absoluto? false
```

5. É possível recuperar os elementos do **Path** individualmente.

src/org/j6toj8/fileio/paths/Paths_Names.java

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome + "\n");

Path path = Paths.get(userHome, "arquivos", "arquivo.txt");
int nameCount = path.getNameCount(); // quantidade de elementos no Path
for (int i = 0; i < nameCount; i++) {
    Path name = path.getName(i); // recupera o elemento específico
    System.out.println(name);
}
```

Saída no console

```
User home: /home/rinaldo

home
rinaldo
arquivos
arquivo.txt
```

6. É possível converter um **Path** relativo para um absoluto.

src/org/j6toj8/fileio/paths/Paths_ToAbsolute.java

```
Path path = Paths.get("arquivos");
System.out.println(path);
System.out.println("É absoluto? " + path.isAbsolute());

System.out.println();

Path absolutePath = path.toAbsolutePath();
System.out.println(absolutePath);
System.out.println("É absoluto? " + absolutePath.isAbsolute());
```

Saída no console

```
arquivos
É absoluto? false

/home/rinaldo/Desenvolvimento/git/java6-to-java8/arquivos
É absoluto? true
```

Neste caso a saída do console vai depender do diretório onde a aplicação está sendo executada.

7. É possível criar *Sub-Paths* a partir de um **Path**.

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

Path path = Paths.get(userHome, "arquivos", "arquivo1.txt");
System.out.println("Path: " + path);

Path subpath1 = path.subpath(0, 1);
System.out.println(subpath1);

Path subpath2 = path.subpath(0, 2);
System.out.println(subpath2);

Path subpath3 = path.subpath(1, 3);
System.out.println(subpath3);

Path subpath4 = path.subpath(2, 4);
System.out.println(subpath4);

Path subpath5 = path.subpath(3, 5); // EXCEÇÃO, pois só existem 4 elementos no
path
```

Saída no console

```
User home: /home/rinaldo
Path: /home/rinaldo/arquivos/arquivo1.txt
home
/home/rinaldo
rinaldo/arquivos
arquivos/arquivo1.txt
Exception in thread "main" java.lang.IllegalArgumentException
    at sun.nio.fs.UnixPath.subpath(UnixPath.java:348)
    at sun.nio.fs.UnixPath.subpath(UnixPath.java:43)
    at org.j6toj8.fileio.paths.Paths_SubPath.main(Paths_SubPath.java:28)
```

8. É possível remover redundâncias de um **Path** com o método **normalize**.

src/org/j6toj8/fileio/paths/Paths_Normalize.java

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

System.out.println();

Path path1 = Paths.get(userHome, "arquivos./arquivo1.txt");
System.out.println("Path: " + path1);
System.out.println("Path normalize: " + path1.normalize());

System.out.println();

Path path2 = Paths.get(userHome, "arquivos../arquivo1.txt");
System.out.println("Path: " + path2);
System.out.println("Path normalize: " + path2.normalize());
```

Saída no console

```
User home: /home/rinaldo

Path: /home/rinaldo/arquivos./arquivo1.txt
Path normalize: /home/rinaldo/arquivos/arquivo1.txt

Path: /home/rinaldo/arquivos../arquivo1.txt
Path normalize: /home/rinaldo/arquivo1.txt
```

9. É possível unir duas instâncias de **Path** com o método **resolve**.

src/org/j6toj8/fileio/paths/Paths_Resolve.java

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

System.out.println();

Path path1 = Paths.get(userHome, "arquivos");
Path path2 = Paths.get("arquivo1.txt");
System.out.println("Absoluto + Relativo: " + path1.resolve(path2));
System.out.println("Relativo + Absoluto: " + path2.resolve(path1));
System.out.println("Absoluto + Absoluto: " + path1.resolve(path1));
System.out.println("Relativo + Relativo: " + path2.resolve(path2));
```

Saída no console

```
User home: /home/rinaldo  
Absoluto + Relativo: /home/rinaldo/arquivos/arquivo1.txt  
Relativo + Absoluto: /home/rinaldo/arquivos  
Absoluto + Absoluto: /home/rinaldo/arquivos  
Relativo + Relativo: arquivo1.txt/arquivo1.txt
```

Perceba que sempre que o argumento é um **Path** absoluto, o resultado final é ele mesmo.

Quando o argumento é um **Path** relativo, ele é acrescentado ao original, seja este absoluto ou relativo.

10. É possível derivar um **Path** de outro com o método **relativize**.

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

System.out.println();

Path pathAbsoluto1 = Paths.get(userHome, "arquivos");
Path pathAbsoluto2 = Paths.get(userHome, "arquivos/arquivo1.txt");
System.out.println("Absoluto 1: " + pathAbsoluto1);
System.out.println("Absoluto 2: " + pathAbsoluto2);

Path pathRelativo1 = Paths.get("arquivo1.txt");
Path pathRelativo2 = Paths.get("arquivos/arquivo1.txt");
System.out.println("Relativo 1: " + pathRelativo1);
System.out.println("Relativo 2: " + pathRelativo2);

System.out.println("Absoluto 1 + Absoluto 2: " + pathAbsoluto1.
    relativize(pathAbsoluto2));
System.out.println("Absoluto 2 + Absoluto 1: " + pathAbsoluto2.
    relativize(pathAbsoluto1));

System.out.println("Relativo 1 + Relativo 2: " + pathRelativo1.
    relativize(pathRelativo2));
System.out.println("Relativo 2 + Relativo 1: " + pathRelativo2.
    relativize(pathRelativo1));

try {
    // Exceção será lançada, pois não é possível chamar relativize em tipos diferentes de Path
    System.out.println("Absoluto + Relativo: " + pathAbsoluto1.
        relativize(pathRelativo1));
} catch (Exception e) {
    e.printStackTrace();
}

try {
    // Exceção será lançada, pois não é possível chamar relativize em tipos diferentes de Path
    System.out.println("Relativo + Absoluto: " + pathRelativo1.
        relativize(pathAbsoluto1));
} catch (Exception e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo

Absoluto 1: /home/rinaldo/arquivos
Absoluto 2: /home/rinaldo/arquivos/arquivo1.txt
Relativo 1: arquivo1.txt
Relativo 2: arquivos/arquivo1.txt
Absoluto 1 + Absoluto 2: arquivo1.txt
Absoluto 2 + Absoluto 1: ...
Relativo 1 + Relativo 2: ../arquivos/arquivo1.txt
Relativo 2 + Relativo 1: ../../arquivo1.txt
java.lang.IllegalArgumentException: 'other' is different type of Path
    at sun.nio.fs.UnixPath.relativize(UnixPath.java:416)
    at sun.nio.fs.UnixPath.relativize(UnixPath.java:43)
    at org.j6toj8.fileio.paths.Paths_Relativize.main(Paths_Relativize.java:33)
java.lang.IllegalArgumentException: 'other' is different type of Path
    at sun.nio.fs.UnixPath.relativize(UnixPath.java:416)
    at sun.nio.fs.UnixPath.relativize(UnixPath.java:43)
    at org.j6toj8.fileio.paths.Paths_Relativize.main(Paths_Relativize.java:40)
```

Todas essas combinações podem aparecer no exame, então entenda bem como cada uma delas se comporta. Lembre-se principalmente de que não é possível derivar um **Path** absoluto de um relativo, e vice-versa.

11. É possível converter um **Path** sintético, que não aponta de fato para um arquivo no sistema de arquivos, em um **Path** real, que aponta para um arquivo ou diretório que existe no sistema de arquivos.

src/org/j6toj8/fileio/paths/Paths_ToRealPath.java

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

Path pathQueExiste = Paths.get(userHome, "arquivo1.txt");
Path pathQueNaoExiste = Paths.get(userHome, "arquivoQueNaoExiste.txt");

try {
    Path realPath = pathQueExiste.toRealPath();
    System.out.println("realPath: " + realPath);
} catch (IOException e) {
    e.printStackTrace();
}

try {
    pathQueNaoExiste.toRealPath(); // LANÇA EXCEÇÃO
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
realPath: /home/rinaldo/arquivo1.txt
java.nio.file.NoSuchFileException: /home/rinaldo/arquivoQueNaoExiste.txt
    at sun.nio.fs.UnixException.translateToIOException(UnixException.java:86)
    at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:102)
    at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:107)
    at sun.nio.fs.UnixPath.toRealPath(UnixPath.java:837)
    at org.j6toj8.fileio.paths.Paths_ToRealPath.main(Paths_ToRealPath.java:25)
```

Perceba que é lançada exceção caso o arquivo realmente não exista no sistema de arquivos.

- Introducing NIO.2

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 454). Wiley. Edição do Kindle.

- [Java NIO2 Path API](#).
- [Class Paths](#). Java Platform SE 7.
- [Path Operations](#). The Java™ Tutorials.

Files

Objetivo

Check, delete, copy, or move a file or directory by using the `Files` class

-

Checar, deletar, copiar ou mover um arquivo ou diretório utilizando a classe `Files`

A classe `Files` contém inúmeros métodos estáticos para lidar com instâncias de `Path`. A maioria dos métodos recebem instâncias de `Path` para realizar alguma operação no arquivo ou diretório representado pelo `Path`. Muitos irão lançar uma exceção caso o arquivo ou diretório não exista no sistema de arquivos.

1. É possível checar vários atributos de um `Path` utilizando a classe `Files`.

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

Path path1 = Paths.get(userHome, "arquivo1.txt");
Path path2 = Paths.get(userHome, "arquivos");

System.out.println("Path 1: " + path1);
System.out.println("Path 2: " + path2);

System.out.println("Path 1 existe? " + Files.exists(path1));
System.out.println("Path 2 existe? " + Files.exists(path2));

System.out.println("Path 1 NÃO existe? " + Files.notExists(path1));
System.out.println("Path 2 NÃO existe? " + Files.notExists(path2));

System.out.println("Path 1 é um diretório? " + Files.isDirectory(path1));
System.out.println("Path 1 é um arquivo comum? " + Files.isRegularFile(path1));
System.out.println("Path 1 é um link simbólico? " + Files.isSymbolicLink(path1));

System.out.println("Aplicação possui permissão de leitura no Path 1? " +
Files.isReadable(path1));
System.out.println("Aplicação possui permissão de escrita no Path 1? " +
Files.isWritable(path1));
System.out.println("Aplicação possui permissão de execução no Path 1? " +
Files.isExecutable(path1));

try {
    System.out.println("Qual o tamanho de Path 1? " + Files.size(path1));
} catch (IOException e1) {
    // Lança exceção se o arquivo não existir
    e1.printStackTrace();
}

try {
    System.out.println("Path 1 é oculto? " + Files.isHidden(path1));
} catch (IOException e) {
    // Lança exceção se o arquivo não existir
    e.printStackTrace();
}

try {
    System.out.println("Path 1 e Path 1 são iguais? " + Files.isSameFile(path1,
path1));
} catch (IOException e) {
    // Lança exceção se o arquivo não existir
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Path 1: /home/rinaldo/arquivo1.txt
Path 2: /home/rinaldo/arquivos
Path 1 existe? true
Path 2 existe? false
Path 1 NÃO existe? false
Path 2 NÃO existe? true
Path 1 é um diretório? false
Path 1 é um arquivo comum? true
Path 1 é um link simbólico? false
Aplicação possui permissão de leitura no Path 1? true
Aplicação possui permissão de escrita no Path 1? true
Aplicação possui permissão de execução no Path 1? false
Path 1 é oculto? false
Path 1 e Path 1 são iguais? true
```

Perceba que algumas chamadas lançam `IOException`. Isso ocorre pois elas irão lançar essa exceção caso o `Path` não exista no sistema de arquivos.

2. É possível verificar se dois `Path`'s são iguais, mesmo que estejam representados de forma diferente.

`src/org/j6toj8/fileio/files/Files_SameFile.java`

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

Path path1 = Paths.get(userHome, "arquivo1.txt");
Path path2 = Paths.get("/", "home", "rinaldo", "arquivo1.txt");
Path path3 = Paths.get("/home/rinaldo/arquivo1.txt");
Path path4 = Paths.get("/home/rinaldo/Downloads/../arquivo1.txt");

try {
    // todos os Path são iguais
    System.out.println("Path 1 e Path 2 são iguais? " + Files.isSameFile(path1,
path2));
    System.out.println("Path 2 e Path 3 são iguais? " + Files.isSameFile(path2,
path3));
    System.out.println("Path 3 e Path 4 são iguais? " + Files.isSameFile(path3,
path4));
    System.out.println("Path 1 e Path 4 são iguais? " + Files.isSameFile(path1,
path4));
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Path 1 e Path 2 são iguais? true
Path 2 e Path 3 são iguais? true
Path 3 e Path 4 são iguais? true
Path 1 e Path 4 são iguais? true
```

Perceba que nesse exemplo todos os **Path** são iguais, pois apontam para o mesmo arquivo. Ou seja, o método `realmetne` verifica se o arquivo no sistema de arquivos é o mesmo, independente da forma como o diretório está sendo representado no **Path**. Isso funcionará inclusive para links simbólicos que apontam para o mesmo arquivo.

3. É possível criar arquivos utilizando a classe **Files**.

src/org/j6toj8/fileio/files/Files_CreateFile.java

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

// Utilizando um nome aleatório de arquivo,
// apenas para o exemplo executar inúmeras vezes sem problemas
String nomeAleatorio = "arquivo" + new Random().nextInt() + ".txt";

Path path = Paths.get(userHome, nomeAleatorio);
System.out.println("Path: " + path);

try {
    System.out.println("Path existe? " + Files.exists(path));
    Files.createFile(path);
    System.out.println("Path existe? " + Files.exists(path));
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Path: /home/rinaldo/arquivo8481614.txt
Path existe? false
Path existe? true
```

4. É possível criar um diretório utilizando a classe **Files**.

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

// Utilizando um nome aleatório de diretório,
// apenas para o exemplo executar inúmeras vezes sem problemas
String nomeAleatorio = "arquivo" + new Random().nextInt();

Path path = Paths.get(userHome, nomeAleatorio);
System.out.println("Path: " + path);

try {
    System.out.println("Path existe? " + Files.exists(path));
    Files.createDirectory(path);
    System.out.println("Path existe? " + Files.exists(path));
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Path: /home/rinaldo/arquivo411247666
Path existe? false
Path existe? true
```

5. Caso o **Path** possua vários elementos a serem criados, é necessário utilizar o método **createDirectories**, no plural, caso contrário será lançada uma exceção.

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

// Utilizando um nome aleatório de diretório,
// apenas para o exemplo executar inúmeras vezes sem problemas
String nomeAleatorio1 = "arquivo" + new Random().nextInt();
String nomeAleatorio2 = "arquivo" + new Random().nextInt();
String nomeAleatorio3 = "arquivo" + new Random().nextInt();

Path path = Paths.get(userHome, nomeAleatorio1, nomeAleatorio2, nomeAleatorio3);
System.out.println("Path: " + path);

try {
    Files.createDirectory(path); // MÉTODO ERRADO, LANÇA EXCEÇÃO
} catch (IOException e) {
    e.printStackTrace();
}

try {
    System.out.println("Path existe? " + Files.exists(path));
    Files.createDirectories(path);
    System.out.println("Path existe? " + Files.exists(path));
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Path: /home/rinaldo/arquivo783746545/arquivo248290405/arquivo801162958
java.nio.file.NoSuchFileException:
/home/rinaldo/arquivo783746545/arquivo248290405/arquivo801162958
    at sun.nio.fs.UnixException.translateToIOException(UnixException.java:86)
    at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:102)
    at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:107)
    at
sun.nio.fs.UnixFileSystemProvider.createDirectory(UnixFileSystemProvider.java:384)
    at java.nio.file.Files.createDirectory(Files.java:674)
    at
org.j6toj8.fileio.files.Files_CreateDirectories.main(Files_CreateDirectories.java:2
6)
Path existe? false
Path existe? true
```

6. É possível copiar uma arquivo ou diretório.

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

// Utilizando um nome aleatório de arquivo,
// apenas para o exemplo executar inúmeras vezes sem problemas
String nomeAleatorio = "arquivo" + new Random().nextInt() + ".txt";

Path path1 = Paths.get(userHome, nomeAleatorio);
Path path2 = Paths.get(userHome, nomeAleatorio + "-copia.txt");
System.out.println("Path 1: " + path1);
System.out.println("Path 2: " + path2);

try {
    System.out.println("Path 1 existe? " + Files.exists(path1));
    Files.createFile(path1);
    System.out.println("Path 1 existe? " + Files.exists(path1));
    System.out.println("Path 2 existe? " + Files.exists(path2));
    Files.copy(path1, path2);
    System.out.println("Path 2 existe? " + Files.exists(path2));
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Path 1: /home/rinaldo/arquivo-1156333590.txt
Path 2: /home/rinaldo/arquivo-1156333590.txt-copia.txt
Path 1 existe? false
Path 1 existe? true
Path 2 existe? false
Path 2 existe? true
```

Ao copiar um arquivo, é necessário que os diretórios já existam, caso contrário será lançada uma exceção.

Ao copiar um diretório, o conteúdo dele não será copiado.

7. É possível copiar a partir de um **FileInputStream** para um **Path**.

src/org/j6toj8/fileio/files/Files_CopyToPath.java

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

// Utilizando um nome aleatório de arquivo,
// apenas para o exemplo executar inúmeras vezes sem problemas
String nomeAleatorio = "copia" + new Random().nextInt() + ".txt";

try (FileInputStream fis = new FileInputStream(userHome + File.separator +
"arquivo1.txt")) {
    Path pathParaCopia = Paths.get(userHome, nomeAleatorio);
    System.out.println("Path 2 existe? " + Files.exists(pathParaCopia));
    Files.copy(fis, pathParaCopia);
    System.out.println("Path 2 existe? " + Files.exists(pathParaCopia));
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Path 2 existe? false
Path 2 existe? true
```

8. É possível copiar para um `FileOutputStream` a partir de `Path`.

src/org/j6toj8/fileio/files/Files_CopyFromPath.java

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

// Utilizando um nome aleatório de arquivo,
// apenas para o exemplo executar inúmeras vezes sem problemas
String nomeAleatorio = "copia" + new Random().nextInt() + ".txt";

try (FileOutputStream fos = new FileOutputStream(userHome + File.separator +
nomeAleatorio)) {
    Path pathParaCopia = Paths.get(userHome, "arquivo1.txt");
    Files.copy(pathParaCopia, fos);

    Path pathCriado = Paths.get(userHome, nomeAleatorio);
    System.out.println("Path criado: " + pathCriado);
    System.out.println("Path criado existe?: " + Files.exists(pathCriado));
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Path criado: /home/rinaldo/copia860242436.txt
Path criado existe?: true
```

9. Também é possível mover um **Path**, seja ele um diretório ou um arquivo.

src/org/j6toj8/fileio/files/Files_MoveFile.java

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

String nomeAleatorio = "arquivo" + new Random().nextInt() + ".txt";

Path arquivoOrigem = Paths.get(userHome, nomeAleatorio);
Path arquivoDestino = Paths.get(userHome, nomeAleatorio + "-movido.txt");
System.out.println("Path Arquivo Origem: " + arquivoOrigem);
System.out.println("Path Arquivo Destino: " + arquivoDestino);

try {
    System.out.println("Arquivo origem existe? " + Files.exists(arquivoOrigem));
    Files.createFile(arquivoOrigem);
    System.out.println("Arquivo origem existe? " + Files.exists(arquivoOrigem));

    System.out.println("Arquivo destino existe? " + Files.exists(arquivoDestino));
    Files.move(arquivoOrigem, arquivoDestino);
    System.out.println("Arquivo destino existe? " + Files.exists(arquivoDestino));
    System.out.println("Arquivo origem existe? " + Files.exists(arquivoOrigem));
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Path Arquivo Origem: /home/rinaldo/arquivo-2044553267.txt
Path Arquivo Destino: /home/rinaldo/arquivo-2044553267.txt-movido.txt
Arquivo origem existe? false
Arquivo origem existe? true
Arquivo destino existe? false
Arquivo destino existe? true
Arquivo origem existe? false
```

Ao mover um diretório, o conteúdo será movido também.

Ao mover um **Path** para um diretório que não exista, será lançado exceção.

10. É possível apagar um **Path**.

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

// Utilizando um nome aleatório de arquivo,
// apenas para o exemplo executar inúmeras vezes sem problemas
String nomeAleatorio = "arquivo" + new Random().nextInt() + ".txt";

Path path = Paths.get(userHome, nomeAleatorio);
System.out.println("Path: " + path);

try {
    System.out.println("Path existe? " + Files.exists(path));
    Files.createFile(path);
    System.out.println("Path existe? " + Files.exists(path));

    Files.delete(path); // tenta apagar o Path e lança exceção se ele não existir
    System.out.println("Path existe? " + Files.exists(path));

    Files.deleteIfExists(path); // tenta apagar o Path e não faz nada se ele não existir
    System.out.println("Path existe? " + Files.exists(path));
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Path: /home/rinaldo/arquivo-113782412.txt
Path existe? false
Path existe? true
Path existe? false
Path existe? false
```

Perceba que existem dois métodos diferentes para apagar. O primeira irá lançar exceção se o arquivo não existir, o segundo não irá lançar.

Ao tentar apagar um diretório com conteúdo será lançada uma exceção.

11. É possível escrever e ler arquivos com **Files** e **Path**.

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

// Utilizando um nome aleatório de arquivo,
// apenas para o exemplo executar inúmeras vezes sem problemas
String nomeAleatorio = "arquivo" + new Random().nextInt() + ".txt";

Path path = Paths.get(userHome, nomeAleatorio);
System.out.println("Path: " + path);
try {
    System.out.println("Path existe? " + Files.exists(path));
    Files.createFile(path); // cria o arquivo
    System.out.println("Path existe? " + Files.exists(path));
} catch (IOException e1) {
    e1.printStackTrace();
}

try (BufferedWriter bw = Files.newBufferedWriter(path)) {
    // escreve no arquivo
    bw.write("1");
    bw.write("2");
    bw.write("3");
} catch (IOException e) {
    e.printStackTrace();
}

try (BufferedReader br = Files.newBufferedReader(path)) {
    // lê as linhas do arquivo
    String line = br.readLine();
    do {
        System.out.println(line);
    } while (br.readLine() != null);
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Path: /home/rinaldo/arquivo-1467732927.txt
Path existe? false
Path existe? true
123
```

Perceba que primeiro foi criado o arquivo. Depois ele foi escrito com um `BufferedWriter`. E depois foi lido com um `BufferedReader`.

12. É possível ler todas as linhas de um arquivo com uma única chamada.

src/org/j6toj8/fileio/files/Files_ReadAllLines.java

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

Path path = Paths.get(userHome, "arquivo.txt");
try {
    List<String> conteudo = Files.readAllLines(path);
    System.out.println(conteudo);
} catch (IOException e) {
    e.printStackTrace();
}
```

arquivo.txt

```
1
2
3
4
5
```

Saída no console

```
User home: /home/rinaldo
[1, 2, 3, 4, 5]
```

13. É possível verificar e alterar a data de modificação de um **Path**.

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

// Utilizando um nome aleatório de arquivo,
// apenas para o exemplo executar inúmeras vezes sem problemas
String nomeAleatorio = "arquivo" + new Random().nextInt() + ".txt";

Path path = Paths.get(userHome, nomeAleatorio);
System.out.println("Path: " + path);

try {
    Files.createFile(path);
    System.out.println("Data de Modificação: " + Files.getLastModifiedTime(path));
    FileTime fileTime = FileTime.from(Instant.now().plusMillis(10000));
    Files.setLastModifiedTime(path, fileTime);
    System.out.println("Data de Modificação: " + Files.getLastModifiedTime(path));
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Path: /home/rinaldo/arquivo805496635.txt
Data de Modificação: 2019-08-14T18:53:23.710066Z
Data de Modificação: 2019-08-14T18:53:33.724Z
```

14. É possível modificar o *Owner* (autor/dono) do arquivo.

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

// Utilizando um nome aleatório de arquivo,
// apenas para o exemplo executar inúmeras vezes sem problemas
String nomeAleatorio = "arquivo" + new Random().nextInt() + ".txt";

Path path = Paths.get(userHome, nomeAleatorio);
System.out.println("Path: " + path);

try {
    Files.createFile(path);
    System.out.println(Files.getOwner(path)); // imprime o owner atual

    // Pega o serviço do sistema para buscar um usuário
    UserPrincipalLookupService service = FileSystems.getDefault()
        .getUserPrincipalLookupService();
    // Busca pelo usuário com nome 'rinaldo'
    UserPrincipal userPrincipal = service.lookupPrincipalByName("rinaldo");

    Files.setOwner(path, userPrincipal); // altera o owner
    System.out.println(Files.getOwner(path));
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Path: /home/rinaldo/arquivo-2139809923.txt
rinaldo
rinaldo
```

Perceba que é necessário utilizar as classes `UserPrincipalLookupService` e `UserPrincipal` para buscar um usuário no sistema e atribuí-lo como novo *Owner*. Essas operações podem depender da permissão do usuário, e por isso podem lançar exceção caso não possam ser executadas.

15. É possível recuperar todos os atributos de um arquivo com uma única chamada, utilizando a classe `BasicFileAttributes`.

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

Path path = Paths.get(userHome, "arquivo.txt");

try {
    BasicFileAttributes attributes = Files.readAttributes(path,
    BasicFileAttributes.class);
    System.out.println("Tamanho: " + attributes.size());

    System.out.println("É diretório? " + attributes.isDirectory());
    System.out.println("É link simbólico? " + attributes.isSymbolicLink());
    System.out.println("É um arquivo comum? " + attributes.isRegularFile());
    System.out.println("Não é nenhuma das opções acima? " + attributes.isOther());

    System.out.println("Data de Criação: " + attributes.creationTime());
    System.out.println("Último acesso: " + attributes.lastAccessTime());
    System.out.println("Última modificação: " + attributes.lastModifiedTime());
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo
Tamanho: 10
É diretório? false
É link simbólico? false
É um arquivo comum? true
Não é nenhuma das opções acima? false
Data de Criação: 2019-08-14T18:45:44.189475Z
Último acesso: 2019-08-14T18:45:44.205475Z
Última modificação: 2019-08-14T18:45:44.189475Z
```

Utilizar a classe **BasicFileAttributes** para recuperar os atributos de um arquivo pode trazer ganhos de performance, já que é necessária apenas uma ida ao sistema operacional para recuperar todos os atributos.

16. É possível modificar as datas de criação, modificação e acesso do arquivo com a classe **BasicFileAttributeView**.

```
String userHome = System.getProperty("user.home");
System.out.println("User home: " + userHome);

Path path = Paths.get(userHome, "arquivo.txt");

try {
    BasicFileAttributeView attributesView = Files.getFileAttributeView(path,
    BasicFileAttributeView.class);
    BasicFileAttributes attributesAntigos = attributesView.readAttributes();

    System.out.println("\nData de Criação original: " +
attributesAntigos.creationTime());
    System.out.println("Último acesso original: " + attributesAntigos.
lastAccessTime());
    System.out.println("Última modificação original: " +
attributesAntigos.lastModifiedTime());

    FileTime fileTime = FileTime.from(Instant.now().plusMillis(10000));
    attributesView.setTimes(fileTime, fileTime, fileTime);

    BasicFileAttributes attributesNovos = attributesView.readAttributes();
    System.out.println("\nData de Criação alterada: " + attributesNovos.
creationTime());
    System.out.println("Último acesso alterada: " + attributesNovos.
lastAccessTime());
    System.out.println("Última modificação alterada: " +
attributesNovos.lastModifiedTime());
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
User home: /home/rinaldo

Data de Criação original: 2019-08-14T19:24:04.548Z
Último acesso original: 2019-08-14T19:24:04.548Z
Última modificação original: 2019-08-14T19:24:04.548Z

Data de Criação alterada: 2019-08-14T19:24:32.995Z
Último acesso alterada: 2019-08-14T19:24:32.995Z
Última modificação alterada: 2019-08-14T19:24:32.995Z
```

Perceba que a partir da classe `BasicFileAttributeView` também é possível ler os atributos do arquivo chamando o método `readAttributes`.

- Introducing NIO.2

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 454). Wiley. Edição do Kindle.

- [Introduction to the Java NIO2 File API.](#)
- [Class Files.](#) Java Platform SE 7.
- [Path Operations.](#) The Java™ Tutorials.

DirectoryStream e FileVisitor

Objetivo

Recursively access a directory tree by using the DirectoryStream and FileVisitor interfaces

- Acessar recursivamente uma árvore de diretório usando as interfaces DirectoryStream e FileVisitor

Nesta seção serão apresentadas duas classes para percorrer diretórios: [DirectoryStream](#) e [FileVisitor](#).

1. É possível checar vários atributos de um [Path](#) utilizando a classe [Files](#).

src/org/j6toj8/fileio/recursiveaccess/Recursive_DirectoryStream.java

```
String userHome = System.getProperty("user.home");
Path path = Paths.get(userHome, "arquivos");
System.out.println("Path: " + path);

try (DirectoryStream<Path> directoryStream = Files.newDirectoryStream(path)) {
    // iteração com sintaxe for-each
    for (Path subPath : directoryStream) {
        System.out.println(subPath);
    }
} catch (IOException e) {
    e.printStackTrace();
}

try (DirectoryStream<Path> directoryStream = Files.newDirectoryStream(path)) {
    // iteração com operação forEach e expressão lambda
    directoryStream.forEach(p -> System.out.println(p));
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
Path: /home/rinaldo/arquivos  
/home/rinaldo/arquivos/arquivo1.txt  
/home/rinaldo/arquivos/arquivo3.txt  
/home/rinaldo/arquivos/subpasta1  
/home/rinaldo/arquivos/arquivo2.txt  
/home/rinaldo/arquivos/arquivo1.txt  
/home/rinaldo/arquivos/arquivo3.txt  
/home/rinaldo/arquivos/subpasta1  
/home/rinaldo/arquivos/arquivo2.txt
```

Perceba que a instância de [DirectoryStream](#):

- Pode ser utilizada com try-with-resources.
- Pode ser iterada com a sintaxe de for-each.
- Pode ser iterada com o método [forEach](#) que recebe uma expressão lambda.
- Lança exceção caso o arquivo não exista.
- Não lista os subdiretórios de forma recursiva, mas sim apenas o primeiro nível.
- Não tem relação com a API de *Streams* do Java 8.

2. É possível acessar toda uma árvore de diretórios utilizando um [FileVisitor](#).

```
// Implementação simples de um SimpleFileVisitor
static class MeuFileVisitor extends SimpleFileVisitor<Path> {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws
IOException {
        // Método que será invocado a cada arquivo encontrado
        System.out.println("Arquivo visitado: " + file + ". Tamanho: " + attrs.size());
        return FileVisitResult.CONTINUE; // instrui o FileVisitor a continuar seguindo
a árvore de arquivos
    }
}

public static void main(String[] args) {
    String userHome = System.getProperty("user.home");
    Path path = Paths.get(userHome, "arquivos");
    System.out.println("Path: " + path);
    try {
        // Utilização da classe MeuFileVisitor para acessar
        // todos os arquivos no diretório e seus subdiretórios
        Files.walkFileTree(path, new MeuFileVisitor());
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Saída no console

```
Path: /home/rinaldo/arquivos
Arquivo visitado: /home/rinaldo/arquivos/arquivo1.txt. Tamanho: 2
Arquivo visitado: /home/rinaldo/arquivos/arquivo3.txt. Tamanho: 10
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/subpasta12/arquivo122.txt.
Tamanho: 10
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/subpasta12/arquivo121.txt.
Tamanho: 10
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/arquivo11.txt. Tamanho: 10
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/arquivo12.txt. Tamanho: 2
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/arquivo13.txt. Tamanho: 2
Arquivo visitado: /home/rinaldo/arquivos/arquivo2.txt. Tamanho: 2
```

Perceba que todos os arquivos do diretório e subdiretórios foram visitados e impressos no console ao invocar `Files.walkFileTree`. Para cada arquivo encontrado foi invocado o método `visitFile` da instância de `MeuFileVisitor`.

Veja que foi retornado `FileVisitResult.CONTINUE` do método `visitFile`. Isso instrui o FileVisitor a continuar visitando a árvore de arquivos. Também é possível retornar `TERMINATE`, `SKIP_SUBTREE` e `SKIP_SIBLINGS`, que serão apresentados a seguir.

3. É possível finalizar a visitação quando for necessário retornando `FileVisitResult.TERMINATE`.

```
static class MeuFileVisitor extends SimpleFileVisitor<Path> {  
    @Override  
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws  
    IOException {  
        System.out.println("Arquivo visitado: " + file + ". Tamanho: " + attrs.size());  
        if (file.getFileName().toString().equals("arquivo122.txt")) {  
            System.out.println("Arquivo encontrado. Finalizando.");  
            return FileVisitResult.TERMINATE; // finaliza o acesso à árvore de arquivos  
        }  
        return FileVisitResult.CONTINUE;  
    }  
}  
  
public static void main(String[] args) {  
    String userHome = System.getProperty("user.home");  
    Path path = Paths.get(userHome, "arquivos");  
    System.out.println("Path: " + path);  
    try {  
        Files.walkFileTree(path, new MeuFileVisitor());  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

Saída no console

```
Path: /home/rinaldo/arquivos  
Arquivo visitado: /home/rinaldo/arquivos/arquivo1.txt. Tamanho: 2  
Arquivo visitado: /home/rinaldo/arquivos/arquivo3.txt. Tamanho: 10  
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/subpasta12/arquivo122.txt.  
Tamanho: 10  
Arquivo encontrado. Finalizando.
```

4. Também é possível tomar decisões antes e depois de visitar diretórios.

```
static class MeuFileVisitor extends SimpleFileVisitor<Path> {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws
IOException {
        System.out.println("Arquivo visitado: " + file + ". Tamanho: " + attrs.size());
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
throws IOException {
        // Ação que será executada antes de visitar um diretório
        if (dir.getFileName().toString().equals("subpasta12")) {
            return FileVisitResult.SKIP_SUBTREE; // ignora o diretório subpasta12
        }
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws
IOException {
        // Ação que será executada depois de visitar um diretório
        return FileVisitResult.CONTINUE;
    }
}

public static void main(String[] args) {
    String userHome = System.getProperty("user.home");
    Path path = Paths.get(userHome, "arquivos");
    System.out.println("Path: " + path);
    try {
        Files.walkFileTree(path, new MeuFileVisitor());
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Saída no console

```
Path: /home/rinaldo/arquivos
Arquivo visitado: /home/rinaldo/arquivos/arquivo1.txt. Tamanho: 2
Arquivo visitado: /home/rinaldo/arquivos/arquivo3.txt. Tamanho: 10
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/arquivo11.txt. Tamanho: 10
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/arquivo12.txt. Tamanho: 2
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/arquivo13.txt. Tamanho: 2
Arquivo visitado: /home/rinaldo/arquivos/arquivo2.txt. Tamanho: 2
```

5. Também é possível ignorar todos os elementos que estão no mesmo nível de um **Path**.

src/org/j6toj8/fileio/recursiveaccess/Recursive_VisitorIgnoreSiblings.java

```
static class MeuFileVisitor extends SimpleFileVisitor<Path> {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws
    IOException {
        System.out.println("Arquivo visitado: " + file + ". Tamanho: " + attrs.size());
        if (file.getFileName().toString().equals("arquivo11.txt")) {
            // ao encontrar o arquivo11.txt irá parar de visitar qualquer
            // diretório ou arquivo que seja "irmão" (que está lado a lado)
            return FileVisitResult.SKIP_SIBLINGS;
        }
        return FileVisitResult.CONTINUE;
    }
}

public static void main(String[] args) {
    String userHome = System.getProperty("user.home");
    Path path = Paths.get(userHome, "arquivos");
    System.out.println("Path: " + path);
    try {
        Files.walkFileTree(path, new MeuFileVisitor());
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Saída no console

```
Path: /home/rinaldo/arquivos
Arquivo visitado: /home/rinaldo/arquivos/arquivo1.txt. Tamanho: 2
Arquivo visitado: /home/rinaldo/arquivos/arquivo3.txt. Tamanho: 10
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/subpasta12/arquivo122.txt.
Tamanho: 10
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/subpasta12/arquivo121.txt.
Tamanho: 10
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/arquivo11.txt. Tamanho: 10
Arquivo visitado: /home/rinaldo/arquivos/arquivo2.txt. Tamanho: 2
```

6. É possível implementar diretamente a interface **FileVisitor**, ao invés de **SimpleFileVisitor**, e implementar todos os seus métodos.

```
static class MeuFileVisitor implements FileVisitor<Path> {  
    @Override  
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws  
    IOException {  
        System.out.println("Arquivo visitado: " + file);  
        return FileVisitResult.CONTINUE;  
    }  
  
    @Override  
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)  
    throws IOException {  
        System.out.println("Antes de visitar o diretório: " + dir);  
        return FileVisitResult.CONTINUE;  
    }  
  
    @Override  
    public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws  
    IOException {  
        System.out.println("Após de visitar o diretório: " + dir);  
        return FileVisitResult.CONTINUE;  
    }  
  
    @Override  
    public FileVisitResult visitFileFailed(Path file, IOException exc) throws  
    IOException {  
        System.out.println("Falhou ao visitar o arquivo: " + file);  
        return FileVisitResult.CONTINUE;  
    }  
}  
  
public static void main(String[] args) {  
    String userHome = System.getProperty("user.home");  
    Path path = Paths.get(userHome, "arquivos");  
    System.out.println("Path: " + path);  
    try {  
        Files.walkFileTree(path, new MeuFileVisitor());  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

Saída no console

```
Path: /home/rinaldo/arquivos
Antes de visitar o diretório: /home/rinaldo/arquivos
Arquivo visitado: /home/rinaldo/arquivos/arquivo1.txt
Arquivo visitado: /home/rinaldo/arquivos/arquivo3.txt
Antes de visitar o diretório: /home/rinaldo/arquivos/subpasta1
Antes de visitar o diretório: /home/rinaldo/arquivos/subpasta1/subpasta12
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/subpasta12/arquivo122.txt
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/subpasta12/arquivo121.txt
Após de visitar o diretório: /home/rinaldo/arquivos/subpasta1/subpasta12
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/arquivo11.txt
Antes de visitar o diretório: /home/rinaldo/arquivos/subpasta1/subpasta11
Após de visitar o diretório: /home/rinaldo/arquivos/subpasta1/subpasta11
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/arquivo12.txt
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/arquivo13.txt
Após de visitar o diretório: /home/rinaldo/arquivos/subpasta1
Arquivo visitado: /home/rinaldo/arquivos/arquivo2.txt
Após de visitar o diretório: /home/rinaldo/arquivos
```

7. É possível definir opções adicionais e limitar a profundidade utilizando outra versão do método `walkFileTree`.

src/org/j6toj8/fileio/recursiveaccess/Recursive_VisitorOptionsAndDepth.java

```
static class MeuFileVisitor extends SimpleFileVisitor<Path> {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws
IOException {
        System.out.println("Arquivo visitado: " + file);
        return FileVisitResult.CONTINUE;
    }
}

public static void main(String[] args) {
    String userHome = System.getProperty("user.home");
    Path path = Paths.get(userHome, "arquivos");
    System.out.println("Path: " + path);
    try {
        HashSet<FileVisitOption> opcoes = new HashSet<FileVisitOption>();
        opcoes.add(FileVisitOption.FOLLOW_LINKS);

        // visita a árvore com limite de profundidade 2
        // e com a opção de também visitar links simbólicos
        Files.walkFileTree(path, opcoes, 2, new MeuFileVisitor());
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Saída no console

```
Path: /home/rinaldo/arquivos
Arquivo visitado: /home/rinaldo/arquivos/arquivo1.txt
Arquivo visitado: /home/rinaldo/arquivos/arquivo3.txt
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/subpasta12
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/arquivo11.txt
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/subpasta11
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/arquivo12.txt
Arquivo visitado: /home/rinaldo/arquivos/subpasta1/arquivo13.txt
Arquivo visitado: /home/rinaldo/arquivos/arquivo2.txt
```

- Working with Directories

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 617). Wiley. Edição do Kindle.

- [List Files in a Directory in Java.](#)
- [A Guide To NIO2 FileVisitor.](#)
- [Class DirectoryStream.](#) Java Platform SE 7.
- [Class FileVisitor.](#) Java Platform SE 7.
- [Path Operations.](#) The Java™ Tutorials.

Files com Streams

Objetivo

Find a file by using the PathMatcher interface, and use Java SE 8 I/O improvements, including Files.find(), Files.walk(), and lines() methods

-

Encontrar um arquivo usando a interface PathMatcher, e usar as melhorias de I/O do Java SE 8, incluindo os métodos Files.find(), Files.walk(), and lines()

Nesta seção serão apresentadas melhorias do Java 8 para encontrar e ler arquivos. São operações que já poderiam ser realizadas com outros métodos antes do Java 8. Porém, com essas melhorias, é possível realizar essas operações utilizando Streams.

1. É possível criar um *Stream* para acessar todos os arquivos, diretórios e subdiretórios de um **Path**.

```
String userHome = System.getProperty("user.home");
Path path = Paths.get(userHome, "arquivos");
System.out.println("Path: " + path);

try {
    System.out.println("\nTodos os arquivos e diretórios: ");
    Files.walk(path) // cria o stream
        .forEach(System.out::println); // imprime no console
} catch (IOException e) {
    e.printStackTrace();
}

try {
    System.out.println("\nOs primeiro 5 arquivos e diretórios: ");
    Files.walk(path)
        .limit(5)
        .forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
Path: /home/rinaldo/arquivos

Todos os arquivos e diretórios:
/home/rinaldo/arquivos
/home/rinaldo/arquivos/arquivo1.txt
/home/rinaldo/arquivos/arquivo3.txt
/home/rinaldo/arquivos/subpasta1
/home/rinaldo/arquivos/subpasta1/subpasta12
/home/rinaldo/arquivos/subpasta1/subpasta12/arquivo122.txt
/home/rinaldo/arquivos/subpasta1/subpasta12/arquivo121.txt
/home/rinaldo/arquivos/subpasta1/arquivo11.txt
/home/rinaldo/arquivos/subpasta1/subpasta11
/home/rinaldo/arquivos/subpasta1/arquivo12.txt
/home/rinaldo/arquivos/subpasta1/arquivo13.txt
/home/rinaldo/arquivos/arquivo2.txt
```

```
Os primeiro 5 arquivos e diretórios:
/home/rinaldo/arquivos
/home/rinaldo/arquivos/arquivo1.txt
/home/rinaldo/arquivos/arquivo3.txt
/home/rinaldo/arquivos/subpasta1
/home/rinaldo/arquivos/subpasta1/subpasta12
```

Perceba que a instância criada é realmente um `Stream<Path>`, de tal forma que é possível realizar as operações disponíveis em qualquer `Stream`, como o método `filter`.

2. Existe uma versão do método `walk` para definir opções adicionais e limitar a profundidade do acesso aos subdiretórios.

src/org/j6toj8/fileio/fileimprovements/Improvements_WalkDepth.java

```
String userHome = System.getProperty("user.home");
Path path = Paths.get(userHome, "arquivos");
System.out.println("Path: " + path);

try {
    System.out.println("\nArquivos e Links simbólicos até o segundo nível: ");
    Files.walk(path, 2, FileVisitOption.FOLLOW_LINKS)
        .forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
Path: /home/rinaldo/arquivos

Arquivos e Links simbólicos até o segundo nível:
/home/rinaldo/arquivos
/home/rinaldo/arquivos/arquivo1.txt
/home/rinaldo/arquivos/arquivo3.txt
/home/rinaldo/arquivos/subpasta1
/home/rinaldo/arquivos/subpasta1/subpasta12
/home/rinaldo/arquivos/subpasta1/arquivo11.txt
/home/rinaldo/arquivos/subpasta1/subpasta11
/home/rinaldo/arquivos/subpasta1/arquivo12.txt
/home/rinaldo/arquivos/subpasta1/arquivo13.txt
/home/rinaldo/arquivos/arquivo2.txt
```

3. É possível pesquisar por um arquivo utilizando o método `find` e filtrar por atributos.

```
String userHome = System.getProperty("user.home");
Path path = Paths.get(userHome, "arquivos");
System.out.println("Path: " + path);

try {
    System.out.println("\nTodos os arquivos, ignorando diretórios, até o segundo nível: ");
    // ao chamar o find:
    // primeiro argumento: o path inicial
    // segundo argumento: o limite de profundidade
    // terceiro argumento: expressão lambda para filtrar
    Files.find(path, 2, (p, a) -> a.isRegularFile())
        .forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
Path: /home/rinaldo/arquivos
```

```
Todos os arquivos, ignorando diretórios, até o segundo nível:
```

```
/home/rinaldo/arquivos/arquivo1.txt
/home/rinaldo/arquivos/arquivo3.txt
/home/rinaldo/arquivos/subpasta1/arquivo11.txt
/home/rinaldo/arquivos/subpasta1/arquivo12.txt
/home/rinaldo/arquivos/subpasta1/arquivo13.txt
/home/rinaldo/arquivos/arquivo2.txt
```

Perceba que ao utilizar o `find` a expressão lambda tem acesso ao `Path` e seus atributos, que é uma instância de `BasicFileAttributes`, permitindo uma maior flexibilidade na busca.

4. É possível listar o conteúdo de um `Path` utilizando o método `list`.

```
String userHome = System.getProperty("user.home");
Path path = Paths.get(userHome, "arquivos");
System.out.println("Path: " + path);

try {
    System.out.println("\nListagem do diretório: ");
    Files.list(path)
        .forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}

try {
    System.out.println("\nListagem do diretório, apenas arquivos: ");
    Files.list(path)
        .filter(p -> Files.isRegularFile(p))
        .forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
Path: /home/rinaldo/arquivos

Listagem do diretório:
/home/rinaldo/arquivos/arquivo1.txt
/home/rinaldo/arquivos/arquivo3.txt
/home/rinaldo/arquivos/subpasta1
/home/rinaldo/arquivos/arquivo2.txt
```

```
Listagem do diretório com filtro:
/home/rinaldo/arquivos/arquivo1.txt
/home/rinaldo/arquivos/arquivo3.txt
/home/rinaldo/arquivos/arquivo2.txt
```

Perceba que o `list` não apresenta elementos dos subdiretórios.

5. É possível recuperar todas as linhas de um arquivo como um *Stream* utilizando o método `lines`.

```
String userHome = System.getProperty("user.home");
Path path = Paths.get(userHome, "arquivos", "subpasta1", "arquivo11.txt");
System.out.println("Path: " + path);

try {
    System.out.println("\nConteúdo do arquivo:");
    Files.lines(path) // recupera todas as linhas do arquivo como Stream
        .forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}

try {
    System.out.println("\nConteúdo do arquivo maior que 2:");
    Files.lines(path)
        .filter(s -> Integer.parseInt(s) > 2) // filtra maior que 2
        .forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
Path: /home/rinaldo/arquivos/subpasta1/arquivo11.txt
```

```
Conteúdo do arquivo:
```

```
1  
2  
3  
4  
5
```

```
Conteúdo do arquivo:
```

```
3  
4  
5
```

- Presenting the New Stream Methods

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 486). Wiley. Edição do Kindle.

- [List Files in a Directory in Java.](#)
- [Class Files.](#) Java Platform SE 7.

WatchService

Objetivo

Observe the changes in a directory by using the WatchService interface

-

Observar as mudanças em um diretório a partir da utilização da Interface WatchService

O WatchService é uma API para monitorar mudanças em arquivos e diretórios. Serão apresentadas as principais formas de realizar essa monitoração.

Para utilizar a API são necessárias 4 classes principais:

- WatchService → representa o serviço em si de monitoração;
 - StandardWatchEventKinds → representa os tipos de alteração que se deseja monitorar: criar, apagar ou modificar;
 - WatchKey → representa um retorno do serviço informando que houveram alterações;
 - WatchEvent → representa um evento em si, onde é possível obter informações do que foi alterado.
1. É possível observar criações ou deleções de arquivos em um diretório.

```
String userHome = System.getProperty("user.home");
Path path = Paths.get(userHome, "arquivos");
System.out.println("Path: " + path);

// criação do WatchService - ainda sem monitorar nada
try (WatchService service = FileSystems.getDefault().newWatchService()) {

    // registro do WatchService no Path para monitorar os evento de CREATE e DELETE
    path.register(service, StandardWatchEventKinds.ENTRY_CREATE,
    StandardWatchEventKinds.ENTRY_DELETE);

    while (true) { // loop infinito
        // take() irá retornar sempre que houverem eventos
        // caso contrário a chamada fica parada esperando eventos ocorrerem
        WatchKey key = service.take();
        List<WatchEvent<?>> pollEvents = key.pollEvents(); // recupera os eventos
ocorridos
        System.out.println("Eventos capturados. Quantidade: " + pollEvents.size());
        for (WatchEvent<?> event : pollEvents) { // iteração sobre todos os eventos
recuperados
            System.out.println("Evento ocorrido. Tipo : " + event.kind() + ". Contexto: "
+ event.context());
        }
        key.reset(); // reseta o WatchKey para que possa ser utilizado novamente
    }
} catch (IOException | InterruptedException e) {
    e.printStackTrace();
}
```

Saída no console

```
Path: /home/rinaldo/arquivos
Eventos capturados. Quantidade: 1
Evento ocorrido. Tipo : ENTRY_DELETE. Contexto: arquivo1.txt
Eventos capturados. Quantidade: 1
Evento ocorrido. Tipo : ENTRY_CREATE. Contexto: arquivo1.txt
```

Isso é o que seria impresso no console caso o **arquivo1.txt** fosse apagado e depois criado novamente.

Perceba os passos que foram feitos:

- Um WatchService foi criado
- O service foi registrado no **Path** com os eventos desejados
- Foi criado um *loop* infinito para realizar a monitoração de forma contínua
- Foi chamado o método **take**, que aguarda até haver eventos e assim, retorná-los

e. Foi chamado o método `pollEvents` para recuperar os eventos que ocorreram

f. Os eventos foram impressos no console

g. O `WatchKey` foi resetado para que pudesse ser utilizado novamente

Esse é o básico de um `WatchService`. Perceba que ele é um recurso que deve ser fechado, por isso está na sintaxe de `try-with-resources`.

2. É possível monitorar mudanças em arquivos de um diretório.

`src/org/j6toj8/fileio/watchservice/WatchService_Modify.java`

```
String userHome = System.getProperty("user.home");
Path path = Paths.get(userHome, "arquivos");
System.out.println("Path: " + path);

// criação do WatchService - ainda sem monitorar nada
try (WatchService service = FileSystems.getDefault().newWatchService()) {

    // registro do WatchService no Path para monitorar o evento de MODIFY
    path.register(service, StandardWatchEventKinds.ENTRY_MODIFY);

    while (true) { // loop infinito
        // take() irá retornar sempre que houverem eventos
        // caso contrário a chamada fica parada esperando eventos ocorrerem
        WatchKey key = service.take();
        List<WatchEvent<?>> pollEvents = key.pollEvents(); // recupera os eventos
        ocorridos
        System.out.println("Eventos capturados. Quantidade: " + pollEvents.size());
        for (WatchEvent<?> event : pollEvents) { // iteração sobre todos os eventos
            recuperados
                System.out.println("Evento ocorrido. Tipo : " + event.kind() + ". Contexto: "
+ event.context());
            }
        key.reset(); // reseta o WatchKey para que possa ser utilizado novamente
    }
} catch (IOException | InterruptedException e) {
    e.printStackTrace();
}
```

Saída no console

```
Path: /home/rinaldo/arquivos
Eventos capturados. Quantidade: 1
Evento ocorrido. Tipo : ENTRY_MODIFY. Contexto: .arquivo1.txt.kate-swp
Eventos capturados. Quantidade: 1
Evento ocorrido. Tipo : ENTRY_MODIFY. Contexto: arquivo1.txt.h26197
Eventos capturados. Quantidade: 1
Evento ocorrido. Tipo : ENTRY_MODIFY. Contexto: arquivo1.txt.h26197
```

Esses foram os eventos que ocorreram ao abrir o `arquivo1.txt` com o editor `Kate`, acrescentar um caracter, e salvar o arquivo.

3. Não é possível monitorar diretamente um arquivo.

`src/org/j6toj8/fileio/watchservice/WatchService_File.java`

```
String userHome = System.getProperty("user.home");
Path path = Paths.get(userHome, "arquivos", "arquivo1.txt"); // NÃO SUPORTADO
System.out.println("Path: " + path);

try (WatchService service = FileSystems.getDefault().newWatchService()) {
    path.register(service, StandardWatchEventKinds.ENTRY_MODIFY); // LANÇA EXCEÇÃO
} catch (IOException e) {
    e.printStackTrace();
}
```

Saída no console

```
Path: /home/rinaldo/arquivos/arquivo1.txt
java.nio.file.NotDirectoryException: /home/rinaldo/arquivos/arquivo1.txt
    at sun.nio.fs.LinuxWatchService$Poller.implRegister(LinuxWatchService.java:249)
    at sun.nio.fs.AbstractPoller.processRequests(AbstractPoller.java:260)
    at sun.nio.fs.LinuxWatchService$Poller.run(LinuxWatchService.java:364)
    at java.lang.Thread.run(Thread.java:748)
```

Perceba que ocorre exceção ao tentar monitorar diretamente o `arquivo1.txt`.

4. É possível recuperar um `WatchKey` imediatamente ou aguardar um período específico com os métodos `poll`.

```
String userHome = System.getProperty("user.home");
Path path = Paths.get(userHome, "arquivos");
System.out.println("Path: " + path);

try (WatchService service = FileSystems.getDefault().newWatchService()) {
    path.register(service, StandardWatchEventKinds.ENTRY_CREATE,
    StandardWatchEventKinds.ENTRY_DELETE);

    System.out.println("Horário antes do poll sem timeout: " + LocalTime.now());
    WatchKey key1 = service.poll(); // retorna imediatamente, mesmo que não haja
    evento
    System.out.println("WatchKey do poll: " + key1);
    System.out.println("Horário depois do poll sem timeout: " + LocalTime.now());

    System.out.println("Horário antes do poll com timeout: " + LocalTime.now());
    WatchKey key2 = service.poll(5, TimeUnit.SECONDS); // retorna após 5 segundos,
    mesmo que não haja evento
    System.out.println("WatchKey do poll com timeout: " + key2);
    System.out.println("Horário depois do poll com timeout: " + LocalTime.now());
} catch (IOException | InterruptedException e) {
    e.printStackTrace();
}
```

Saída no console

```
Path: /home/rinaldo/arquivos
Horário antes do poll sem timeout: 14:55:10.298
WatchKey do poll: null
Horário depois do poll sem timeout: 14:55:10.298
Horário antes do poll com timeout: 14:55:10.298
WatchKey do poll com timeout: null
Horário depois do poll com timeout: 14:55:15.300
```

Perceba que o primeiro `poll` retorna imediatamente, mesmo que nenhum evento tenha ocorrido. Já o segundo aguarda por 5 segundos para retornar, mesmo que não haja evento.

Nos cenários de monitoração, o ideal é utilizar o `take`, caso contrário seria necessário invocar o `poll` inúmeras vezes, enquanto o `take` apenas aguarda indefinidamente até que haja um evento.

- Monitoring a Directory for Changes

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 625). Wiley. Edição do Kindle.

- [A Guide to WatchService in Java NIO2.](#)
- [Class Files.](#) Java Platform SE 7.
- [Watching a Directory for Changes.](#) The Java™ Tutorials.

Java Collections

As seções de **Expressões Lambda** e **Streams** já possuem bastante conteúdo sobre os tópicos dos objetivos deste capítulo, e as explicações em detalhes podem ser encontradas lá. Aqui serão apresentados apenas exemplos adicionais especificamente em coleções (*Collections*), pois alguns exemplos das outras seções utilizam outros tipos de *Streams*.

Diamond Operator (Operador Diamante)

Objetivo

Develop code that uses diamond with generic declarations

-

Desenvolver código que usa o diamond (diamante) com declarações de generics

O Diamond Operator (ou Operador Diamante) foi criado no Java 7 para remover código desnecessário ao declarar classes que usam **Generics** (ou tipos genéricos). Abaixo um exemplo que é possível omitir o tipo de algumas classes utilizando o *Diamond Operator*.

src/org/j6toj8/collections/diamond/Collections_Diamond.java

```
List<String> l1 = new ArrayList<String>(); // sem diamond
List<String> l2 = new ArrayList<>(); // com diamond
List<> l3 = new ArrayList<String>(); // NÃO COMPILA - diamond só pode ser utilizado do lado direito

Map<String, String> m1 = new HashMap<String, String>(); // sem diamond
Map<String, String> m2 = new HashMap<>(); // com diamond
Map<> m3 = new HashMap<String, String>(); // NÃO COMPILA - diamond só do lado direito

Map<List<String>, List<String>> m4 = new HashMap<List<String>, List<String>>(); // sem diamond
Map<List<String>, List<String>> m5 = new HashMap<>(); // com diamond
Map<List<String>, List<String>> m6 = new HashMap<<>, <>>(); // NÃO COMPILA - a única sintaxe válida é <>
Map<List<String>, List<String>> m7 = new HashMap<List<String>, <>>(); // NÃO COMPILA - a única sintaxe válida é <>

Map<Map<List<String>, List<String>>, Map<List<String>, List<String>>> m8 = new
HashMap<>(); // com diamond

Map<> m9 = new HashMap<>(); // NÃO COMPILA - é necessário informar o tipo do lado esquerdo
```

- Using the Diamond Operator

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 596). Wiley. Edição do Kindle.

- [Guide to the Diamond Operator in Java.](#)

Collections e lambda

Objetivo

Develop code that iterates a collection, filters a collection, and sorts a collection by using lambda expressions

- Desenvolver código que itera uma coleção, filtra uma coleção, e classifica em ordem uma coleção utilizando expressões lambda

1. É possível iterar diretamente sobre uma coleção utilizando forEach.

src/org/j6toj8/collections/lambda/CollectionsLambda_ForEach.java

```
List<Integer> list = Arrays.asList(1, 6, 7, 2, 9);
list.forEach(System.out::println);
```

Saída no console

```
1
6
7
2
9
```

2. É possível filtrar a coleção.

src/org/j6toj8/collections/lambda/CollectionsLambda_Filter.java

```
List<Integer> list = Arrays.asList(1, 6, 7, 2, 9);
list.stream()
    .filter(n -> n > 2)
    .forEach(System.out::println);
```

Saída no console

```
6
7
9
```

3. É possível classificar em ordem a coleção.

src/org/j6toj8/collections/lambda/CollectionsLambda_Sort.java

```
List<Integer> list = Arrays.asList(1, 6, 7, 2, 9);
list.stream()
    .sorted()
    .forEach(System.out::println);
```

Saída no console

```
1
2
6
7
9
```

4. É possível combinar as operações.

src/org/j6toj8/collections/lambda/CollectionsLambda_Combined.java

```
List<Integer> list = Arrays.asList(1, 6, 7, 2, 9, 54, 13, 87, 23, 97, 34, 17, 34,
89, 35, 26);
list.stream()
    .sorted()
    .filter(n -> n > 10)
    .filter(n -> n % 2 == 0)
    .forEach(System.out::println);
```

Saída no console

```
26
34
34
54
```

- Using Streams

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 185). Wiley. Edição do Kindle.

- [The Java 8 Stream API Tutorial](#).

Buscar por dados

Objetivo

Search for data by using methods, such as `findFirst()`, `findAny()`, `anyMatch()`, `allMatch()`, and `noneMatch()`

-
Buscar por dados utilizando métodos, como `findFirst()`, `findAny()`, `anyMatch()`, `allMatch()`, e `noneMatch()`

1. É possível recuperar o primeiro ou um elemento qualquer da coleção.

src/org/j6toj8/collections/datasearch/DataSearch_FindFirstAny.java

```
List<Integer> list = Arrays.asList(1, 6, 2, 9, 54, 13, 87, 23, 97, 34, 17, 34);

list.parallelStream()
    .findFirst()
    .ifPresent(n -> System.out.println("First: " + n));

list.parallelStream()
    .findAny()
    .ifPresent(n -> System.out.println("Any: " + n));
```

Saída no console

```
First: 1
Any: 9
```

2. É possível verificar se os elementos da coleção atendem ou não a algum critério.

src/org/j6toj8/collections/datasearch/DataSearch_Match.java

```
List<Integer> list = Arrays.asList(1, 6, 2, 9, 54, 13, 87, 23, 97, 34, 17, 34);

boolean anyMatch = list.stream().anyMatch(n -> n > 50);
System.out.println("anyMatch: " + anyMatch);

boolean allMatch = list.stream().allMatch(n -> n > 50);
System.out.println("allMatch: " + allMatch);

boolean noneMatch = list.stream().noneMatch(n -> n > 50);
System.out.println("noneMatch: " + noneMatch);
```

Saída no console

```
anyMatch: true
allMatch: false
noneMatch: false
```

- Using Streams

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 185). Wiley. Edição do Kindle.

- [The Java 8 Stream API Tutorial](#).

Fazendo cálculos e coletando resultados de Streams

Objetivo

Perform calculations on Java Streams by using count, max, min, average, and sum methods and save results to a collection by using the collect method and Collector class, including the averagingDouble, groupingBy, joining, partitioningBy methods

- Realizar cálculos em Streams usando os métodos count, max, min, average, e sum e salvar resultados em uma coleção usando o método collect e a classe Collector, incluindo os métodos averagingDouble, groupingBy, joining, partitioningBy

1. É possível pegar o maior ou menor valor, ou a quantidade de elementos da coleção.

src/org/j6toj8/collections/calculations/Collections_MaxMinCount.java

```
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9);

OptionalInt max = list.stream()
    .mapToInt(Integer::intValue) // transforma para int
    .max();
System.out.println("Max: " + max.getAsInt());

OptionalInt min = list.stream()
    .mapToInt(Integer::intValue) // transforma para int
    .min();
System.out.println("Min: " + min.getAsInt());

long count = list.stream()
    .mapToInt(Integer::intValue) // transforma para int
    .count();
System.out.println("Count: " + count);
```

Saída no console

```
Max: 9
Min: 1
Count: 9
```

2. É possível pegar a média dos valores da coleção.

src/org/j6toj8/collections/calculations/Collections_AveragingDouble.java

```
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9);  
  
Double media = list.stream()  
    .collect(Collectors.averagingDouble(n -> n.doubleValue()));  
  
System.out.println("Média: " + media);
```

Saída no console

```
Média: 5.0
```

3. É possível agrupar os valores da coleção por uma regra específica.

src/org/j6toj8/collections/calculations/Collections_GroupingBy.java

```
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9);  
  
Map<Integer, List<Integer>> mapaDivisaoPor3 = list.stream()  
    .collect(Collectors.groupingBy(n -> n % 3));  
  
System.out.println("Mapa de resto da divisão por 3: " + mapaDivisaoPor3);
```

Saída no console

```
Mapa de resto da divisão por 3: {0=[3, 6, 9], 1=[1, 4, 7], 2=[2, 5, 8]}
```

4. É possível concatenar os valores da coleção.

src/org/j6toj8/collections/calculations/Collections_Joining.java

```
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9);  
  
String juncao = list.stream()  
    .map(n -> n.toString())  
    .collect(Collectors.joining());  
  
System.out.println("Junção dos valores como String: " + juncao);
```

Saída no console

```
Junção dos valores como String: 123456789
```

5. É possível separar os valores da coleção em um mapa com chaves **true** e **false**, de acordo com uma função lambda.

src/org/j6toj8/collections/calculations/Collections_PartitioningBy.java

```
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9);  
  
Map<Boolean, List<Integer>> mapaParImpar = list.stream()  
    .collect(Collectors.partitioningBy(n -> n % 2 == 0));  
  
System.out.println("Mapa de pares e ímpares: " + mapaParImpar);
```

Saída no console

```
Mapa de pares e ímpares: {false=[1, 3, 5, 7, 9], true=[2, 4, 6, 8]}
```

- Using Streams

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 185). Wiley. Edição do Kindle.

- [The Java 8 Stream API Tutorial](#).

Melhorias de Java 8 em Coleções

Objetivo

Develop code that uses Java SE 8 collection improvements, including the `Collection.removeIf()`, `List.replaceAll()`, `Map.computeIfAbsent()`, and `Map.computeIfPresent()` methods

- Desenvolver código que use as melhorias em coleções do Java SE 8, incluindo os métodos `Collection.removeIf()`, `List.replaceAll()`, `Map.computeIfAbsent()`, e `Map.computeIfPresent()`

O Java 8 trouxe vários métodos em Collections que utilizam como argumento uma função lambda, facilitando algumas operações. Serão apresentados exemplos dos 4 métodos relacionados a esse objetivo.

1. É possível remover um item de uma coleção condicionalmente com uma função lambda.

src/org/j6toj8/collections/improvements/Collections_RemoveIf.java

```
List<Integer> list = new ArrayList<Integer>(Arrays.asList(1,2,3,4,5,6,7,8,9));  
  
System.out.println("Lista antes do removeIf: " + list);  
list.removeIf(n -> n % 2 == 0); // remove números pares  
System.out.println("Lista depois do removeIf: " + list);
```

Saída no console

```
Lista antes do removeIf: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Lista depois do removeIf: [1, 3, 5, 7, 9]
```

2. É possível modificar todos os elementos da coleção de acordo com uma operação lambda.

src/org/j6toj8/collections/improvements/Collections_ReplaceAll.java

```
List<Integer> list = new ArrayList<Integer>(Arrays.asList(1,2,3,4,5,6,7,8,9));

System.out.println("Lista antes do replaceAll: " + list);
list.replaceAll(n -> n * 2); // multiplica todos os elementos por 2
System.out.println("Lista depois do replaceAll: " + list);
```

Saída no console

```
Lista antes do replaceAll: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Lista depois do replaceAll: [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

3. É possível colocar valores em um **Map** a partir de uma função lambda, apenas se a chave **não** estiver presente no **Map**.

src/org/j6toj8/collections/improvements/Collections_ComputeIfAbsent.java

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("A", "A".hashCode());
map.put("B", "B".hashCode());

System.out.println("Map antes do computeIfAbsent: " + map);
map.computeIfAbsent("A", k -> k.hashCode());
map.computeIfAbsent("C", k -> k.hashCode());
System.out.println("Map depois do computeIfAbsent: " + map);
```

Saída no console

```
Map antes do computeIfAbsent: {A=65, B=66}
Map depois do computeIfAbsent: {A=65, B=66, C=67}
```

4. É possível alterar valores em um **Map** a partir de uma função lambda, apenas se a chave estiver presente no **Map**.

```
HashMap<String, Integer> map = new HashMap<String, Integer>();
map.put("A", "A".hashCode());
map.put("B", "B".hashCode());

System.out.println("Map antes do computeIfPresent: " + map);
// k = chave; v = valor
map.computeIfPresent("A", (k, v) -> k.hashCode() * v);
map.computeIfPresent("B", (k, v) -> k.hashCode() * v);
map.computeIfPresent("C", (k, v) -> k.hashCode() * v);
System.out.println("Map depois do computeIfPresent: " + map);
```

Saída no console

```
Map antes do computeIfPresent: {A=65, B=66}
Map depois do computeIfPresent: {A=4225, B=4356}
```

- Using Streams

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 185). Wiley. Edição do Kindle.

- [The Java 8 Stream API Tutorial](#).

Maps e Streams

Objetivo

Develop code that uses the `merge()`, `flatMap()`, and `map()` methods on Java Streams

- Desenvolver código que usa os métodos `merge()`, `flatMap()`, e `map()` em Streams Java.

Dos objetivos desta seção, apenas o método `merge` ainda não foi apresentado em outras seções.

1. É possível colocar um novo valor em um mapa, ou alterar o valor que já estava presente, utilizando o método `merge`.

src/org/j6toj8/collections/mergemap/Collections_Merge.java

```
HashMap<Integer, String> map = new HashMap<Integer, String>();
map.put(1, "String1-");
map.put(2, "String2-");

System.out.println("Map antes do merge: " + map);
map.merge(1, "StringA", (v1, v2) -> v1.concat(v2));
map.merge(2, "StringB", (v1, v2) -> v1.concat(v2));
map.merge(3, "StringC", (v1, v2) -> v1.concat(v2));
map.merge(4, "StringD", (v1, v2) -> v1.concat(v2));
System.out.println("Map depois do merge: " + map);
```

Saída no console

```
Map antes do merge: {1=String1-, 2=String2-}
Map depois do merge: {1=String1-StringA, 2=String2-StringB, 3=StringC, 4=StringD}
```

Perceba que, para as chaves que já estavam presentes no **Map**, foi aplicada a função lambda. Para as chaves que ainda não estavam presentes, foi apenas inserida a **String** passada como valor.

2. É possível transformar valores em uma coleção com o método **map**.

src/org/j6toj8/collections/mergemap/Collections_Map.java

```
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,8,9);

list.stream()
    .map(e -> e * 2)
    .forEach(System.out::println);
```

Saída no console

```
2
4
6
8
10
12
14
16
18
```

3. É possível percorrer outro Stream, em sequência com o Stream atual, utilizando o método **flatMap**.

```
List<String> list = Arrays.asList("Manoel");

System.out.println("\n Com map: ");
// com map - as letras da String viram
// um Stream dentro de outro Stream
list.stream()
    .map(s -> Arrays.stream(s.split ""))
    .forEach(System.out::println);

System.out.println("\n Com flatMap: ");
// com flatMap - as letras da String viram dados
// do próprio Stream
list.stream()
    .flatMap(s -> Arrays.stream(s.split ""))
    .forEach(System.out::println);
```

Saída no console

```
Com map:
java.util.stream.ReferencePipeline$Head@e9e54c2

Com flatMap:
M
a
n
o
e
l
```

Perceba que uma transformação que resulta em outro Stream é percorrida como se fosse o próprio Stream original.

- Additions in Java 8

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 152). Wiley. Edição do Kindle.

- Using Streams

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide (p. 185). Wiley. Edição do Kindle.

- [The Java 8 Stream API Tutorial](#).

- [Merging Two Maps with Java 8](#).

Assume the following

- **Ausência das instruções de pacote e importação:**

Se o código de amostra não incluir instruções de pacote ou de importação, e a pergunta não se referir explicitamente a essas instruções ausentes, suponha que toda a amostra de código esteja no mesmo pacote ou que existam instruções de importação para suportá-las.

- **Nenhum nome de arquivo ou caminho de diretório para classes:**

Se uma pergunta não indicar os nomes de arquivo ou os locais de diretório das classes, assuma um dos seguintes, o que permitirá que o código seja compilado e executado:

- Todas as classes estão em um arquivo
- Cada classe está contida em um arquivo separado e todos os arquivos estão em um diretório

- **Quebras de linha não intencionais:**

O código de exemplo pode ter quebras de linha não intencionais. Se você vir uma linha de código que parece ter sido quebrada, e isso criar uma situação em que a quebra de linha é significativa (por exemplo, um literal de String citado foi dividido), suponha que a quebra de linha seja uma extensão da mesma linha e não contém um retorno explícito (*carriage return*) que causaria uma falha de compilação.

- **Fragments de código:**

Um fragmento de código é uma pequena seção do código-fonte que é apresentada sem seu contexto. Suponha que todo o código de suporte necessário exista e que o ambiente ofereça suporte completo à compilação e execução corretas do código mostrado e seu ambiente omitido.

- **Comentários descritivos:**

Considere comentários descritivos, como "setter e getters vão aqui", pelo valor nominal. Suponha que o código correto exista, compile e execute com êxito para criar o efeito descrito.

Apêndice A: Dicas para ter sucesso na prova!

Além de validar conceitos importantes da linguagem Java a prova de certificação também exigirá que você esteja atento a detalhes específicos nos códigos de cada questão. Sem o auxílio da IDE, você será o compilador.

Veja alguns exemplos simples que se mostram óbvios mas podem te enganar na hora de marcar a resposta certa.

Cenário 1

```
class Duke {  
    static int version = 6;  
  
    public static void main(String[] args){  
        System.out.println("Java" + version)  
        version = version + 2;  
    }  
}
```

1. Compila e imprime "Java6"
2. Compila e imprime "Java8"
3. Não compila
4. Compila, mas ocorre um erro durante a execução

Sem pensar muito a resposta seria a *opção 1*, certo? Errado. Olhando mais atentamente ao código é possível notar que na primeira linha do método `main` está faltando um ; (ponto-e-vírgula). Esse pequeno detalhe mostra que a opção certa é a 3.

Sempre que existir uma resposta falando *código não compila*, verifique duas vezes as regras de compilação antes de testar o comportamento do código e sua possível resposta.

Checklist mental para validar uma compilação



- Ponto-e-vírgula
- Visibilidade
- Escopo de variáveis
- Nomes e parâmetros de métodos
- ...

Resposta: 3

Cenário 2

```
class Duke {  
    static int version = 6;  
  
    public static void main(String[] version) {  
        version = 8;  
        System.out.println("Java" + version);  
    }  
}
```

1. O código compila e roda, imprimindo "Java8".
2. O código não compila.
3. O código compila e roda, imprimindo "Java6".
4. O código compila mas dá erro em execução.

Se você escolheu a opção 1, você errou... Esse exemplo tem outra pegadinha com o conceito de *shadowing*. Usa-se o mesmo nome de variável mas com um escopo diferente. Inicialmente o tipo `int` engana sua resposta mas esse código não compila ao tentar atribuir um valor `int` à uma variável do tipo `String[]`.

Resposta: 2

Cenário 3

```
class Duke {  
  
    public static void main(String[] args) {  
        boolean dukeClones = new boolean[100];  
        System.out.println(dukeClones[10]);  
    }  
}
```

1. Imprime 1
2. Imprime 0
3. Imprime `false`
4. Imprime `true`
5. Imprime `null`
6. Erro de execução
7. Não compila

A escolha mais comum seria a opção 3, onde confirma que o valor padrão de cada posição de um array do tipo *boolean* é *false*. Esta opção estaria certa, se não fosse uma pegadinha. Este código na verdade **não compila**. A opção certa seria a número 7. Isso porque a variável *dukeClones* é um *boolean* simples tentando receber um array do tipo *boolean*.

Em uma inicialização implícita como membro de uma classe, ou cada posição de um array, etc, a variável recebe o valor padrão respeitando a seguinte regra:



- *boolean* → ***false***
- *char* → **vazio**, equivalente a 0
- Primitivos numéricos inteiros → **0**
- Primitivos numéricos com ponto flutuante → **0.0**
- Referências (Objetos) → **null**

Resposta: 7

Apêndice B: Teste seu conhecimento!

Hora de colocar em prática tudo que foi visto neste livro. Existem algumas opções gratuitas e várias opções pagas. A opção mais interessante para ambos os casos é a **Whizlabs**.

Todos os testes listados abaixo são em inglês, provavelmente o mesmo idioma que você fará seu teste.



Todas as opções são referentes à prova *1Z0-813* (atualização do Java 6 ou inferior para o Java 8).

Gratuito

Link	Qtd. Questões
Whizlabs	25
Oracle	8

Pagos

Link	Qtd. Questões	Preço
Whizlabs (Recomendado)	360	\$9.95
Kaplan IT Training (30-180 dias)	?	\$89
Oracle (30 dias)	?	R\$ 169,00
Oracle (180 dias)	?	R\$ 290,00
Exam Collection	61	\$69.99
Pass-Guaranteed.com	?	?

Apêndice C: Referências

Em todas sessões do livro são feitas referências diretas as fontes de inspiração ou argumentação base para a criação deste conteúdo. A seguir o resumo das principais e de outras referências que também fizeram parte dessa jornada de aprendizado.

Para criação da estrutura do projeto deste livro foi utilizado como base o projeto [ebook-with-asciidoc](#).



Boa parte das fontes estão em inglês (**en-US**), mas algumas também podem ser encontradas em português (**pt-BR**).

- **(en-US)** Excelente livro que mostra o passo-a-passo para tirar a certificação completa do Java 8 ou atualizar para a versão 8, objetivo deste livro.

Boyarsky, Jeanne; Selikoff, Scott. OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide: Exam 1Z0-809 (English Edition) 1st Edition. Wiley. Edição do Kindle.

[Link na loja Amazon Brasil.](#)

- **(en-US)** Blog do Eugen (Baeldung) com dezenas de artigos focados em Java, entre outros assuntos.

<https://www.baeldung.com/category/java/>

- **(en-US)** Tutoriais da própria Oracle com explicações detalhadas da linguagem Java.

<https://docs.oracle.com/javase/tutorial/java/>

- **(en-US)** Documentação oficial do Java.

<https://docs.oracle.com/javase/8/docs/>

- **(pt-BR)** Curso da Alura, que apesar de focar no Java 7 (ao menos no momento da escrita desse texto), mostra como não cair em pegadinhas na prova. Principalmente em relação à situações que exigem pensar como um compilador e apontar a falta de um ;. São mais de 80 horas de conteúdo. Vale muito a pena para quem quer ir a fundo de cada detalhe da linguagem (pelo menos até a versão 7).

<https://www.alura.com.br/formacao-certificacao-java>

Material Complementar

- **(pt-BR)** Canal do YouTube [RinaldoDev](#) com vídeos explicativos de diversos conceitos da linguagem Java.
- **(en-US)** [Java Challengers](#), uma playlist no Youtube do canal do [Rafael Del Nero](#) que explica desafios de Java para ensinar conceitos importantes da linguagem.
- **(en-US)** [Oracle Dev Gym](#), desafios online e gratuitos para testar suas habilidades de Java.