

**easy  
java  
Magazine**

Edição 48

Certificação Java

Uma visão geral sobre a  
tradicional OCAJP8, antiga SCJP



# ECLIPSE LUNA

Conheça os novos recursos na prática

ISSN 2179625-4



# Sumário

## Introdução à Certificação de Programador Java

[ John Soldera ]

### Conteúdo sobre Novidades

#### Eclipse Luna: Sua IDE pronta para o Java 8

[ Wellington Ferro ]

### Conteúdo sobre Boas Práticas

#### Design Patterns: aplicando padrões de projetos

[ Arthur Gonçalves Gomes Junior ]

## CURSOS ONLINE

A Revista Java Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.



### CONHEÇA ALGUNS DOS CURSOS:

- Curso de noSQL (Redis) com Java
- Curso Básico de JDBC
- Java Básico: Aplicações Desktop
- JSF com Primefaces
- Conhecendo o Apache Struts

Para mais informações :

[www.devmedia.com.br/curso/javamagazine](http://www.devmedia.com.br/curso/javamagazine)

(21) 3382-5038



[www.devmedia.com.br](http://www.devmedia.com.br)

[versão para impressão]

Link original:

<http://www.devmedia.com.br/articles/viewcomp.asp?comp=32166>

# Introdução à Certificação de Programador Java

Este artigo apresenta uma visão geral sobre a tradicional certificação Java OCAJP8, incluindo questões de exemplo.

---

## Fique por dentro

Esse artigo apresenta uma visão geral sobre a certificação para programador Java OCAJP8 e introduz os principais conhecimentos necessários à realização do exame, incluindo 10 exemplos de perguntas baseadas no conteúdo que pode ser cobrado na prova oficial da certificação. As respostas dessas

questões desafiadoras são comentadas em detalhes. Além disso, esse artigo discorre sobre informações importantes da prova, como tempo e pontuação necessária para aprovação em um dos exames mais procurados da Oracle.

A certificação de programador OCAJP8, voltada para a plataforma Java SE 8 (*Java Standard Edition 8*), permite ao programador Java confirmar o seu conhecimento nas bases do desenvolvimento Java, dominando a maior parte das funcionalidades do Java SE, que atualmente está na versão 8. A certificação para programador Java OCAJP8 é uma das mais requisitadas, visto que existe uma alta demanda de programadores Java no mercado de trabalho.

Essa certificação confere ao profissional o título de *Oracle Certified Associate*, sendo a mais básica oferecida pela Oracle. O exame é composto por 70 questões de múltipla escolha e o candidato tem 120 minutos para respondê-lo. Para obter a certificação é preciso acertar 63% das questões, sendo elas voltadas aos conceitos básicos da linguagem pertencentes ao Java SE 8 (que também abrange as versões anteriores do Java SE).

Para conhecimento, antigamente essa certificação era chamada de *Sun Certified Java Programmer* (SCJP), mas depois que a Sun foi comprada pela Oracle a certificação de programador Java passou a ser chamada de *Oracle Certified Associate Java SE Programmer* (OCAJP). Além dessa, é disponibilizada uma versão da certificação de programador que inclui conceitos mais abrangentes, relativos ao nível *Oracle Certified Professional*, obtida com a aprovação na prova *Oracle's Certified Professional Java SE Programmer* (OCPJP).

Felizmente, existem cursos oferecidos pela própria Oracle para o programador se preparar para qualquer certificação Java. Entretanto, o candidato pode também se preparar lendo livros específicos para a certificação, artigos ou ainda estudando os livros que ensinam a linguagem Java e testando exemplos no próprio JDK.

Através da certificação OCAJP8, o programador confirma o seu domínio sobre os fundamentos da linguagem Java, sendo avaliado em uma grande variedade de funcionalidades das APIs padrões e nos principais recursos da linguagem em si incluídos no Java SE 8. A prova aplicada na certificação inicia com temas básicos sobre laços de repetição e declaração de variáveis, e segue até tópicos mais complexos, como **Arrays** e **Exceções**.

Como a prova envolve o básico, ela não cobre tópicos específicos como criação de GUIs e programação para Internet, mas o programador deve mostrar no exame que domina a maioria das bibliotecas padrão (APIs básicas). Além da teoria, o programador deve demonstrar que domina a prática, compreendendo trechos de programas criados usando os mecanismos de controle da linguagem, sem ter de necessariamente produzir código.

# Conteúdo da certificação OCAJP8

O conteúdo da prova abrange diversos conceitos fundamentais da programação Java, iniciando em fundamentos como definição de variáveis, tipos de dados e uso de operadores, e alcançando assuntos mais complexos como herança, encapsulamento e polimorfismo.

Os tópicos da prova são apresentados a seguir:

- Conceitos básicos da linguagem Java:
  - o Definir o escopo das variáveis;
  - o Definir a estrutura das classes Java;
  - o Criar aplicações Java executáveis a partir do uso do método **main()**. Executar um programa Java no *console* e interpretar a saída do mesmo;
  - o Importar outros pacotes Java para fazê-los acessíveis em seu código;
  - o Conhecer as características principais da linguagem Java, como independência de plataforma, orientação a objetos, sintaxe simplificada, etc.
- Trabalhar com tipos de dados Java:
  - o Declarar e inicializar variáveis, incluindo conversão (*casting*) de tipos primitivos;
  - o Saber diferenciar variáveis primitivas e variáveis de referência a objetos;
  - o Saber como ler ou alterar campos de objetos;
  - o Explicar o ciclo de vida dos objetos (criação, gerenciar referências e coleta de lixo);
  - o Desenvolver código que usa classes *wrapper*, como **Boolean** e **Integer**.
- Usar operadores e comandos de decisão:

- o Usar os operadores Java e parênteses para sobrescrever a precedência de operadores;
- o Testar a igualdade entre *strings* e outros objetos usando `==` e o método **equals()**;
- o Criar blocos de comandos com *if* e *else*, incluindo o operador ternário;
- o Usar o comando *switch*.
- Criar e usar *arrays*:
  - o Declarar, instanciar e inicializar um *array* unidimensional;
  - o Declarar, instanciar e inicializar um *array* multidimensional.
- Usar laços de repetição:
  - o Criar e usar laços **while**;
  - o Criar e usar laços **for** incluindo o **for** melhorado;
  - o Criar e usar laços **do-while**;
  - o Comparar estruturas de repetição;
  - o Usar os comandos **break** e **continue**.
- Trabalhando com métodos e encapsulamento:
  - o Criar métodos com argumentos e valores de retorno, incluindo métodos sobrecarregados;
  - o Aplicar a palavra-chave **static** a métodos e variáveis;
  - o Criar e sobrepor construtores e conhecer a sua interação com construtores **default**;
  - o Aplicar modificadores de acesso;
  - o Aplicar os princípios de encapsulamento em classes;
  - o Determinar o efeito sobre referências a objetos ou valores primitivos quando os mesmos são passados como parâmetro a métodos que mudam os valores de tais variáveis.
- Trabalhar com Herança:
  - o Descrever herança e seus benefícios;

- o Desenvolver código que demonstra o uso de polimorfismo, incluindo sobrescrita de métodos e variáveis e diferenciar entre o tipo de uma referência e o tipo de um objeto;
  - o Determinar quando *casting* é necessário;
  - o Usar *super* e *this* para acessar objetos e construtores;
  - o Usar classes abstratas e interfaces.
- Tratamento de Exceções:
    - o Diferenciar exceções verificadas, **RuntimeExceptions** e **Errors**;
    - o Usar estruturas **try-catch** e determinar como é o fluxo normal das exceções;
    - o Descrever as vantagens de se usar gerenciamento de exceções;
    - o Criar e invocar métodos que lancem exceções;
    - o Reconhecer classes e categorias comuns de exceções.
  - Trabalhar com um conjunto selecionado de classes:
    - o Manipular dados usando **StringBuilder** e seus métodos;
    - o Criar e manipular Strings;
    - o Criar e manipular as principais classes de calendário (tempo e datas);
    - o Declarar e usar **ArrayList** para um determinado tipo de dados;
    - o Usar expressões lambda.

## Observações importantes sobre a linguagem Java

É muito importante ter o domínio de todos os tópicos mencionados na descrição da certificação, pois grande parte das questões aborda situações especiais que exigem muita atenção. Deste modo, a fim de facilitar o estudo do candidato, alguns comentários muito úteis serão dispostos a seguir, já antecipando algumas das armadilhas comuns inseridas no exame.

Para começar, um tópico básico da certificação diz respeito aos fundamentos que definem a estrutura das classes Java (ou seja, a forma como são organizadas as classes, os métodos, variáveis, classes internas e tudo o mais que compõe uma classe), abrangendo também a assinatura padrão do método **main()**, escopo de variáveis e importação de pacotes.

Quanto à estrutura das classes Java, deve-se saber que um arquivo de código fonte Java pode ter no máximo uma classe pública e que a mesma deve ter o mesmo nome do arquivo *.java*. Outra dica importante, no que se refere a interfaces, é que elas sempre têm métodos públicos, mesmo no caso deles serem declarados sem o modificador de acesso. Além disso, como as interfaces têm a função de estabelecer um contrato de definição de objetos, seus métodos não devem apresentar implementação, pois precisam ser implementados pelas classes que as descendem.

No exame é fundamental saber que a assinatura padrão do método **main()** deve ser respeitada a rigor para que uma classe Java seja executável. No entanto, isso não impede que sejam criados outros métodos **main()** nas classes Java usando assinaturas diferentes.

Diante de tais armadilhas do método **main()**, é necessário ter em mente que a assinatura desse método suporta apenas a alteração do nome do único parâmetro que ele tem (o vetor de *strings*) para ser executável. Deste modo, uma possível questão pode perguntar o que acontece ao tentar executar uma determinada classe que tem um único método **main()** privado. Erro de compilação? Erro de execução? Não executa? Executa normalmente? Não encontra o método **main()** executável? Para dominar situações como essa é importante deixar de lado o conforto de usar IDEs como o Eclipse e o NetBeans e partir para a linha de comando do console, a fim de conhecer as mensagens claramente como são na saída padrão (console) em situações especiais.

Saber compilar mentalmente os programas apresentados na prova e ver se os mesmos contêm erros é vital. Além disso, é recomendado saber determinar se um dado tipo de erro ocorre, pois existem diversos tipos de falhas que podem impedir um programa de chegar até o final, tais como: método **main()** não encontrado, erro de compilação e erro de execução. Sendo assim, é importante saber quais erros levam o processo de compilação a falhar e como são tratados os erros de execução mais comuns, como divisão por zero. Muitas armadilhas podem estar exatamente na questão de saber quando o erro é detectado. Além disso, alguns erros podem não apresentar serem erros de compilação, sendo nesse caso necessário saber como o compilador reage.

De uma forma geral, a prova de certificação Java é conhecida por conter situações bem peculiares que exigem muita atenção.

Mesmo candidatos que trabalham há anos com a linguagem podem nunca ter testado certas situações especiais que são elegíveis de cair prova. Muitas destas situações, como veremos no decorrer do artigo, podem ser criadas a partir dos fundamentos da linguagem Java. Em suma, a prova contém o básico da programação Java, que não necessariamente é simples, e certamente o fará um programador melhor e mais seguro.

Outro conceito básico no qual o candidato deve apresentar domínio é o escopo das variáveis, pois o mesmo influencia diretamente na compilação dos programas. De forma geral, determinar o escopo de uma variável tem relação com determinar o tempo de vida da mesma, pois ela é criada em algum momento e geralmente é acessível durante o intervalo de tempo em que o bloco de código em que ela é declarada segue em execução. Por exemplo, uma variável declarada antes de um laço **for** é visível dentro e fora do mesmo, mas uma variável declarada internamente ao laço **for** é invisível fora do mesmo, ou seja, seu escopo é contido no laço **for**. Uma coisa que pode causar confusão é que em um mesmo método é possível criar várias vezes variáveis com o mesmo nome, mas em escopos diferentes (por exemplo, dentro de laços de repetição diferentes), caso contrário ocorre erro de compilação.

É aconselhado também saber que variáveis locais que ficam fora de escopo são eliminadas automaticamente (por exemplo, quando um método ou laço de repetição termina). De uma forma geral, qualquer objeto que fique fora de escopo e não tenha nenhuma referência para si mesmo será eliminado automaticamente. Entretanto, atribuir **null** propositalmente a uma variável de referência não elimina necessariamente o objeto, pois, como dito, a eliminação ocorre somente quando não existem mais referências àquele objeto. Por outro lado, a eliminação de objetos sem referências não ocorre instantaneamente na linguagem Java, pois dependemos do Garbage Collector para realizar a exclusão real do objeto na memória.

Outro conhecimento necessário é saber diferenciar variáveis locais (internas aos métodos), variáveis de instância e variáveis de classe, pois possuem escopos diferentes. Além disso, dominar a criação e sobrescrita de métodos e atributos é necessário para se aplicar corretamente os conceitos de herança, encapsulamento e polimorfismo. Finalmente, saber usar adequadamente as palavras-chave **this** e **super** é fundamental para resolver muitas das armadilhas comuns no exame.

O candidato deve ter um amplo conhecimento dos modificadores de acesso a métodos, classes e variáveis, onde, por exemplo, o modificador **final** desempenha um papel importante, permitindo que o valor de uma determinada variável seja definido apenas uma vez durante o ciclo de vida da mesma, ou seja, criando constantes. Outro modificador bastante utilizado é o **static**, que pode ser aplicado a atributos, métodos e classes internas. Muitas armadilhas podem ser derivadas desse modificador, mas ele não representará um problema se o candidato souber corretamente diferenciar entre contexto de instância e classe.

Outro assunto que pode parecer muito simples é importar classes ou pacotes na declaração de classes. Porém, como fazer para se utilizar duas classes de mesmo nome que pertencem a pacotes diferentes? Essa é uma situação um tanto rara, mas não impossível, e requer que uma das classes seja acessada sempre pelo nome completo, sem usar **import**.

Portanto, podemos verificar que existe uma boa complexidade nos tópicos básicos da linguagem Java, sendo importante conhecer claramente o ciclo de vida dos objetos, saber diferenciar objeto de referência, diferenciar variáveis primitivas de objetos, conhecer *wrappers*, *strings* e outros tipos imutáveis, saber como ocorre a passagem de parâmetros em métodos (valor ou referência), e todos os demais tópicos mencionados anteriormente, como o uso de estruturas condicionais, de repetição, *arrays* (e também **ArrayList**), conceitos básicos de herança, encapsulamento e polimorfismo, exceções e classes específicas.

Para completar, exceções é um assunto um tanto traíçoeiro e exige bastante estudo, pois um bloco **try-catch** pode conter diversas clausulas **catch** e é necessário saber qual clausula **catch** será processada de acordo com a exceção que foi lançada no **try**. Além disso, existe uma hierarquia de exceções, onde a classe **Exception** tem classes filhas, sendo a própria **Exception** filha de um tipo mais genérico de erro (**Error**). É requerido no exame saber as exceções mais básicas como, por exemplo, determinar como é tratada a divisão por zero ou o acesso ao método de uma variável que tem valor **null**.

Outra informação importante sobre exceções é que existem exceções verificadas e não verificadas (**RuntimeException**), de forma que não é obrigatório que sejam capturadas para tratamento. Uma possível questão na prova pode apresentar um método que lança uma exceção. Porém, se ela for verificada, deve-se obrigatoriamente incluir a palavra-chave **throws** na declaração do método ou tratar com **try-catch**.

## Questões de exemplo para a OCAJP

A seguir serão apresentadas diversas questões de exemplo que têm grandes chances de serem cobradas na certificação Java OCAJP8. Logo após, indicaremos as respostas corretas e analisaremos cada questão.

### Questão 1

Dado o código exposto na **Listagem 1**, qual resultado é o correto?

- A) a b c d
- B) d b a c
- C) a d b c
- D) b d c a
- E) b a c d
- F) d a b c

**Listagem 1.** Código fonte da primeira pergunta.

```
01. public class Letter {  
02.     Letter() { System.out.print("a "); }  
03. }
```

```
03. { System.out.print("b "); }
04. public static void main(String[] args) {
05.     new Letter().go();
06. }
07. void go() { System.out.print("c "); }
08. static { System.out.print("d "); }
09. }
```

## Questão 2

Dado o código exposto na **Listagem 2**, qual resultado é o correto?

- A) A saída é **true** e **MyStuff** segue o contrato de **Object.equals()** e **Object.hashCode()**.
- B) A saída é **false** e **MyStuff** segue o contrato de **Object.equals()** e **Object.hashCode()**.
- C) A saída é **true** e **MyStuff** não segue o contrato de **Object.equals()** e **Object.hashCode()**.
- D) A saída é **false** e **MyStuff** não segue o contrato de **Object.equals()** e **Object.hashCode()**.
- E) A compilação falha.
- F) Uma exceção é lançada em tempo de execução.

**Listagem 2.** Código fonte da segunda pergunta.

```
01. public class Stuff {
02.     Stuff(String n) { name = n; }
03.     String name;
04.     public static void main(String[] args) {
05.         Stuff m1 = new Stuff("Escaninho");
06.         Stuff m2 = new Stuff("Porta Canetas");
```

```
07. System.out.println(m2.equals(m1));
08. }
09. public boolean equals(Object o) {
10. Stuff m = (Stuff) o;
11. if(m.name != null && name != null)
12. return m.name.equals(name);
13. else
14. return false;
15. }
16. }
```

## Questão 3

Dado o código exposto na **Listagem 3**, qual resultado é o correto?

- A) [22, 15, 17]
- B) [22, 22, 15, 17]
- C) [17, 2, 15, 22]
- D) [17, 15, 22, 22]
- E) A compilação falha.
- F) Uma exceção é lançada em tempo de execução.

**Listagem 3.** Código fonte da terceira pergunta.

```
01. import java.util.*;
02. public class Numbers {
03. public static void main(String[] args) {
04. List p = new ArrayList();
05. p.add(17);
06. p.add(22);
07. p.add(15);
08. p.add(2);
```

```
09. p.sort();
10. System.out.println(p);
11. }
12. }
```

## Questão 4

Dado o código exposto na **Listagem 4**, qual resultado é o correto?

- A) 0 Element1 0 Element1 1
- B) 0 Element1 0 Element1 1 Element2 2
- C) 0 Element1 0 Element2 1 Element1 2 Element1 2 Element2
- D) 1 Element1 1 Element2 2 Element1 2 Element1 2 Element2
- E) 0 Element1 0 Element2 1 Element1 2 Element1 2 Element2
- F) A compilação falha.

**Listagem 4.** Código fonte da quarta pergunta.

```
01. public class Loop {
02.     public static void main(String[] args) {
03.         String[] sa = {"Element1 ", "Element2 "};
04.         for(int x = 0; x < 3; x++) {
05.             for(String s: sa) {
06.                 System.out.print(x + " " + s);
07.                 if( x == 1) break;
08.             }
09.         }
10.     }
11. }
```

## Questão 5

Dado o código exposto na **Listagem 5**, qual resultado é o correto?

- A) 16 mph, lope
- B) 18 mph, lope
- C) 24 mph, lope
- D) 27 mph, lope
- E) A compilação falha.
- F) Uma exceção é lançada em tempo de execução.

**Listagem 5.** Código fonte da quinta pergunta.

```
01. interface Rider {  
02.     String getGait();  
03. }  
04. public class Horse implements Rider {  
05.     int weight = 2;  
06.     public static void main(String[] args) {  
07.         new Horse().go(8);  
08.     }  
09.     void go(int speed) {  
10.         ++speed;  
11.         weight++;  
12.         int walkrate = speed * weight;  
13.         System.out.print(walkrate + getGait());  
14.     }  
15.     String getGait() {  
16.         return " mph, lope";  
17.     }  
18. }
```

## Questão 6

Dado o código exposto na **Listagem 6**, qual resultado é o correto?

- A) two three
- B) three two
- C) one
- D) three three
- E) A compilação falha.
- F) Uma exceção é lançada em tempo de execução.

**Listagem 6.** Código fonte da sexta pergunta.

```
01. class One {  
02.     String getType() { return "one"; }  
03. }  
04. class Two extends One {  
05.     String getType() { return "two"; }  
06. }  
07. public class Three extends Two {  
08.     String getType() { return "three"; }  
09. public static void main(String[] args) {  
10.     One g1 = new Two();  
11.     One g2 = new Three();  
12.     System.out.println(g1.getType() + " "  
13.         + g2.getType());  
14. }  
15. }
```

## Questão 7

Dado o código exposto na **Listagem 7**, qual resultado é o correto?

- A) javanese c c
- B) cat javanese c c
- C) cat javanese j j
- D) javanese j j
- E) javanese j c
- F) Uma exceção é lançada em tempo de execução.

**Listagem 7.** Código fonte da sétima pergunta.

```
01. class Cat {  
02.     public String type = "c ";  
03.     public Cat() {  
04.         System.out.print("cat ");  
05.     }  
06. }  
07. public class Javanese extends Cat {  
08.     public Javanese() {  
09.         System.out.print("javanese ");  
10.    }  
11.    public static void main(String[] args) {  
12.        new Javanese().go();  
13.    }  
14.    void go() {  
15.        type = "j ";  
16.        System.out.print(this.type + super.type);  
17.    }  
18. }
```

## Questão 8

Dado o código exposto na **Listagem 8**, qual resultado é o correto?

- A) 1 2 3 4 5 6 7 8 9 10
- B) 10 11
- C) 10
- D) 11
- E) A compilação falha.
- F) Uma exceção é lançada em tempo de execução.

**Listagem 8.** Código fonte da oitava pergunta.

```
01. public class Loop {  
02.     int process(int i) {  
03.         for (i = 1; i <= 10; i+=2);  
04.         System.out.println(i);  
05.     return i;  
06. }  
07. public static void main(String[] params) {  
08.     new Loop().process(10);  
09. }  
10. }
```

## Questão 9

Dado o código exposto na **Listagem 9**, qual resultado é o correto?

- A) true true
- B) true false
- C) false false
- D) false true

- E) A compilação falha.
- F) Uma exceção é lançada em tempo de execução.

**Listagem 9.** Código fonte da nona pergunta.

```
01. public class A {  
02.     public static void main(String[] args) {  
03.         System.out.println(new C() instanceof A);  
04.         System.out.println(new D() instanceof B);  
05.     }  
06.     public static void main(String[] args, int a) {  
07.         System.out.println(new D() instanceof A);  
08.         System.out.println(new C() instanceof B);  
09.     }  
10. }  
11. class B{}  
12. class C extends A {}  
13. class D extends B {}
```

## Questão 10

Dado o código exposto na **Listagem 10**, qual resultado é o correto?

- A) 5:6.0:100
- B) 7:200.0:100
- C) 7:200.0:10
- D) 5:6.0:10
- E) A compilação falha.
- F) Uma exceção é lançada em tempo de execução.

**Listagem 10.** Código fonte da décima pergunta.

```
01. public class Test {  
02.     int y = 10;  
03.     public Test change(int x, Double z) {  
04.         x = 7;  
05.         z = new Double(200);  
06.         y = 100;  
07.         return this;  
08.     }  
09.     public void print(int x, Double z) {  
10.         System.out.println(x + ":" + z + ':' + y);  
11.     }  
12.     public static void main(String[] args) {  
13.         int x = 5;  
14.         Double z = 6d;  
15.         new Test().change(x, z).print(x, z);  
16.     }  
17. }
```

## Questão 11

Dado o código exposto na **Listagem 11**, qual resultado é o correto?

- A) "0"
- B) "10"
- C) "Exceção"
- D) "Divisão por zero"
- E) A compilação falha.
- F) Uma exceção é lançada em tempo de execução.

**Listagem 11.** Código fonte da décima primeira pergunta.

```
01. package test11;
```

```
02. class E{  
03.     public static void main(String args[]){  
04.         int x = 0, y=10;  
05.         try{  
06.             y /=x;  
07.         }  
08.         System.out.print("Divisão por zero");  
09.     catch(Exception e){  
10.         System.out.print("Exceção");  
11.     }  
12. }
```

## Questão 12

Dado o código exposto na **Listagem 12**, qual resultado é o correto?

- A) 214 false false 0 0.25
- B) 34 false false 0 0.25
- C) 214 false false 0.5 0.25
- D) 34 true true 0 0
- E) 31 false false 0 0
- F) 31 false false 0 0.3333333333333333

**Listagem 12.** Código fonte da décima segunda pergunta.

```
01. package test12;  
02. public class Numbers {  
03.     public static void main(String[] args) {  
04.         int a[] = {1,2,0x1,4,5};  
05.         long b[][] = { {1,2,3,1},{5,4,2,9},{0,3,41,-11}};  
06.         double z = 11;  
07.         System.out.print(a[1] + a[2] + "" + a[3]);
```

```
08. System.out.print(" " + (a[3]==b[1][2]));  
09. System.out.print(" " + (a[2]==b[2][1]));  
10. System.out.print(" " + a[1]/b[2][2]);  
11. System.out.print(" " + z/b[2][2]);}
```

## Respostas das questões

As alternativas corretas são apresentadas a seguir:

1. B
2. D
3. E
4. E
5. E
6. A
7. C
8. D
9. E
10. A
11. E
12. B

## Análise das questões

A seguir, faremos uma análise detalhada de cada uma das questões, a partir da qual o leitor estará apto a assimilar as questões que errou no teste, em detrimento das armadilhas, falta de atenção e conceitos introdutórios que falamos anteriormente.

## Estudando a Questão 1

A resposta correta é a alternativa **B**, que traz o resultado *d b a c*.

Como podemos verificar, existem quatro comandos

**System.out.println()** na classe **Letter** e cada um imprime uma letra diferente. O que precisamos determinar aqui é a ordem com que as letras são impressas.

Como de costume em qualquer questão, devemos primeiro verificar se o programa compila. Para isso, analisemos cada linha de código na **Listagem 1**. Nesse momento, podemos verificar na linha 2 que o construtor foi corretamente declarado. Na linha 3 é declarado um bloco de código para ser executado na inicialização da classe, o que é perfeitamente normal, e na linha 4 temos a declaração do método executável **main()**. Em seguida, é declarado o método de instância **go()** na linha 7, e por fim, na linha 8, temos a declaração de mais um bloco de código, que será executado normalmente na inicialização da classe, pois esse bloco é estático.

Uma vez que tenhamos constatado que a classe realmente compila e compreendido a sua estruturação, podemos simular a execução da mesma. O início da execução de qualquer programa Java é realizado através de um comando que pode ser executado pelo *console* do sistema operacional. Logo após, a JVM executará a classe indicada no comando.

Como primeiro passo da execução de um programa Java, o **ClassLoader** (carregador de classes) do JDK vai carregar a classe **Letter** disparando a execução do bloco de comando estático na linha 8 (pois blocos estáticos pertencem à classe), imprimindo assim um "d". Depois disso, quando a execução do método **main()** iniciar, o comando a ser executado na linha 5 criará uma instância da classe **Letter**, que irá disparar a execução do bloco de código presente na linha 3 e, logo após, é executado o construtor (linha 2), o que resultará na impressão de "b" e "a". Por fim, é chamado o método **go()**, que imprime a letra "c". Dessa forma temos a sequência "d b a c".

## Estudando a Questão 2

A resposta correta para essa questão é a letra D, cujo texto diz que a saída é **false** e **MyStuff** não segue o contrato de **Object.equals()** e **Object.hashCode()**.

Primeiramente, de modo semelhante ao que fizemos na questão anterior, vamos conferir se a classe compila. Eliminada essa dúvida, constatamos que durante a sua execução são criados dois objetos: **m1** e **m2** (linhas 5 e 6). Como verificado mais à frente no código, o ponto central dessa questão envolve a comparação de objetos (o que inclui o contrato de **Object.equals()** e **Object.hashCode()**), como ocorre na linha 7, onde é chamado o método **equals()** para comparar dois objetos do tipo **Stuff**.

Ao executar passo a passo o código fonte dessa questão, verificamos que o retorno do método **equals()** será **false**, pois ele compara os atributos **name** de ambos os objetos, que são diferentes.

Porém, apenas isso não é suficiente para responder essa questão, pois ela pergunta se o contrato dos métodos **equals()** e **hashCode()** é seguido. Para verificar isso, precisamos conhecer o que é esse contrato. Ele basicamente define como deve ser a implementação dos métodos **equals()** e **hashCode()**, de forma que as classes do *framework* de *collections* que usam *hashcodes* funcionem corretamente quando forem criadas listas de objetos do tipo **Stuff**.

Ao analisar mais detalhadamente, o contrato do método **equals()** estabelece que o mesmo tem a função de comparar dois objetos, o que causa impacto na definição do método **hashCode()**, que tem a função de criar códigos de *hashcode* únicos para os objetos que chamam esse método. Como regra do método **equals()**, se o objeto recebido como parâmetro for de tipo incompatível ou for **null**, deve-se retornar **false**, caso contrário, ele deve retornar **true** se os objetos forem iguais (e **false** se forem diferentes). Aqui temos uma definição importante, relacionada a como determinar se dois objetos são iguais. Como isso depende da aplicação, fica a cargo do programador realizar essa definição, que deve refletir a lógica do mundo real. No caso da classe **Stuff**, por exemplo, faz sentido definir que dois objetos **Stuff** são iguais quando seus nomes são iguais.

Portanto, agora chegamos a um novo problema: definir o código fonte para determinar quando dois objetos da classe **Stuff** são iguais. A solução mais lógica é empregar o método **equals()** da classe **String** (como na linha 12 da **Listagem 2**). Esse método vai avaliar o conteúdo das *strings* caractere a caractere, pois se compararmos ambos os objetos com o clássico `==`, teremos problema quando compararmos objetos **Stuff** distintos (que têm endereços de memória diferentes) e que contêm atributos **name** alocados em endereços de memória também distintos, mas que apresentam o mesmo conteúdo (pois **name** é uma **String**). Lembre-se que duas Strings podem conter exatamente o mesmo texto, mas serem objetos diferentes).

O contrato do método **equals()** também define outras propriedades que devem ser seguidas, a saber: reflexividade (um objeto deve ser igual a ele mesmo), simetria (se um objeto é igual ao segundo, o segundo é igual ao primeiro) e transitividade (se um objeto é igual a um segundo, e o segundo é igual a um terceiro, então o primeiro deve ser igual ao terceiro), que são atendidas pelo exemplo apresentado na **Listagem 2**.

Como parte do contrato de definição dos métodos **Object.equals()** e **Object.hashCode()**, temos a norma para o método **hashCode()** que determina que para dois objetos serem iguais eles devem ter o mesmo *hashcode*. Porém, na **Listagem 2** esse critério falha, pois o método **hashCode()** não foi sobreescrito e será utilizada a sua definição implementada na classe **Object**, que usa o endereço de memória do próprio objeto para definir o *hashcode*, enquanto **equals()** vai usar o valor das *strings*, quebrando assim a consistência entre ambos, o que permite a geração de *hashcodes* diferentes para objetos **Stuff** iguais (segundo **equals()**).

Portanto, a comparação efetuada na **Listagem 2** ocorre corretamente, retornando **false**, mas o contrato é violado pelo método **hashCode()** da classe **Object**.

## Estudando a Questão 3

A resposta correta para essa questão é a letra E. O código apresentado falha porque não existe o método **sort()** sem parâmetros na classe **ArrayList** do Java SE 8. Para que a linha 9 funcione corretamente é necessário adicionar um parâmetro ao método **sort()**, um **Comparable**. O mesmo tem a função de ordenar um vetor de objetos seguindo um determinado critério, a ser definido pelo programador.

## Estudando a Questão 4

A resposta correta para essa questão é a alternativa E. Analisando o código fonte na **Listagem 4** pode-se verificar uma armadilha para “pegar” candidatos não muito atentos ou que não conheçam o **for** melhorado (*enhanced for*) (linha 5), pois com sua notação simplificada é possível percorrer o vetor **sa** de forma que o elemento corrente seja referenciado pela variável **s**. Sabendo disso, essa questão é definida pelo fato do candidato ter domínio em laços de repetição, sendo necessário percorrer mentalmente os laços e anotar o que for impresso no console, o que, neste caso, leva à alternativa mencionada.

## Estudando a Questão 5

Assim como nas questões anteriores, a resposta correta para a quinta questão é a alternativa E. Analisando o código fonte na **Listagem 5**, a princípio tudo parece estar em ordem, uma vez que não há nenhum erro de compilação aparente. Contudo, há um erro muito sutil nesse programa que vai levar à falha na compilação. Como sabemos, não se pode reduzir a visibilidade de um método herdado, mesmo se esse método for herdado de uma interface. Assim, a falha de compilação ocorre na linha 15, onde se tenta reduzir a visibilidade do método **getGait()** para **default**. Lembre-se que métodos de qualquer interface são naturalmente públicos (mesmo quando não são declarados com a palavra **public**, como visto na linha 2).

## Estudando a Questão 6

A resposta correta para essa questão é a alternativa A. Como podemos verificar na **Listagem 6**, essa questão pode ser traíçoeira se o candidato não conhecer bem o mecanismo de herança. Primeiramente, a classe **One** é declarada normalmente e retorna “one” no método **getType()**. Em seguida, a classe **One** é estendida pela classe **Two**, que sobrescreve o método **getType()**, retornando “two”. O mesmo ocorre na classe **Three**, que sobrescreve o método **getType()**, retornando “three”. A parte duvidosa começa quando se cria um objeto **Two** e um objeto **Three** e as referências a esses objetos são mantidas como sendo da classe **One** (linhas 10 e 11).

Manter um objeto da classe **Two** com uma referência da classe **One** limita o acesso apenas aos métodos declarados na classe **One**, porém a implementação a ser executada na chamada ao método **getType()** será sempre a da versão mais especializada, que é a da classe **Two**. Portanto, o resultado da execução da linha 12 será “two” e “three”, dados pela versão mais especializada do método **getType()**.

## Estudando a Questão 7

A resposta para essa questão é a alternativa C. Com o código fonte apresentado na **Listagem 7** essa questão parece inofensiva e de fácil resposta, no entanto contém algumas sutilezas que podem levar ao erro, pois ela demanda um conhecimento apurado sobre a lógica da chamada de construtores de superclasses, além de ser necessário saber utilizar corretamente as palavras-chave **super** e **this**.

Como primeiro comando do método **main()**, na linha 12, é criado um objeto do tipo **Javanese** (por curiosidade, é um tipo de gato) e é chamado o método **go()**. Podemos verificar que o construtor da classe **Javanese** contém um único comando que imprime “javanese” na tela. Porém, não devemos esquecer que o primeiro comando que é executado sempre no início de cada construtor é o construtor da superclasse, mesmo se esse estiver omitido.

De acordo com as regras da linguagem Java, no caso de omissão da chamada ao construtor de uma superclasse, o Java chama automaticamente o construtor *default* dessa superclasse. Como o construtor **default** é, por definição, um construtor sem parâmetros, é chamado então o construtor da classe **Cat** declarado na linha 4, imprimindo na tela o termo “cat”, e depois a execução retorna ao construtor da classe **Javanese**, que imprime “javanese”.

Por fim, na linha 15 a variável **type** da classe **Javanese** é definida como “j”, sobrescrevendo a variável **type** da classe **Cat**. Em seguida é acessado **this.type** e **super.type**, sendo o valor de ambos os comandos iguais a “j”, pois trata-se apenas de uma única variável (e em ambos os casos será impresso o mesmo valor).

## Estudando a Questão 8

A resposta correta para essa questão é a alternativa D. Em uma primeira leitura do código parece impossível imprimir só um valor na tela, pois o laço **for** conta de 1 a 10. Contudo, na verdade o laço **for** presente nesse código não imprime nenhum valor na tela. Isso ocorre porque o comando **System.out.println**, na linha 4, não pertence ao **for**, visto o corpo do **for** nada mais é que um comando em branco, ou seja, apenas um ponto-e-vírgula. Deste modo, para chegar ao resultado correto devemos seguir a inicialização do **for** com **1** e contar de dois em dois, obtendo **3, 5, 7, 9**, e, finalmente, **11**, que vai fazer a condição do **for** falhar, terminando sua execução. Em seguida, na linha 4, é impresso **11** no **System.out.println**, que é o valor corrente da variável **i**.

## Estudando a Questão 9

A resposta correta para essa questão é a alternativa E, que indica que a compilação falhou. Essa questão não é nem um pouco diferente das demais. Analisando a **Listagem 9**, à primeira vista parece tudo em ordem: temos alguns objetos sendo criados e alguns testes com **instanceof**. Apesar disso, esse código não compila, o que inicialmente parece não ter explicação.

Ao analisar o código mais detalhadamente, observando os testes nas linhas 3 e 4, é verificado se um **C** é um **A** e se um **D** é um **B**. E está tudo certo com esses comandos, sendo impresso **true** em ambos os testes caso o programa funcionasse. Porém, nas linhas 7 e 8 é verificado se um **D** é um **A** e se um **C** é um **B**. Como podemos constatar, essas condições nunca serão verdadeiras, pois **D** não faz parte da hierarquia de classes de **A** e **C** não faz parte da hierarquia de classes de **B**. A forma como o Java lida com essa situação é impedindo a compilação, pois esses tipos são incompatíveis, ou seja, o Java interrompe a compilação ao encontrar esses testes.

## Estudando a Questão 10

A resposta correta para essa questão é a alternativa A. Como podemos verificar na **Listagem 10**, essa questão trabalha o assunto passagem de parâmetros em Java, misturando diversas situações. Inicialmente, temos a variável **x** declarada no método **main()** com o valor **5** e a variável **z** que recebe o valor real **6** (observe que **6d** significa **6**, porém convertido para **double**), conforme as linhas 13 e 14. Como ambas as variáveis são de tipos básicos, elas até podem receber valores diferentes durante a execução do método **change(x, z)** (linhas 4 e 5), mas quando o mesmo concluir sua execução e retornar, seus valores não

sofrerão nenhuma alteração no método **main()**, pois em Java os tipos básicos são passados por valor e não por referência.

Quanto à última variável, **y**, ela inicia com o valor 10 (vide linha 2) e depois, no método **change()**, recebe 100, mas essa mudança é permanente, pois **y** é uma variável de instância. Por fim, o método **print()** é executado (linha 15), imprimindo todas as variáveis. Observe que **x** vale **7**, **z** vale **6** e **y** vale **100**.

Uma observação relevante é que nas chamadas encadeadas de métodos da linha 15 é usada a mesma instância da classe **Test** em cada chamada de método. Inicialmente, o construtor retorna um objeto da classe **Test** e este invoca **change()**, que por sua vez retorna **this**. Em seguida, **this** invoca **print()**, possibilitando que a mesma instância chame ambos os métodos: **change()** e **print()**.

## Estudando a Questão 11

Essa questão introduz um problema de divisão por zero, bastante evidente na **Listagem 11**. Sabemos que a linguagem Java trata a divisão por zero lançando uma exceção que precisa ser capturada, e felizmente, o código fonte dessa questão apresenta o tratamento correto para a exceção lançada. Porém, ao analisar a linha 8, encontramos um **System.out.println()** em uma posição inválida no código, pois ele não está dentro do **try** nem do **catch**, causando assim um erro de compilação, que é a resposta dessa questão.

## Estudando a Questão 12

Essa questão apresenta algumas armadilhas envolvendo números inteiros (**int** e **long**), números reais (**double**) e operações de divisão. Observe que estas armadilhas se tornam ainda mais desafiadoras, uma vez que tais operações são combinadas à concatenação de *strings*, sobrecarregando o operador **+**. Como todas as alternativas disponíveis apresentam possíveis resultados de execução, já sabemos que o código-fonte dessa questão compila e não lança exceções, o que nos poupa um trabalho razoável de verificação.

Outro ponto de destaque vai para a forma como o Java manipula *arrays*. Lembre-se que o Java considera que a primeira posição deste tem índice zero (ou seja, os *arrays* iniciam no zero), fato que pode enganar candidatos desatentos logo no primeiro comando (linha 7): **System.out.print(a[1] + a[2] + "" + a[3])**. Além desta, existe ainda mais uma armadilha nessa questão, relacionada à sobrecarga do operador **+**, pois a sua utilização é diferente quando envolve operandos numéricos ou operandos do tipo **String**. Sendo assim, é necessário determinar o que ocorre a cada vez que este operador é empregado: soma ou concatenação de *strings*?

As regras básicas nos dizem que quando o operador **+** é aplicado diversas vezes, formando uma sequência, a expressão inteira é avaliada da esquerda para a direita, de forma que sempre que o operador **+** é aplicado a dois números, resulta em soma, e sempre que pelo menos um dos operandos é **String**, ocorre a concatenação do lado esquerdo com o lado direito do **+**, resultando em uma nova **String**.

Revisada essa definição, podemos verificar que ocorre a soma de **a[1]** com **a[2]**, resultando em **2 + 0x1**, que é igual a **3** (operандos numéricos). Em seguida, esse resultado é concatenado com a **String** vazia "", criando assim a **String** "3". Por fim, esse texto é concatenado com **a[3]**, que vale **4**, obtendo como resultado a **String** "34". Observe que **0x1** nada mais é do que **1** em hexadecimal.

O próximo comando, que está na linha 8 (**System.out.print(" " + (a[3]==b[1][2]))**), retorna **false**, pois **a[3]** vale **4** e **b[1][2]** vale **2**, ou seja, tratam-se de números diferentes. Como resultado, o Java concatena a **String** em branco ("") com o valor **boolean false**, resultando na **String** "false". De maneira semelhante, temos outro comando na linha 9 que compara dois números diferentes, o que também resulta na impressão da **String** "false".

O próximo comando, na linha 10 (**System.out.print(" " + a[1]/b[2][2])**), contém outra armadilha para os desavisados: a "divisão inteira". À primeira vista, temos a divisão de 2 por 4 (pois 2 é **int** e 4 é um **long**), de forma que **0,5** é o resultado exato. Entretanto, as regras básicas nos dizem que quando temos a divisão de dois números inteiros, o resultado será o quociente da divisão; nesse caso **0**, ao invés de **0,5**.

Por fim, na linha 11 temos o seguinte comando:

**System.out.print(" " + z/b[2][2])**. De acordo com o código, verifica-se que **z** é um **double** com valor **1** (não importa se foi inicializado com um **long**). Sendo assim, o que temos é a divisão de um **double** (**1**) por um **long** (**4**). Deste modo é aplicada a divisão de números reais (pois um operando é real: **double**), o que resulta em **0,25**, completando a questão.

Essa certificação é uma das mais simples entre todas as opções que são oferecidas pela Oracle, mas contém muito conteúdo e pode apresentar desafios potenciais dados por casos especiais e situações propositalmente enganosas. Sendo assim, a concentração na prova e o domínio dos conceitos básicos são fundamentais para o sucesso no exame.

Além da certificação para programador Java OCAJP8 oferecida para o nível *Associate* (o mais básico), a certificação para o Java SE é oferecida para os níveis *Professional* e *Master*, sendo o nível *Master* o mais abrangente em termos de conteúdo. Outra certificação disponibilizada pela Oracle é a *Java EE Certification*, que foca na versão *enterprise* do Java e é aplicada nos níveis *Professional* e *Master*, apresentando três categorias que podem ser escolhidas: *Business Component Developer*, *Web Services Developer* e *Web Component Developer*. Para completar, é oferecida a certificação Java ME, para aplicações que têm o objetivo de serem executadas em aparelhos móveis que suportam Java, geralmente celulares ou *smartphones*.

## Links

### **Javadoc da plataforma Java SE 8.**

<http://docs.oracle.com/javase/8/docs/api/>

### **Site da Oracle University.**

<http://education.oracle.com/>

### **Site da Oracle para a Certificação OCAJP8.**

[http://education.oracle.com/pls/web\\_prod-plq-dad/db\\_pages.getpage?page\\_id=5001&get\\_params=p\\_exam\\_id:1Z0-808](http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=5001&get_params=p_exam_id:1Z0-808)



**John Soldera**

é bacharel em Ciências da Computação pela UCS (Universidade de Caxias do Sul), mestre em Computação Aplicada pela Unisinos e cursa atualmente doutorado em Ciências da Computação pela UFRGS (Universidade Federal do Rio Grande do Sul) [...]

---



[www.devmedia.com.br](http://www.devmedia.com.br)

[versão para impressão]

Link original: <http://www.devmedia.com.br/articles/viewcomp.asp?comp=32167>

# Eclipse Luna: Sua IDE pronta para o Java 8

Conheça neste artigo os novos recursos da versão mais atual do Eclipse.

---

## Fique por dentro

Este artigo é útil para usuários iniciantes ou avançados, curiosos ou já conhecedores da IDE mais utilizada pelos desenvolvedores Java do mercado, o Eclipse. Infelizmente muitos desses usuários já possuem certo vício na utilização dessa, e devido a isso não aproveitam o máximo da ferramenta, assim como dos novos recursos adicionados a cada nova versão, seja pela falta de conhecimento ocasionado pela pequena divulgação do conteúdo ou simplesmente pela ausência de interesse. Com base nisso, serão apresentados nesse artigo alguns dos novos recursos do Eclipse Luna, que estejam relacionados ou não à nova versão do Java, o Java 8.

Atualmente é impossível imaginar um programador ou uma equipe de desenvolvimento que busca alta produtividade trabalhando sem o auxílio de uma IDE (*Integrated Development Environment*). O uso desse tipo de ferramenta já está tão comum na vida do profissional de TI que ele nem percebe o quanto dependente seu trabalho é dela, bem como o fato de que a ferramenta evolui constantemente, buscando facilitar e agregar mais e mais funcionalidades que auxiliem o trabalho de desenvolvimento do software.

Esse é o caso da IDE Eclipse, que recentemente completou 10 anos com o lançamento de sua mais nova versão, denominada Luna. Essa versão é resultante de 76 projetos diferentes, com mais de 10 milhões de novas linhas de código contribuídas para seu desenvolvimento.

A utilização de uma IDE como ferramenta de desenvolvimento é praticamente um vício de trabalho. Recheada de recursos visuais, auxiliadores de codificação e depuração, plugins para integração com outras ferramentas (como servidores de aplicação, repositórios, gestores de configuração), plataforma para testes, etc., pode-se dizer que é impossível uma empresa desenvolver um projeto sem ela. Isso tudo, no entanto, resulta num estado de comodismo por parte do programador em nunca pensar na possibilidade de trocar de ferramenta, ou até mesmo imaginar que a que ele use atualmente não possa melhorar em uma próxima versão com alguma *feature* que realmente traga diferença para sua rotina. É bastante comum pensar assim, mas esse não é o caso do Eclipse, pois a cada versão recursos são adicionados para tentar atender ao máximo as demandas da comunidade.

Desde que surgiu, o Eclipse sempre trouxe melhorias significativas a cada novo lançamento, além de ter à disposição vários plug-ins “não oficiais” construídos por terceiros. Sendo assim, o objetivo deste artigo é mostrar que vale muito a pena investir algum tempo experimentando algumas das novidades incorporadas ao Luna, que vão de pequenas coisas, como o Splitter Editor e o novo Dark Theme, até features mais impactantes, como a nova interface gráfica para modelagem EMF. Deste modo, ao longo da parte prática deste artigo serão apresentadas algumas dessas novas funcionalidades com exemplos e destaque visuais.

Além das novidades gerais (relacionadas à utilização da IDE como um todo) incorporadas ao Luna, não se pode esquecer que essa é a primeira versão do Eclipse a disponibilizar suporte total e nativo ao recém-lançado Java 8. E como já é de conhecimento que essa versão do Java é considerada a que abriga a maior quantidade de alterações, certamente as IDEs que a suportam estão recheadas de novidades para usufruir ao máximo da linguagem, tanto em questões de configuração quanto em uso programático propriamente dito.

## Novidades e melhorias gerais

Ao falar rapidamente das novidades do Eclipse Luna, vale ressaltar que algumas delas são melhorias e correções de bugs ou aperfeiçoamento de funcionalidades já existentes. A equipe do Eclipse elencou algumas das solicitações mais reincidientes nos fóruns para determinar o que deveria ser melhorado ou corrigido em termos de funcionalidades.

Mais detalhes e demonstrações serão abordadas na parte prática, mas para atiçar a leitura deste artigo, citaremos de forma breve as mudanças mais interessantes e/ou significantes na workbench do Eclipse. Assim, para agradar as preferências visuais ou auxiliar na agilidade do desenvolvedor, o Luna oferece um novo tema Dark compatível com os populares temas darks dos Sistemas Operacionais mais conhecidos. O Split Editor, que há muito tempo está na fila de melhorias do Eclipse, finalmente está à disposição do desenvolvedor. Com ele é possível alterar o mesmo arquivo em diferentes editores ao mesmo tempo, o que é especialmente útil quando se lida com arquivos muito grandes. Além disso, outro recurso visual introduzido foi a nova opção de reordenação de perspectivas, que ajuda o desenvolvedor a organizar melhor suas áreas de trabalho mais utilizadas e transitar mais facilmente entre elas.

Além de features para desenvolvimento, o Eclipse resolveu facilitar a vida do desenvolvedor simplificando a abertura de arquivos no sistema local (agora pode ser feito diretamente da IDE). Ademais, através da nova view Terminals será possível acessar o shell do seu sistema operacional a partir da IDE, sem a necessidade de nenhum plugin. Falando em integração entre sistemas, para os usuários do Ubuntu, juntamente com a Canonical foram resolvidos os problemas de integração que existiam na aparência da ferramenta, principalmente referentes ao menu.

Vale ressaltar ainda que o Eclipse não “significa” apenas programação. Trata-se de uma ferramenta completa e através do projeto Sirus o Luna permite a arquitetos e analistas modelar e criar graficamente suas atividades com integração e atualização total entre os artefatos envolvidos. Além disso, a especificação UML 2.5 agora é suportada pelo Eclipse, viabilizando a criação e validação dos diagramas que são compreendidos por ela.

Essas e muitas outras novidades estão presentes no Eclipse Luna. Portanto, no decorrer do artigo, serão demonstradas algumas delas com exemplos e descrições de seu funcionamento e utilidade.

## Suporte ao Java 8

O que não pode faltar em um artigo sobre as novidades do Eclipse Luna é um tópico dedicado ao suporte fornecido ao Java 8. Embora já exista uma integração com esta versão do Java através de plugins e atualizações de versões anteriores, como o Eclipse Kepler, é no Luna que esse suporte é totalmente nativo.

Como curiosidade, vale destacar que foram feitos esforços enormes para que, pela primeira vez, as duas IDEs mais populares para desenvolvimento Java (Eclipse e NetBeans) lançassem suas versões compatíveis com o Java 8 concomitantemente com o lançamento da linguagem. Portanto, assim que o Java 8 foi lançado, tanto Eclipse quanto NetBeans já disponibilizaram para os usuários uma versão compatível de suas ferramentas.

Dentre as várias novidades para o Java 8 presentes no Eclipse Luna, pode-se citar grandes alterações e atualizações em recursos disponíveis no editor de arquivos Java, como no Quick Fix (agora é possível alterar as configurações do projeto ou o JRE direto do editor) e no Quick Assists (agora é possível a criação de funções lambda através de funções anônimas e vice-versa). Além dos recursos existentes no Java Editor, novas opções foram adicionadas aos wizards de formatação e compilação de código.

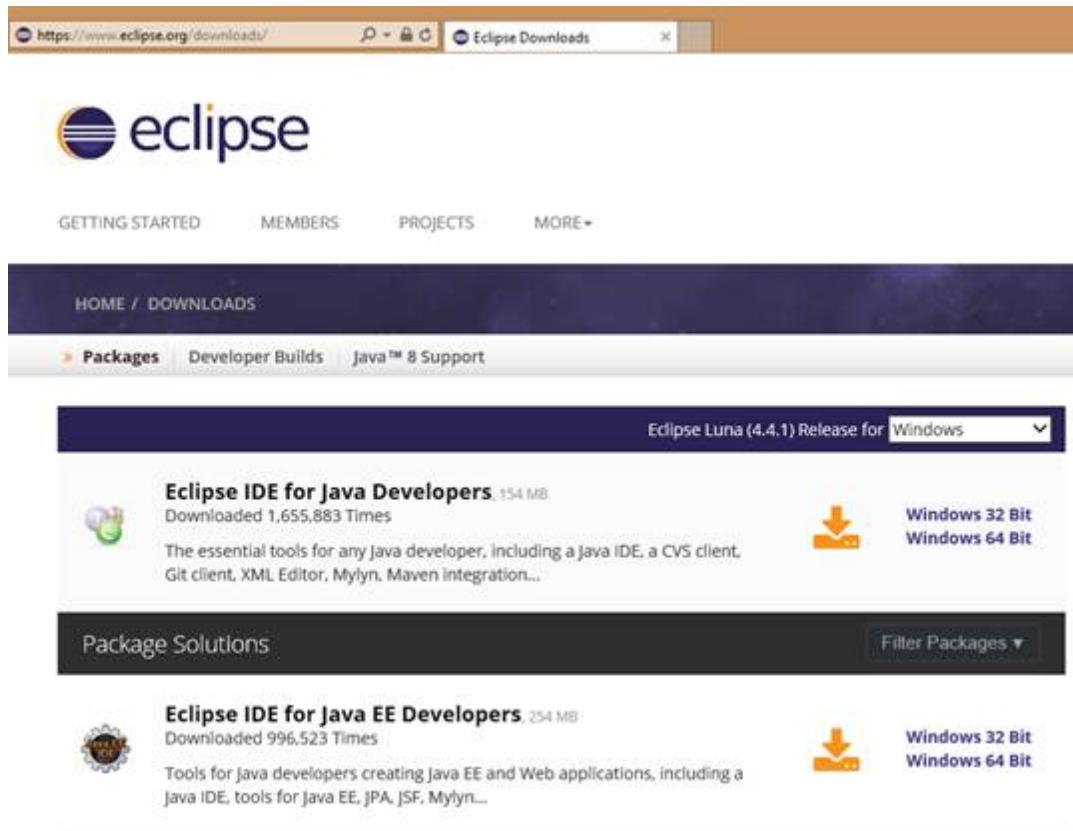
Quick Fix é um recurso utilizado em IDEs que possibilita a correção de alguns erros ou problemas diretamente do editor onde o código está sendo escrito. Para isso, opções relativas ao erro e/ou teclas de atalho são exibidas ao desenvolvedor. Da mesma forma, ou seja, diretamente do editor, o Quick Assist auxilia na escrita do código propriamente dito, como por exemplo, através das famosas opções de auto-complete.

Assim como os novos recursos para simplificar a codificação, as ferramentas de Debug e testes com JUnit também foram atualizadas, bem como funcionalidades para refatoração do seu código. Enfim, várias delas serão demonstradas no decorrer do artigo para instigar a curiosidade de conhecer não somente o Eclipse Luna, mas também a nova versão do Java, que está repleta de novas JSRs.

## Instalação

Caso o leitor esteja sendo apresentado ao Eclipse pela primeira vez, pode ficar desesperado, pois a instalação é bastante simples e não exige nenhum tipo de configuração especial.

Uma vez que alguma versão do Java (para os propósitos desse artigo, dê preferência à 8) já esteja instalada em seu sistema operacional, basta acessar a página de downloads do Eclipse (indicada na seção **Links**) e baixar a versão desejada, como indica a **Figura 1**.



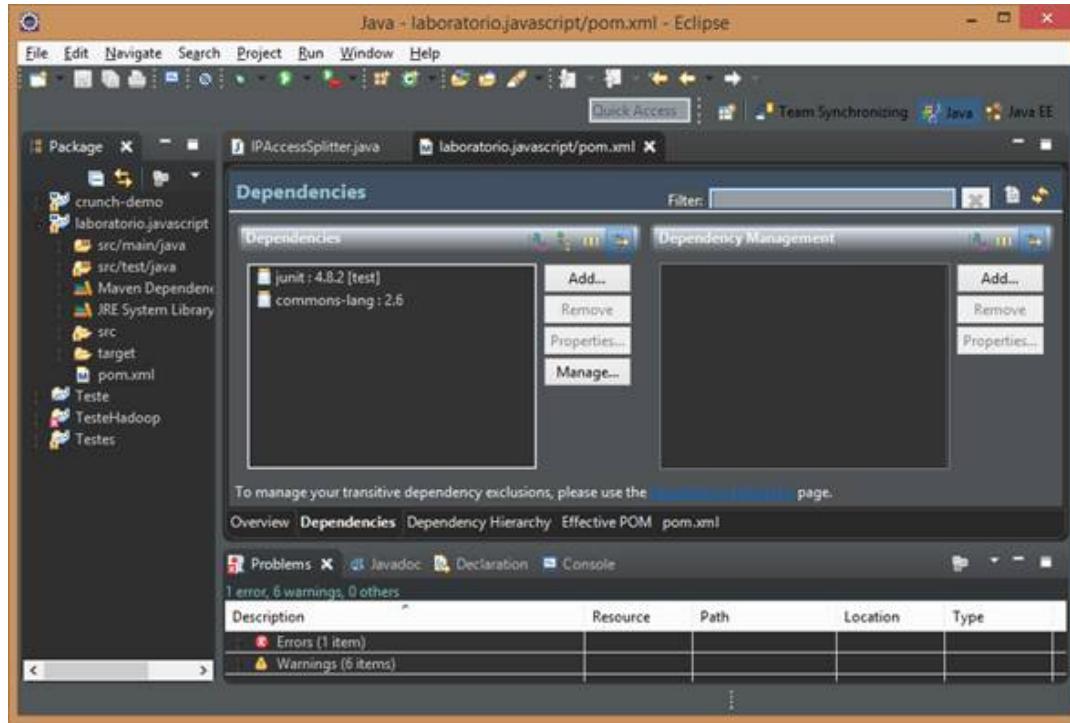
**Figura 1.** Página para download do Eclipse.

Seja em sua versão para desenvolvedores Java SE, ou para desenvolvedores Java EE, após o download é necessário apenas descompactar e executar a ferramenta, que já estará pronta para uso.

## Recursos visuais

Um novo tema de janelas foi introduzido no Eclipse Luna, o Dark Theme. Este pode ser encontrado e habilitado facilmente através da sequência de passos nos menus *Window > General > Appearance*. Esse tipo de tema é muito popular entre os usuários de ferramentas open source, e também é uma forma de demonstrar o poder da nova biblioteca de estilos do Eclipse, uma vez que para esse tema funcionar corretamente o Eclipse precisa se adequar aos recursos visuais disponíveis por cada Sistema Operacional. Para usuários mais avançados da ferramenta, plug-ins podem ser desenvolvidos para incrementar ainda mais o estilo de seu próprio Eclipse.

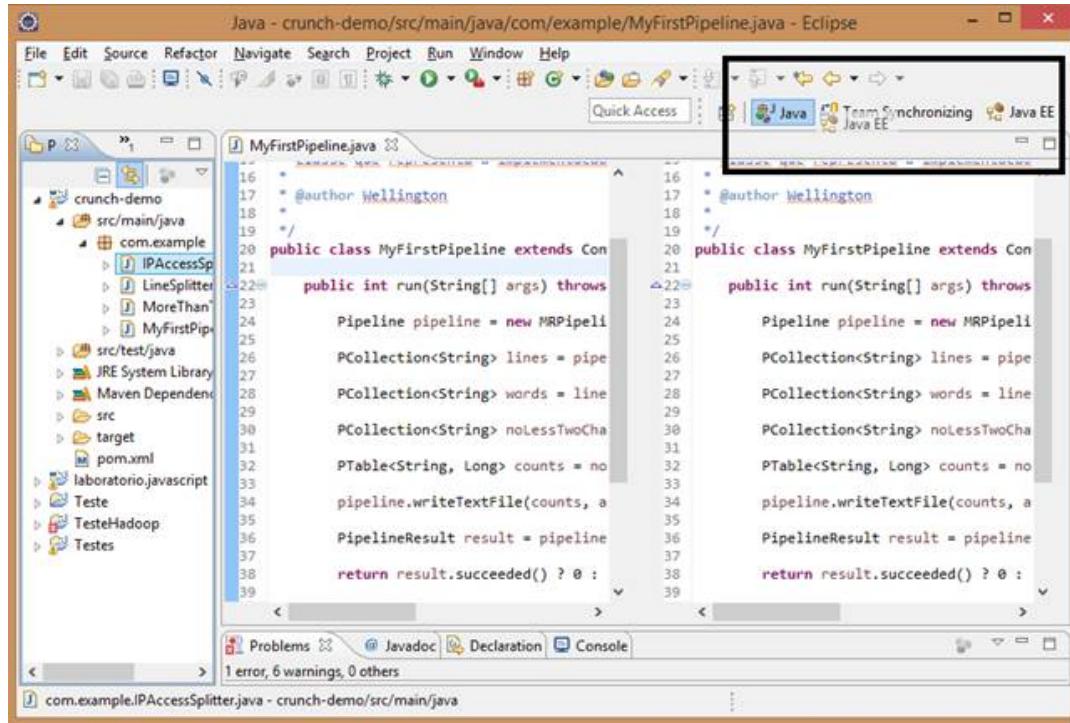
Uma amostra desse novo tema pode ser vista na **Figura 2**. Como você pode notar, as novidades vão muito além de janelas e wizards, pois toda a paleta de ícones do Eclipse foi redesenhada para ter a mesma qualidade de aparência tanto no tema escuro quanto nos clássicos. Além disso, os destaque visuais das sintaxes e editores de linguagens (Java, JavaScript e XML, por exemplo) também absorveram as modificações introduzidas pelo Dark Theme.



**Figura 2.** Tema Dark.

Outra pequena, porém sensível melhoria na usabilidade do Eclipse Luna é a nova forma de ajuste de posicionamento e organização da paleta de perspectivas mais utilizadas pelo desenvolvedor, localizadas no canto superior direito da IDE. É possível distribuir os botões das perspectivas clicando e arrastando-os para a posição desejada, como demonstra a

**Figura 3.**

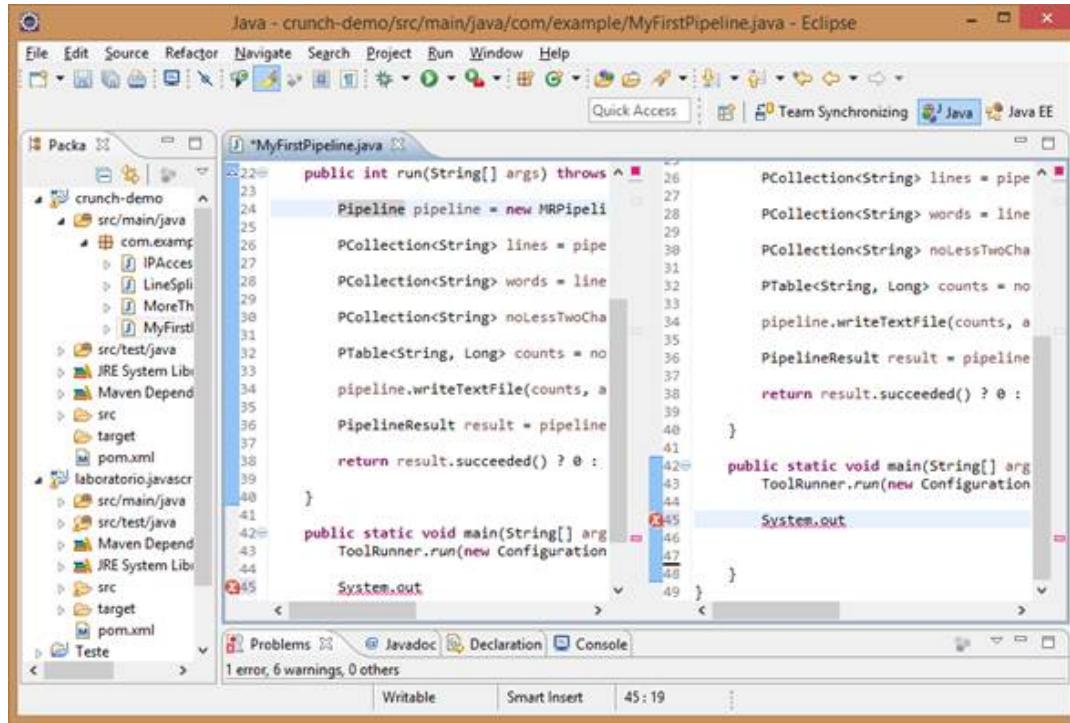


**Figura 3.** Ordenação das Perspectivas.

## Split Editors

Essa é uma solicitação antiga, feita pela comunidade de usuários do Eclipse há muitos anos. Desde 2002 a solicitação 8009 existe no Bugzilla da ferramenta, sugerindo a implementação do mecanismo de Split Editor na IDE. Esse pedido surge da necessidade recorrente de se alterar partes diferentes de um mesmo arquivo ao mesmo tempo em editores separados. Sem o Split Editor, isso não é possível, e a usabilidade fica prejudicada, pois nos obriga a navegar para baixo e para cima repetidas vezes pelo mesmo arquivo em um único editor. Com o Split Editor, no Luna é possível dar split no mesmo arquivo com as teclas de atalho **CTRL + {** para dividir o arquivo verticalmente e **CTRL + \_** para dividi-lo horizontalmente. Na **Figura 4** podemos verificar a edição de duas partes diferentes de uma mesma classe através do editor Java do Eclipse.

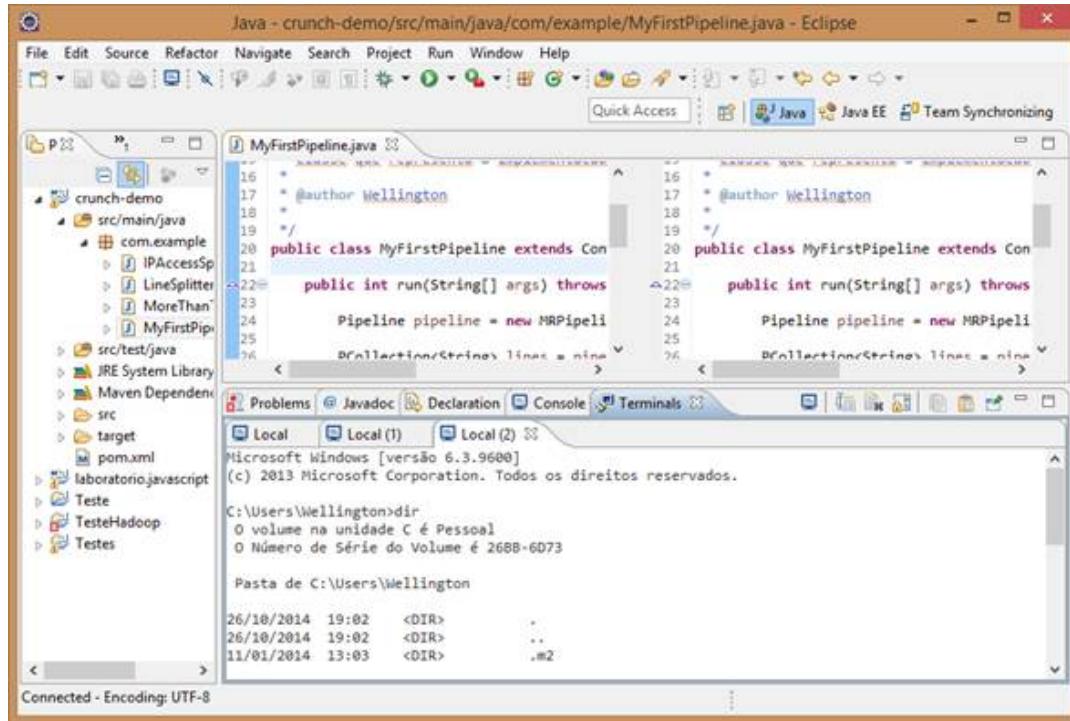
Além do Split Editor, melhorias para editar arquivos e pastas foram adicionadas ao Luna. Deste modo, para manipular pastas e arquivos em seus locais de origem, agora é possível abri-los diretamente no sistema de arquivos ao clicar com o botão direito do mouse em *Show In > System Explorer*.



**Figura 4.** Split Editors.

## Terminal de Comando

Com a intensão constante de buscar oferecer tudo o que é necessário para que o desenvolvedor permaneça dentro da IDE e de dentro dela consiga fazer todas as atividades do seu trabalho, o Eclipse Luna oferece uma nova View chamada Terminals, acessível em *Window > Show View > Other... > Target Explorer View > Terminals*. Com este recurso, o desenvolvedor consegue ter acesso ao terminal ou shell de seu sistema operacional diretamente da IDE. A **Figura 5** mostra um terminal Windows (prompt de comando) sendo executado na view Terminals.



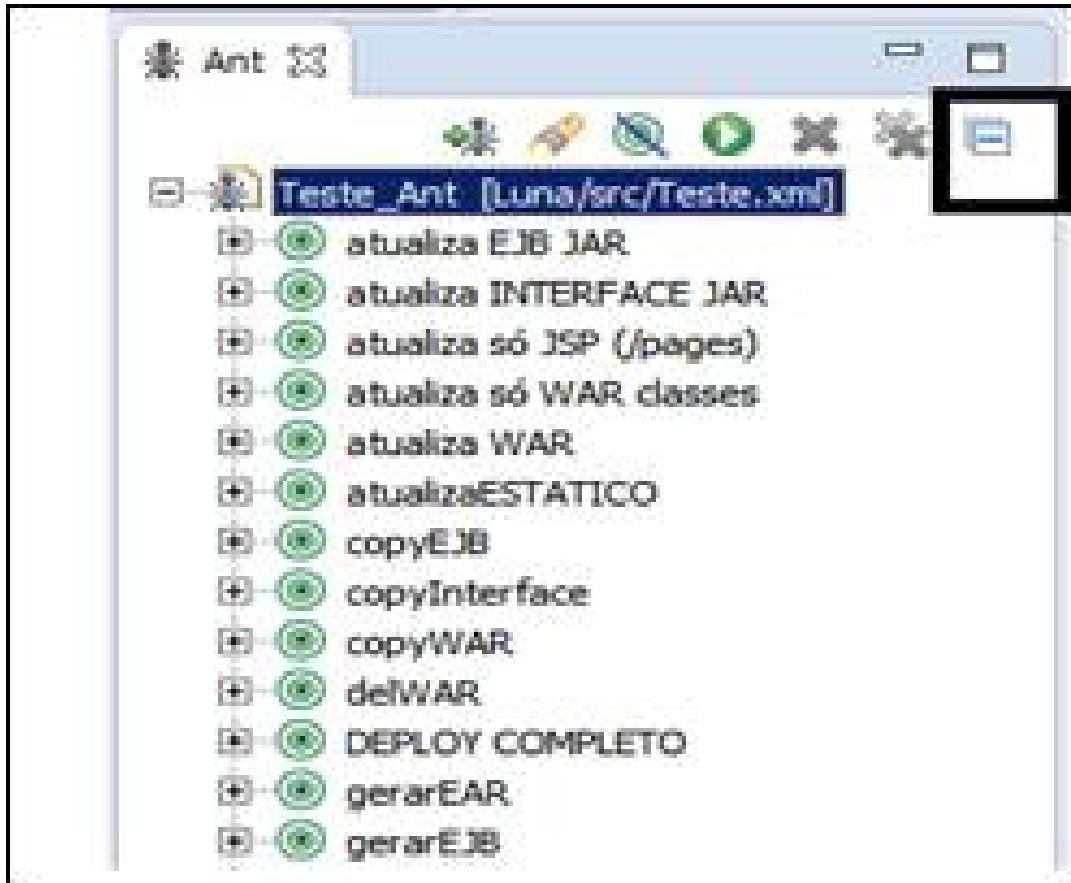
**Figura 5.** View Terminals.

Além de conexões com o terminal local, também é possível acesso a terminais remotos através de Telnet ou SSH. Na view Terminals também estão disponíveis suporte a sintaxe com colorações para destaque visual, bem como autopreenchimento, cópia e colagem de instruções, e uma série de outras integrações de acordo com a plataforma (Sistema Operacional) executada.

## Outros recursos

Junto com as grandes novidades apresentadas, várias outras pequenas, que as vezes passam despercebidas, também foram introduzidas. Sendo assim, nos próximos parágrafos analisaremos rapidamente algumas delas.

A começar pela view *Ant*, por exemplo, acessível através do menu *Window > Show View > Ant*, um novo ícone (*Collapse All*) foi adicionado ao layout para poder recolher de uma só vez todas as atividades (*Targets*) do arquivo XML Ant que estejam abertas (veja o destaque na **Figura 6**).



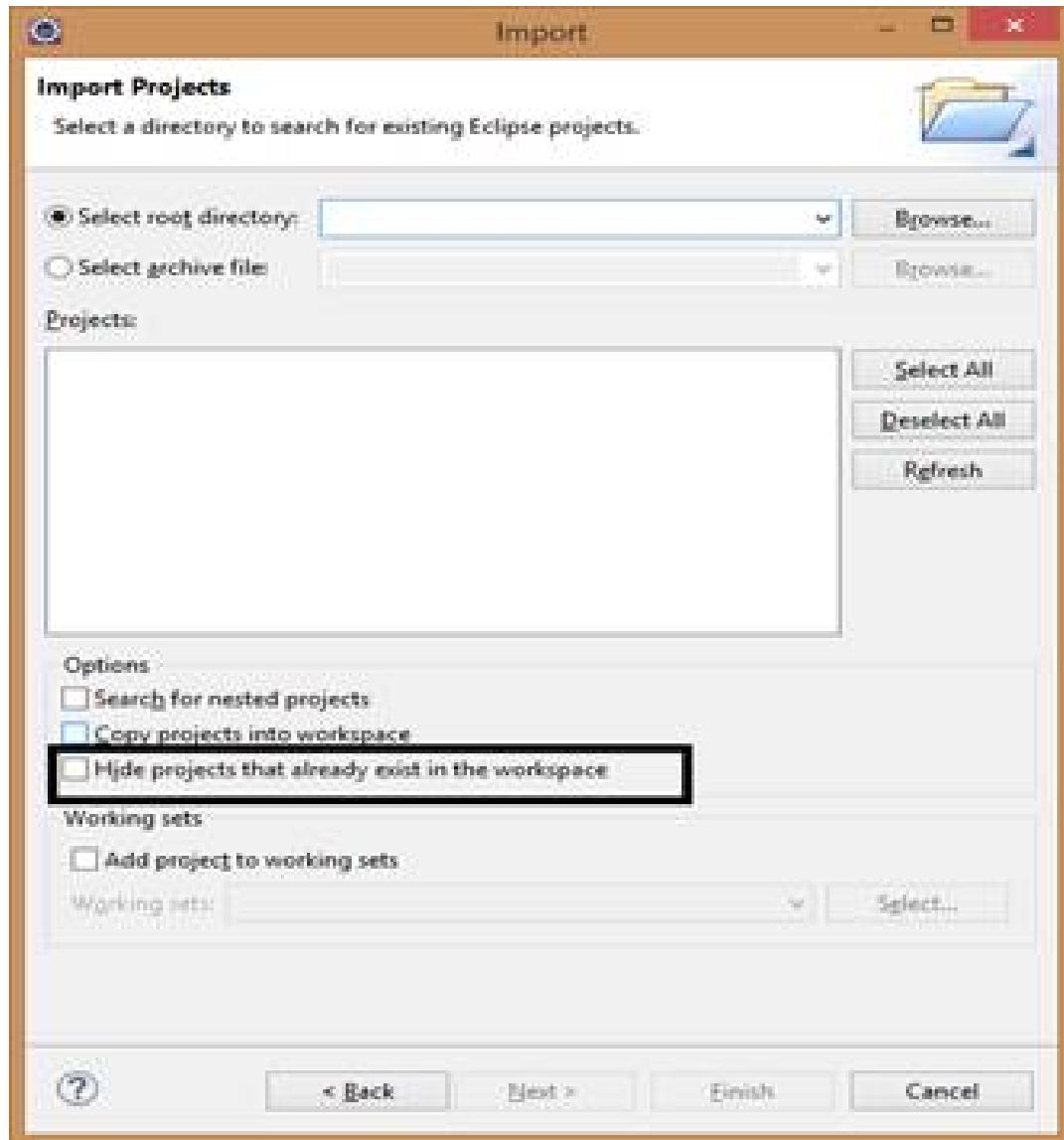
**Figura 6.** Botão Collapse All (View Ant).

Outra mudança ocorreu nos ícones da área de trabalho do Eclipse, que nas versões anteriores eram disponibilizados no formato GIF; agora possuem o formato PNG. Embora quase imperceptível, essa alteração visa principalmente melhorar a aparência dos botões dentro do Dark Theme ou sua integração a sistemas operacionais que antes apresentavam problemas de layout, como as diferentes versões do Linux. Na **Figura 7** é apresentada uma paleta de ícones do Eclipse para ilustrar o que foi descrito.



**Figura 7.** Paleta de ícones do Eclipse Luna.

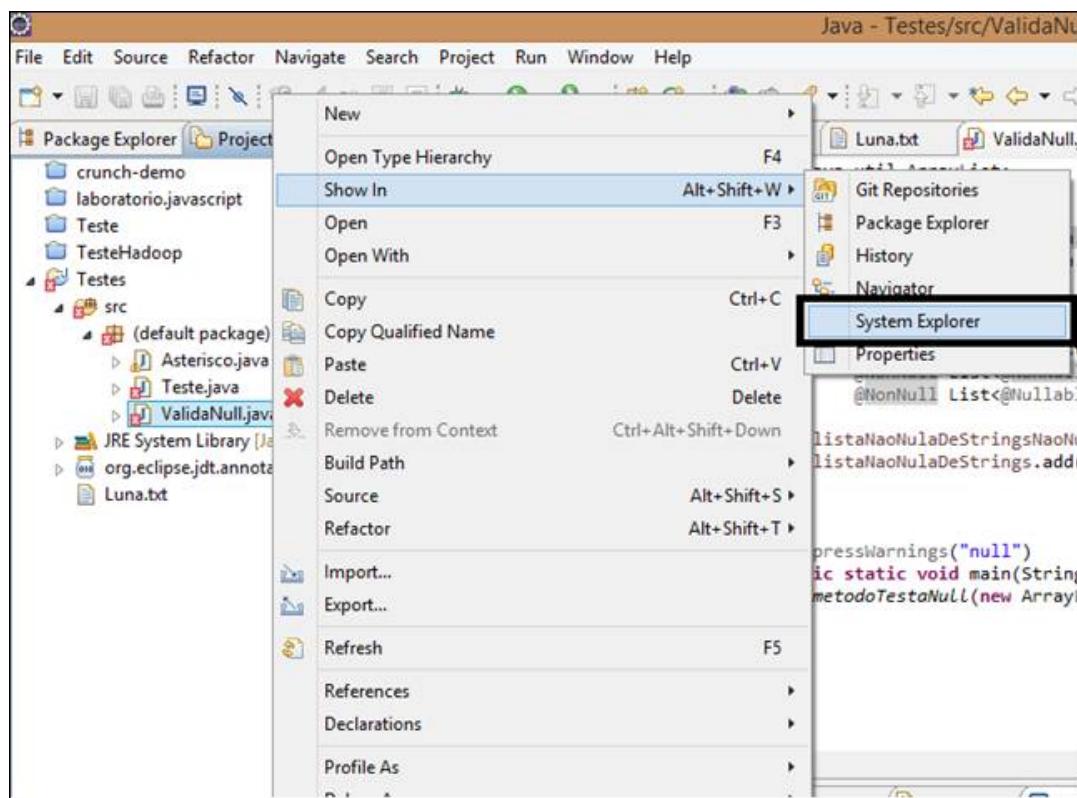
A importação de projeto no Eclipse Luna também trouxe uma novidade: a filtragem de projetos conflitantes durante o processo. Conforme a **Figura 8** apresenta, você pode esconder os projetos que já existem no workspace, o que é muito útil, principalmente para quem trabalha com várias versões de um mesmo projeto ou integrações Maven de projetos dependentes.



**Figura 8.** Importação de Projetos.

Outro recurso extremamente útil está na integração da IDE com o sistema operacional onde ela está sendo executada. Várias vezes nos deparamos com a necessidade de acessar o local no sistema de arquivos onde determinado artefato do projeto se localiza. Isso ocorre com frequência quando, por exemplo, queremos chegar ao arquivo fonte de uma classe Java, de um arquivo de propriedades ou até mesmo ao local de um diretório qualquer. Para isso, antigamente era necessário sair da IDE e navegar pelo Windows Explorer, por exemplo, até o local desejado.

Agora o Eclipse oferece a possibilidade de, por dentro da própria IDE, o desenvolvedor ter acesso direto aos arquivos e diretórios do sistema de arquivos do sistema operacional. Como mostra a **Figura 9**, ao clicar com o botão direito em uma pasta, pacote, classe ou qualquer outro arquivo, aparecerá o menu *Show in*, que foi aprimorado com a opção *System Explorer*. Ao ser selecionada, esta opção irá abrir o local (dentro do sistema de arquivos do SO, como o Windows Explorer, Nautilus, etc.) onde o elemento indicado se encontra.



**Figura 9.** Integração para navegação ao sistema de arquivos.

Outra pequena atualização feita no Eclipse Luna está relacionada ao *Quick Access*, acessível através da barra de ferramentas do Eclipse. Agora ele pode ser escondido e acessado apenas como pop-up, a partir das teclas de atalho *CTRL + 3*. Além disso, a numeração das linhas dos arquivos com texto aparecerá sempre por padrão, podendo ser configurada posteriormente conforme o gosto do usuário.

## Total suporte ao Java 8

Até agora foram apresentadas funcionalidades e recursos gerais introduzidos no Eclipse Luna para auxiliar no desenvolvimento de aplicações Java como um todo. A partir deste tópico serão apresentadas algumas funcionalidades específicas para a versão 8 da linguagem Java que mostram como o Eclipse Luna se atualizou para prover recursos que atendam às necessidades desta nova versão.

Inicialmente, é possível notar no Java Editor que a IDE já é capaz de identificar os blocos de código implementados pelo programador que fazem uso de recursos da versão 8 ou superior do Java, como por exemplo, expressões Lambda. E, caso o projeto ainda não esteja configurado para ser compilado para a versão 8, o usuário é alertado através de um novo Quick Fix, que permite essa mudança de compilação e configuração do seu projeto para a versão mais adequada do Java (veja a **Figura 10**). Mas isso é apenas o início.

Vamos nos aprofundar nesses recursos!



**Figura 10.** Quick Fix para atualizar versão do Java.

Provavelmente a maior mudança, e que mais trouxe impacto ao Java, foi a inclusão das Expressões Lambda. Por isso, o Eclipse não poderia deixar de oferecer uma gama de facilidades para programar com esse novo recurso. De maneira superficial, uma expressão Lambda pode ser entendida como um método ou uma funcionalidade de uma classe anônima.

Métodos de classe anônima são aqueles que criamos diretamente dentro do corpo da interface instanciada. Isto elimina a necessidade de termos que criar uma classe implementando a interface e posteriormente sobrescrever os métodos abstratos nela definidos. Apenas para contextualização, na **Listagem 1** apresentamos um exemplo de implementação de classe anônima que herda da interface **Runnable** a obrigação de implementar o método abstrato **run()**.

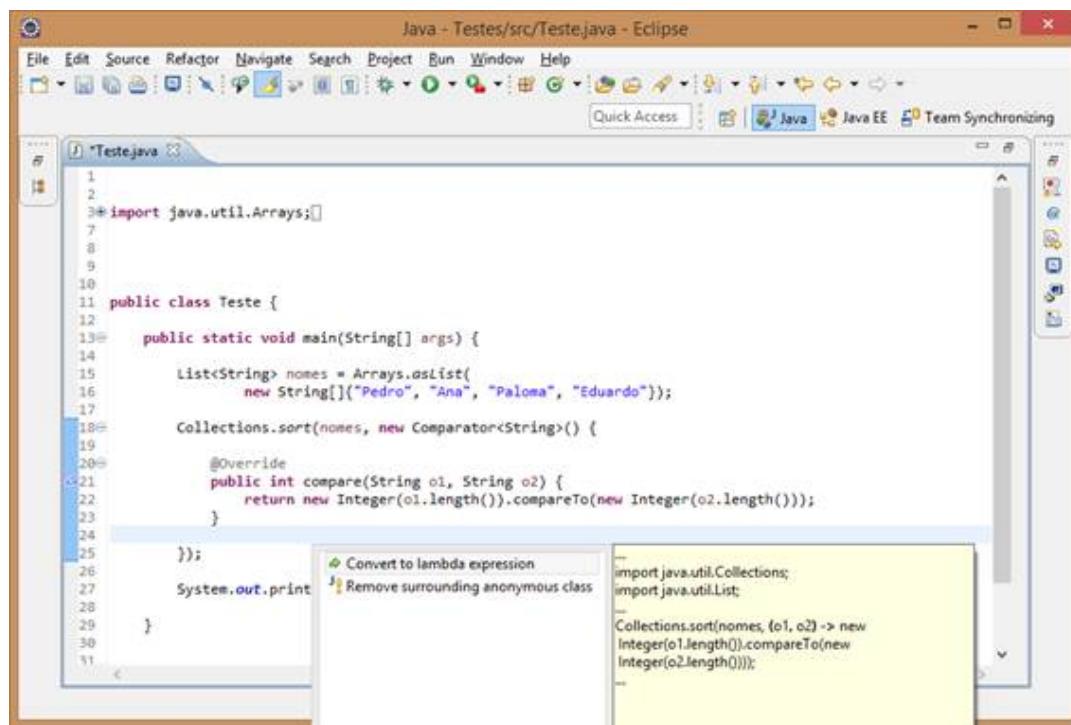
**Listagem 1.** Exemplo de codificação de método de uma classe anônima.

```
public class Teste {
    public static void main(String[] args) {
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("Método run() implementado em classe anônima");
            }
        });
        t.run();
    }
}
```

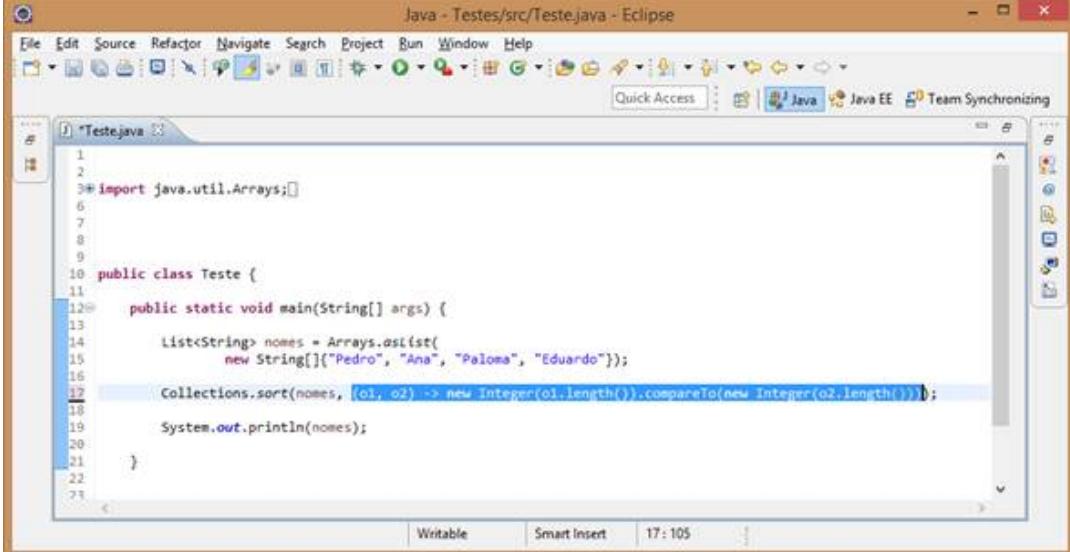
```
}
```

Dentro do Eclipse Luna, com o Java 8 configurado, é oferecida a possibilidade de converter o corpo de uma classe anônima em uma expressão Lambda e vice-versa, sem nenhum trabalho adicional. Como apresentado na **Figura 11**, a implementação de uma classe anônima a partir da interface **Comparator** é convertida em apenas uma linha de expressão Lambda ao utilizar o Quick Assist **CTRL+1** (veja a **Figura 12**).

Para alguns o código resultante pode parecer pouco legível ou comprehensível, devido à não familiaridade com expressões Lambdas, que antes não tinham suporte dentro do Java. Porém, com o tempo e o costume ao uso das novas sintaxes o desenvolvedor terá mais poder e opções para escolher a melhor e mais produtiva maneira de implementar seu código.



**Figura 11.** Conversão de classe anônima para expressão Lambda.



The screenshot shows the Eclipse Luna interface with a Java file named 'Teste.java' open. The code contains a main method that sorts a list of names using a Lambda expression:

```

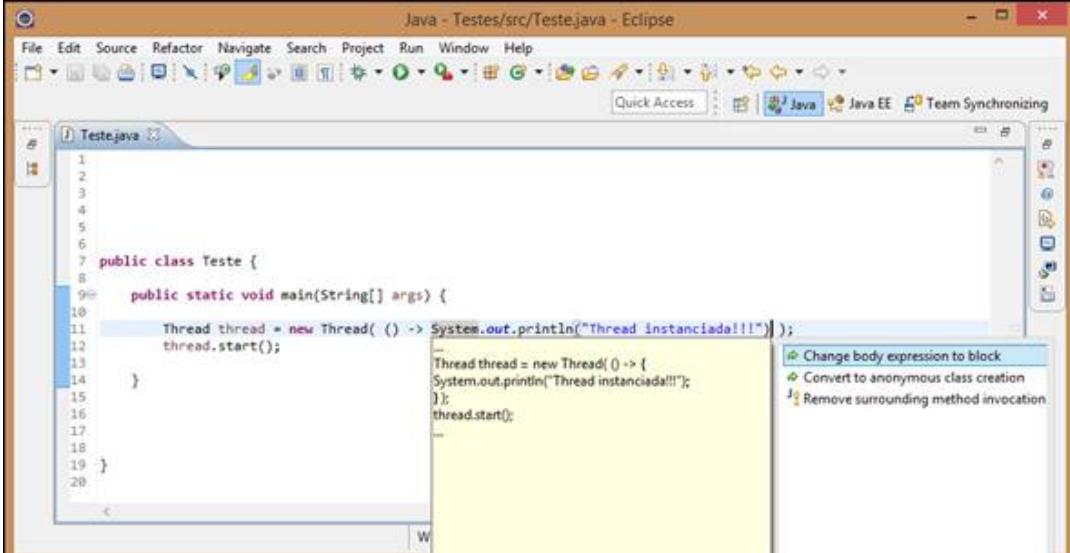
1
2
3+import java.util.Arrays;[]
4
5
6
7
8
9
10 public class Teste {
11
12     public static void main(String[] args) {
13
14         List<String> nomes = Arrays.asList(
15             new String[]{"Pedro", "Ana", "Paloma", "Eduardo"});
16
17         Collections.sort(nomes, (o1, o2) -> new Integer(o1.length()).compareTo(new Integer(o2.length())));
18
19         System.out.println(nomes);
20
21     }
22
23 }

```

The line `Collections.sort(nomes, (o1, o2) -> new Integer(o1.length()).compareTo(new Integer(o2.length())));` is highlighted in blue, indicating it has been converted into a Lambda expression.

**Figura 12.** Classe anônima convertida para expressão Lambda.

Através do Quick Assist **CTRL + 1** é possível ainda converter o corpo de uma expressão Lambda em um bloco de código (veja a **Figura 13**), para melhor organização ou preferência de desenvolvimento. E o contrário também é válido, ou seja, converter um bloco de código de volta para uma expressão Lambda.



The screenshot shows the Eclipse Luna interface with the same 'Teste.java' file. A Lambda expression is selected in the code editor:

```

1
2
3
4
5
6
7
8
9+ public class Teste {
10
11     public static void main(String[] args) {
12
13         Thread thread = new Thread( () -> System.out.println("Thread instanciada!!!"));
14         thread.start();
15
16     }
17
18 }

```

A context menu is open over the Lambda expression, showing options: **Change body expression to block**, **Convert to anonymous class creation**, and **Remove surrounding method invocation**.

**Figura 13.** Conversão de expressão Lambda em um bloco de código.

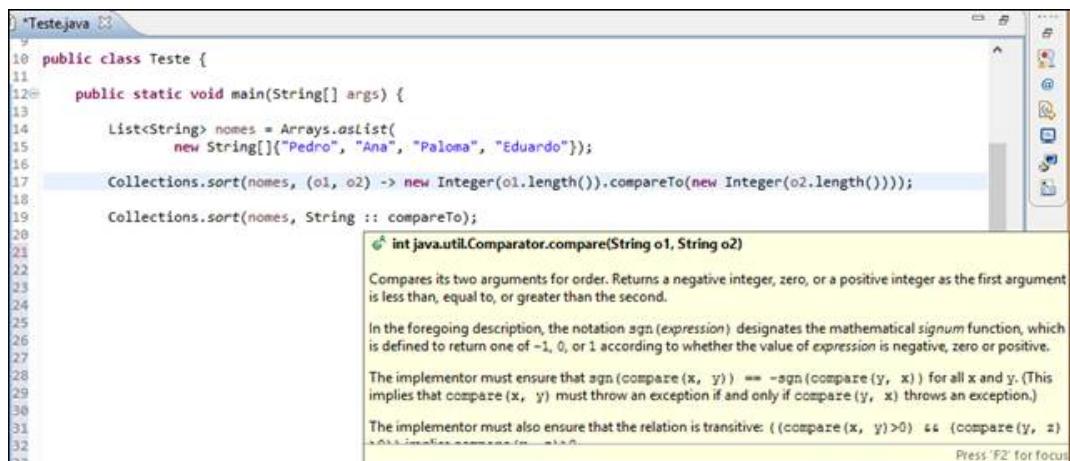
Outro recurso que faz parte das novidades do Java 8 são as Interfaces Funcionais. A partir disso, desenvolver com coleções, por exemplo, ficou muito mais prático, o que gera mais facilidade e agilidade quando trabalhamos, principalmente, com grandes estruturas de dados, uma vez que com interfaces funcionais uma série de métodos agregadores (*join*, *count*, *distinct*, etc.) já está pronta para realizar diferentes operações comuns ao dia a dia.

Note que o foco não é detalhar o que é e como funcionam as interfaces funcionais. Aqui, apenas considere que com a sua criação a quantidade de código gerado pode diminuir bastante e dessa forma é abstraída uma série de informações ao programador.

A partir da nova versão do Java, um método implementado por uma interface funcional pode ser chamado através de uma expressão Lambda (**Figura 14**, linha 17), de uma referência ao método pela sintaxe '::' (**Figura 14**, linha 19) ou até mesmo pela codificação tradicional, isto é, sem utilizar os novos recursos de sintaxe. Devido a essas possibilidades, a IDE precisa estar preparada para oferecer todos os recursos informativos (como warnings) ou corretivos (como erros de compilação por parâmetros preenchidos incorretamente) pertinentes ao método acessado.

A **Figura 14** apresenta a descrição com a assinatura e javadoc do método **compare()**, da agora interface funcional (fruto da anotação **@FunctionalInterface** criada no Java 8)

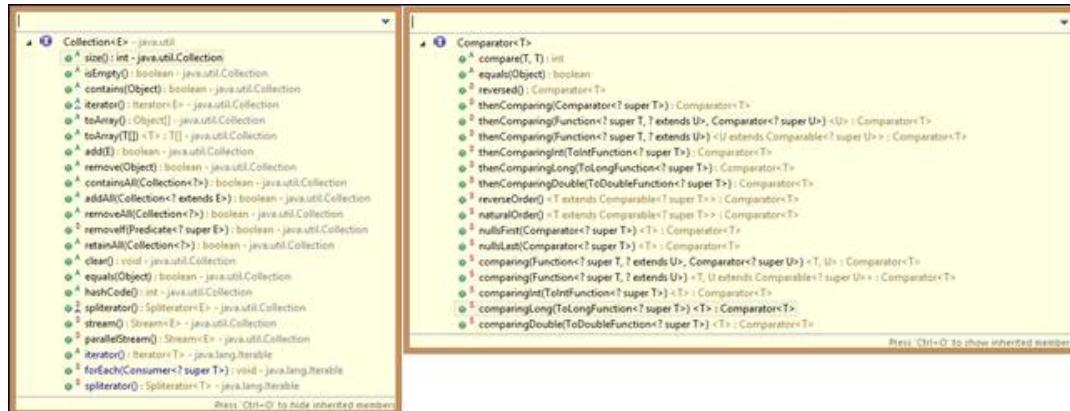
**Comparator**, utilizado pelo o método **sort()** da classe **Collections**. Caso houvesse algum erro de compilação, como por exemplo, um tipo de argumento de uma classe inválida passado ao chamar o método ou algum valor de retorno de método incompatível com a assinatura da funcionalidade, a IDE também será capaz de identificar e alertar o programador oferecendo as devidas sugestões de ajustes.



**Figura 14.** Visualização do método implementado pela interface funcional.

Após a criação das Interfaces Funcionais, as interfaces passaram a exercer um papel ainda mais importante na linguagem Java, pois agora, além de oferecer as assinaturas dos métodos abstratos para serem implementados pela(s) classe(s) filha(s), elas fornecem implementações concretas de métodos, que podem ser default ou estático. Devido a isso o Eclipse Luna possui um pequeno, porém útil recurso informativo ao programador que utilizar views como *Outline*, *Search*, *Type Hierarchy*, *Quick Outline*, etc. Através das letras A (*Abstract*), D (*Default*) e S (*Static*) a IDE informa ao programador se o método apresentado da interface se refere a um método abstrato (que obriga a implementação pela classe filha), default ou estático (que já possuem implementações na própria interface funcional). A **Figura 15** mostra o detalhe das letras informativas para os métodos de

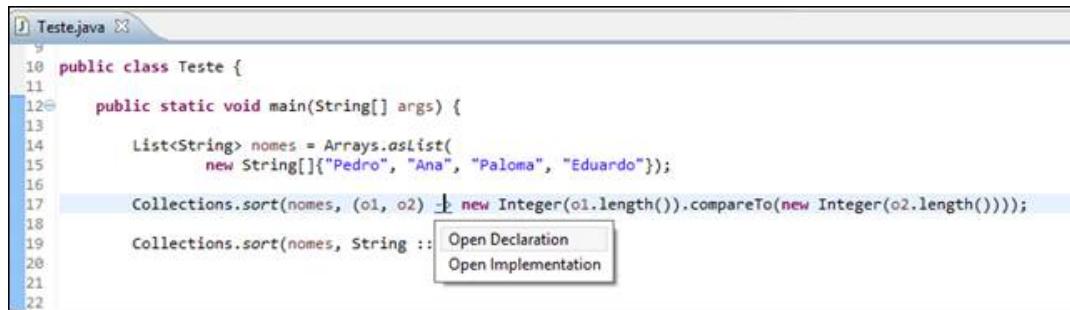
interfaces funcionais, detalhamento esse que pode ser obtido acionando as teclas de atalho *CTRL+3* dentro de alguma interface Java.



**Figura 15.** Mais informações sobre os métodos das interfaces.

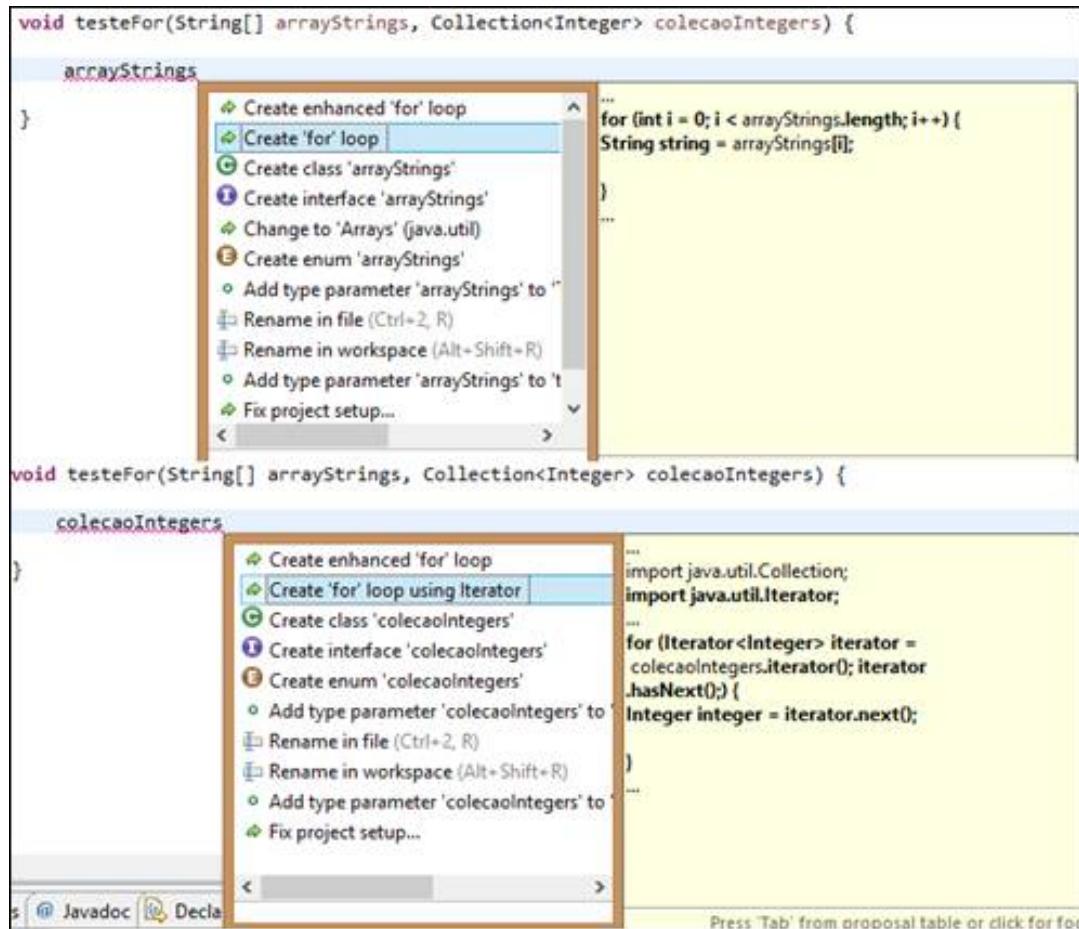
E não é porque se trata de um novo conceito de programação que as antigas facilidades do Eclipse deixarão de acompanhar. Da mesma forma que você consegue visualizar o método implementado pela interface funcional, também é possível navegar até a sua declaração e/ou demais implementações existentes no projeto por outras classes ou interfaces que estendam a interface funcional em questão, conforme demonstra a **Figura 16**.

Dentro do editor Java existe ainda uma série de outras pequenas novidades visuais. No decorrer das imagens do artigo, por exemplo, você pode perceber que a coloração das variáveis locais e parâmetros de método agora é padrão dentro do Eclipse, ajudando a destacar melhor o código.



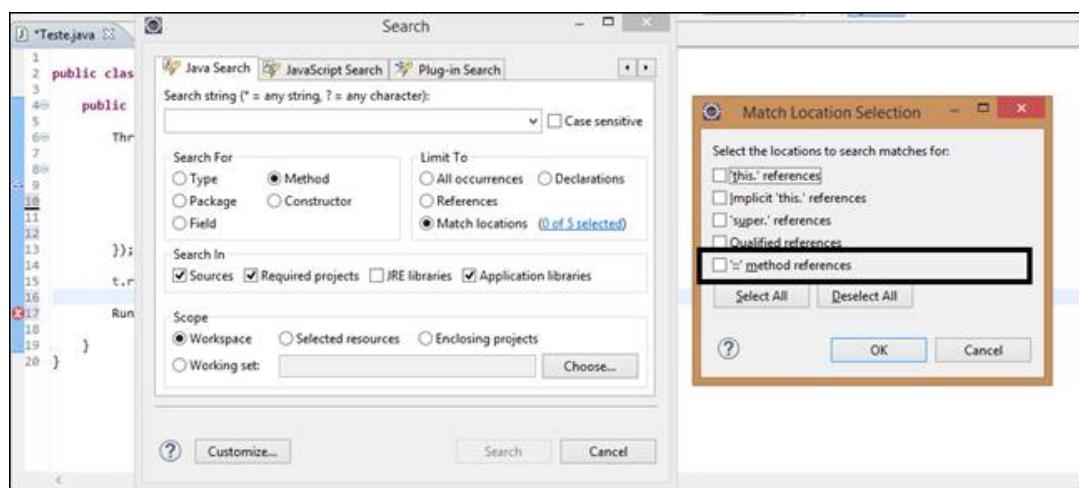
**Figura 16.** Navegação para o método implementado pela interface funcional.

Além dos Quick Assists analisados para as Expressões Lambda, referências de métodos e classes anônimas, foram adicionados quick fixes para auxiliar na criação de laços **for** a partir de arrays e coleções, como apresenta a **Figura 17**. Assim, ao utilizar o *CTRL + 1* em uma referência de coleção ou array, por exemplo, o Eclipse apresentará uma série de possibilidades para se trabalhar com a estrutura de dados.



**Figura 17.** Facilitador para a criação de laço.

Novos recursos também foram adicionados à ferramenta de busca do Eclipse. Como a sintaxe do Java 8 foi atualizada para contemplar a expressão `::`, utilizada para fazer referências a métodos, é natural que a IDE incorpore no Java Search (acessível pelas teclas de atalho *CTRL+H*) mais uma possibilidade de filtro (`:: 'method references') nos critérios de localização de métodos, como expõe a **Figura 18**.

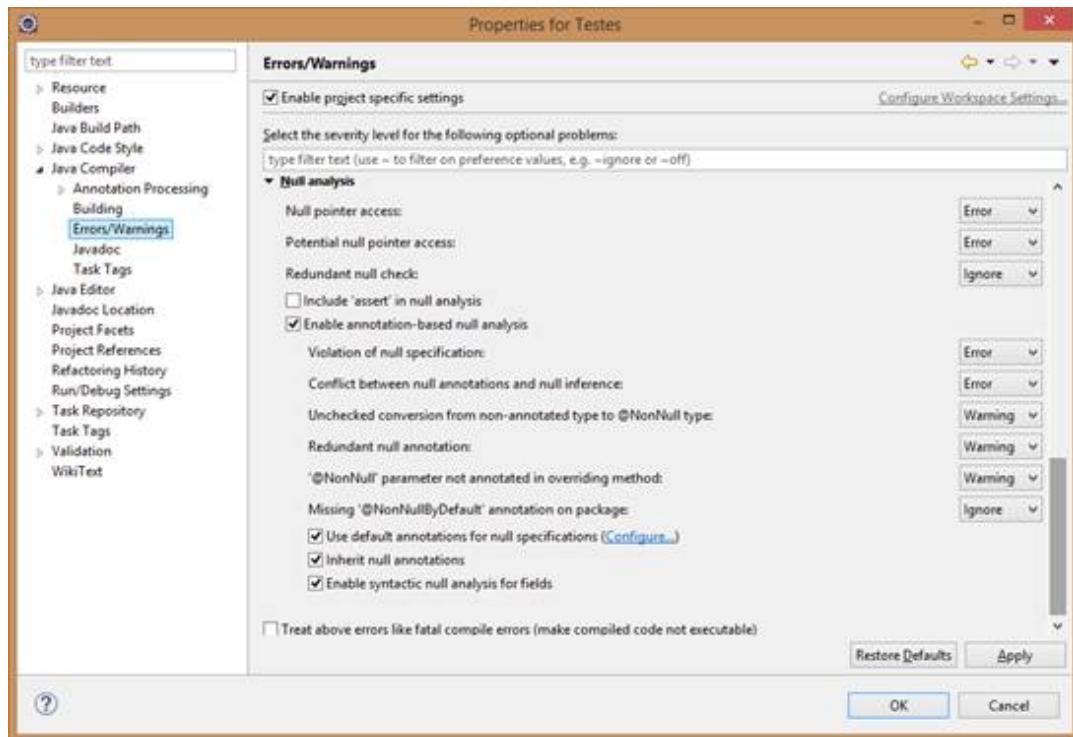


**Figura 18.** Novo filtro para localização de métodos por referência.

Acompanhando o grupo de novas especificações que formam a versão 8 da linguagem Java, existe uma em específico que contempla as Type Annotations: a JSR-308. É por causa dessa especificação que a verificação de exceções de ponteiros nulos (o famoso e mais comum erro de execução de códigos Java, o **NullPointerException**) é elevada a um novo patamar na ferramenta de desenvolvimento, pois viabiliza a utilização das Null Type Annotations do Eclipse Luna. Estas anotações ajudam o desenvolvedor a se certificar da segurança do seu código em relação a possíveis acessos nulos em parâmetros, variáveis, argumentos, etc.

A primeira coisa a ser feita para testar esse recurso é configurar o seu projeto para habilitar o que é chamado, dentro do Eclipse, de **Null Analysis**. Isto pode ser feito acessando o menu *Project > Properties > Java Compiler > Error/Warnings > Null analysis*, conforme apresentado na **Figura 19**.

As Type Annotations, introduzidas no Java 8 pela JSR-308, trazem para a linguagem novas opções para o uso de anotações, pois nas versões anteriores era possível utilizar anotações apenas nas declarações de classes, atributos, métodos, etc. Com o advento das Type Annotations, passa a ser possível anotar os tipos de classe em qualquer lugar que eles forem utilizados, como por exemplo, em listas de parâmetros, argumentos de métodos, declarações de variáveis, generics, etc.



**Figura 19.** Configurando Null analysis.

Efetuada a configuração, já podemos utilizar os recursos de análise de ponteiros nulos do Eclipse. Apenas para demonstração, a **Listagem 2** apresenta um código que utiliza dois exemplos das Null Type Annotations: **@NonNull** e **@Nullable**. Estas anotações servem, respectivamente, para avaliar os momentos em que parâmetros e argumentos de métodos são utilizados, alertando o usuário em tempo de compilação das possibilidades de ocorrência de erros resultantes de elementos nulos; ou possibilitando a utilização de elementos nulos como parte da lógica da aplicação.

**Listagem 2.** Código para testar as anotações @NonNull e @Nullable.

```
import java.util.ArrayList;
import java.util.List;
import org.eclipse.jdt.annotation.NonNull;
import org.eclipse.jdt.annotation.Nullable;

public class ValidaNull {

    private static void metodoTestaNull(
        @NonNull List<@NonNull String> listaNaoNulaDeStringsNaoNulas,
        @NonNull List<@Nullable String> listaNaoNulaDeStrings) {

        listaNaoNulaDeStringsNaoNulas.add(null);
        listaNaoNulaDeStrings.add(null);
    }

    @SuppressWarnings("null")
    public static void main(String[] args) {
        metodoTestaNull(new ArrayList<String>(), null);
    }
}
```

De acordo com a **Figura 20**, percebe-se que onde há a anotação **@Nullable** não é mostrada irregularidade de compilação, uma vez que essa anotação sinaliza que existe a possibilidade de, por exemplo, um método receber um parâmetro nulo ou uma lista receber um registro nulo.

Porém, em outros dois pontos do código, nos quais foi empregada a anotação **@NonNull**, erros de compilação ocorrem e alertam o programador quando é tentado passar um parâmetro nulo a um método ou inserir um elemento nulo em uma lista, uma vez que essa anotação foi criada para ser empregada em pontos que devem impedir o recebimento de elementos nulos.

Esse recurso é especialmente útil para sistemas que possuem trechos de código extremamente críticos (que não podem falhar em tempo de execução), nos quais deve ser garantido que nenhum **NullPointerException** seja lançado por falha de codificação da funcionalidade. Dessa forma, a análise de elementos nulos trata esses possíveis erros em tempo de compilação como se fossem erros de sintaxe, deixando o programador mais atento, por exemplo, ao chamar um método.

The screenshot shows two code snippets in Eclipse Luna's code editor. Both snippets are identical except for the class name and the position of the error markers.

```

import java.util.ArrayList;
import java.util.List;

import org.eclipse.jdt.annotation.NonNull;
import org.eclipse.jdt.annotation.Nullable;

public class ValidaNull {
    private static void metodoTestaNula(
        @NonNull List<@NonNull String> listaNaoNulaDeStringsNaoNulas,
        @NonNull List<@Nullable String> listaNaoNulaDeStrings) {
        listaNaoNulaDeStringsNaoNulas.add(null);
        listaNaoNulaDeStrings.add(null); // Null type mismatch: required '@NonNull String' but the provided value is null
    }

    @SuppressWarnings("null")
    public static void main(String[] args) {
        metodoTestaNula(new ArrayList<String>(), null);
    }
}

import java.util.ArrayList;
import java.util.List;

import org.eclipse.jdt.annotation.NonNull;
import org.eclipse.jdt.annotation.Nullable;

public class ValidaNull {
    private static void metodoTestaNula(
        @NonNull List<@NonNull String> listaNaoNulaDeStringsNaoNulas,
        @NonNull List<@Nullable String> listaNaoNulaDeStrings) {
        listaNaoNulaDeStringsNaoNulas.add(null);
        listaNaoNulaDeStrings.add(null); // Null type mismatch: required '@NonNull List<@Nullable String>' but the provided value is null
    }

    @SuppressWarnings("null")
    public static void main(String[] args) {
        metodoTestaNula(new ArrayList<String>(), null);
    }
}

```

In both snippets, there are two error markers: one in the first method's body and one in the main method's body. The error marker in the first method's body points to the line `listaNaoNulaDeStrings.add(null);` with the message "Null type mismatch: required '@NonNull String' but the provided value is null". The error marker in the main method's body points to the line `metodoTestaNula(new ArrayList<String>(), null);` with the message "Null type mismatch: required '@NonNull List<@Nullable String>' but the provided value is null". Both error markers include the instruction "Press F2 for focus".

**Figura 20.** Erros apresentados após a habilitação do recurso Null Analysis.

Enfim, vale ressaltar que além dos recursos apresentados na parte prática desse artigo, o Eclipse Luna possui muitas outras novidades e aprimoramentos nesta nova release; mas que em qualquer outra que já tenha sido lançada anteriormente. Devido a isso, certamente é necessário se aprofundar mais nas atualizações existentes.

Como o foco do artigo foi voltado para o programador que atua basicamente com código Java, para os próximos passos vale a pena se aprofundar em conhecer as integrações e melhorias gráficas que surgiram no Eclipse para análise e desenho de software, como por exemplo, a atualização do framework EMF e a integração com diagramas UML 2.5.

Também não se pode esquecer que as novidades nativas do Eclipse Luna são evidentemente as que recebem maior destaque, porém vários plug-ins de terceiros merecem atenção para serem testados e analisados. Até mesmo durante a escrita deste artigo plug-ins estão sendo confeccionados para introduzir alguma melhoria ou integração, afinal, nenhum fornecedor de ferramenta open source para desenvolvimento Java pode se dar ao luxo de não pensar em alguma forma de integração junto ao projeto Eclipse.

Um melhor entendimento dos recursos apresentados será possível apenas com um maior aprofundamento a partir do uso da ferramenta, e não só com estudos teóricos. Dessa forma, erros, dúvidas e outras formas de funcionamento dos recursos apresentados serão descobertos. Esse maior aprofundamento pode ser feito consultando as referências na seção **Links** e visitando a página oficial da documentação do Eclipse Luna, onde todos os highlights estão explicados conceitualmente, desde os motivos de sua concepção até sua execução na prática.

Estão prontos para (re)descobrir o novo Eclipse Luna? A hora é agora!

## Links

### **Release do Eclipse Luna.**

<https://projects.eclipse.org/releases/luna>

### **Newsletter do Eclipse.**

[https://www.eclipse.org/community/eclipse\\_newsletter/2014/june/article1.php](https://www.eclipse.org/community/eclipse_newsletter/2014/june/article1.php)

### **Solicitação 8009 Bugzilla do Eclipse.**

[https://bugs.eclipse.org/bugs/show\\_bug.cgi?id=8009](https://bugs.eclipse.org/bugs/show_bug.cgi?id=8009)

### **Blog do Eclipse.**

<http://eclipsesource.com/blogs>

### **Documentação do Eclipse.**

<http://help.eclipse.org/luna/index.jsp>

### **Download do Eclipse.**

<https://www.eclipse.org/downloads>



Wellington Ferro

Técnico em Telecomunicações pela Centro Federal de Educação Tecnológica de São Paulo, Bacharel em Ciência da Computação pela Universidade Anhembi Morumbi e atua na área de Engenharia de Software trabalhando profissionalmente com a [...]

Repare que ao rodar a classe **TestaAtendimento**, o objeto que tratará a requisição e atenderá a expectativa do negócio será identificado de forma automática. Se um objeto não atender a expectativa do negócio, o próximo objeto será imediatamente invocado, porém sem se conhecerem, pois estão totalmente desacoplados um do outro.

**Listagem 20.** Cadeia de responsabilidades – Teste.

```

1 package br.com.java.magazine.test;
2
3 import br.com.java.magazine.bean.DiretorBean;
4 import br.com.java.magazine.bean.InspetorDeAlunosBean;
5 import br.com.java.magazine.bean.ProfessorDeHistoriaBean;
6 import br.com.java.magazine.bean.ProfessorDeLiteraturaBean;
7 import br.com.java.magazine.bean.Requisicao;
8 import br.com.java.magazine.bean.Responsavel;
9 import br.com.java.magazine.business.impl.CadeiaDeResponsabilidades;
10
11 public class TestaAtendimento {
12
13     public static void main(String[] args) {
14
15         CadeiaDeResponsabilidades atendimento = new CadeiaDeResponsabilidades();
16
17         Responsavel responsavel = new Responsavel(
18                 new ProfessorDeHistoriaBean
19                         (Requisicao.AQUISICAO_DE_CADEIRA.getDescricao()),
20                 new ProfessorDeLiteraturaBean
21                         (Requisicao.AQUISICAO_DE_CADEIRA.getDescricao()),
22                 new InspetorDeAlunosBean
23                         (Requisicao.AQUISICAO_DE_CADEIRA.getDescricao()),
24                 new DiretorBean(Requisicao.AQUISICAO_DE_CADEIRA.getDescricao()));
25
26         atendimento.solicitaAtendimento(responsavel);
27     }
28 }
```

O exemplo abordado é simples e cotidiano, com o objetivo de deixar o mais claro possível quando e como utilizar o pattern *Chain Of Responsibility*. Existem outros exemplos que podem ser citados em que a adoção do *Chain Of Responsibility* como solução faria sentido. Casos mais complexos como aprovações de planos de saúde, por exemplo. Neste caso, imagine que um corretor de seguros pretende realizar a venda de uma apólice para um possível asssegurado e reúne as informações necessárias, porém não é capaz de definir a aprovação da aquisição para o solicitante, pois ele não possui essa responsabilidade.

Na realidade, dentro de um sistema corporativo o corretor de seguros seria o cliente. Como não possui a responsabilidade de aprovar a aquisição de uma apólice, ele encaminha a solicitação para que outras alçadas o façam. De fato, ele nem sabe quem aprovará e pouco importa para ele. De acordo com o grau de complexidade das informações do solicitante, a aprovação ainda pode ser “*delegada*” para alçadas maiores. Esta é uma excelente alternativa dentro de um sistema corporativo para atender ao que é conhecido como *tomada de decisão*. Podemos ter vários objetos de negócio, totalmente desacoplados uns dos outros, contendo cada qual a sua devida regra de negócio e a capacidade de tomar uma decisão de acordo com a sua competência (Responsabilidade). Assim, se um desses objetos precisar sofrer qualquer alteração na regra, ou se uma nova regra precisar ser criada, os outros objetos não terão qualquer conhecimento. E se um objeto não for capaz de tratar e tomar uma decisão, o próximo

objeto será chamado imediatamente sem qualquer intervenção interna ou externa. Será automático.

Ademais, o design pattern *Chain of Responsibility* é um recurso que pode auxiliar outros frameworks como o Spring, por exemplo, que trabalha muito bem com injeção de dependência e inversão de controle (*IoC*), mas que precisa de uma implementação de código inteligente e elegante para solucionar problemas como o citado anteriormente. Sendo assim, ao se deparar com situações como as analisadas, não deixe de considerar esse padrão de projeto como uma ótima opção para o seu código.

## Links

### **OODesign.com – Object Oriented Design.**

<http://www.oodesign.com/>

### **Chain of Responsibility.**

[http://sourcemaking.com/design\\_patterns/chain\\_of\\_responsibility](http://sourcemaking.com/design_patterns/chain_of_responsibility)

### **Chain of Responsibility in Java: Before and after.**

[http://sourcemaking.com/design\\_patterns/chain\\_of\\_responsibility/java/1](http://sourcemaking.com/design_patterns/chain_of_responsibility/java/1)

### **Chain of Responsibility Pattern.**

[http://www.tutorialspoint.com/design\\_pattern/chain\\_of\\_responsibility\\_pattern.htm](http://www.tutorialspoint.com/design_pattern/chain_of_responsibility_pattern.htm)

### **Java Design Patterns.**

<http://www.javaworld.com/blog/java-design-patterns>

### **GoF Design Patterns - with examples using Java and UML2.**

<http://www.usp.br/thienne/coo/material/GoFDesignPatterns.pdf>



Arthur Gomes

É formado em Análise de Sistemas pela UNIESP, trabalha com desenvolvimento de sistemas faz aproximadamente dez anos, é especialista em plataforma Java, certificado em várias soluções IBM e Oracle como o InfoSphere DataStage 8.5, W [...]



[www.devmedia.com.br](http://www.devmedia.com.br)

[versão para impressão]

Link original: <http://www.devmedia.com.br/articles/viewcomp.asp?comp=32168>

# Design Patterns: na teoria e na prática – Parte 1

Nesse artigo você entenderá e aplicará os padrões de projeto em seus sistemas.

---

## Fique por dentro

Programadores mais novos atualmente se preocupam em tornarem-se especialistas em determinado framework, ignorando a complexidade envolvida por trás e consequentemente deixam de programar orientado a objetos em total plenitude, limitando-se apenas a criar trechos de código que se conectam e são executados por esses mesmos frameworks. A consequência disso é a recorrência cada vez maior a estratégias procedurais para resolver problemas nem tanto complexos. É muito fácil encontrar uma funcionalidade que pode conter três ou mais condições para obter um resultado dentro de qualquer sistema, e que muitas vezes na pressa, fazem com que o desenvolvedor opte em criar um "encadeamento" (IF-ELSE), ignorando completamente um dos princípios básicos da OO, a Abstração, muitas vezes forçando um alto acoplamento. Dito isso, este artigo demonstrará como evitar o alto acoplamento, identificando um problema dentro de um cenário apresentado e aplicando o design pattern correto para solucioná-lo, sem o auxílio de qualquer framework.

Em 1994, antes mesmo de James Gosling apresentar a linguagem de programação Java para o mundo, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, também conhecidos como "The Gang of Four" lançaram o livro "*Design Patterns: Elements of Reusable Object-Oriented Software*". Para muitos evangelistas da linguagem orientada a objetos, esse foi o divisor de águas entre programar orientado a objetos e compreender o conceito real de programar orientado a objetos.

Nesse livro foram catalogados vinte e três padrões de projeto divididos em três categorias:

- **Padrões de criação** – Abstract Factory, Builder, Factory Method, Prototype e Singleton;
- **Padrões estruturais** – Adapter, Bridge, Composite, Decorator, Facade, Flyweight e Proxy;
- **Padrões comportamentais** – Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method e Visitor.

Atualmente, quase 100% dos sistemas desenvolvidos em Java com o padrão MVC utilizam-se de frameworks de apoio, sendo mais comuns o JSF 2 para as camadas de apresentação e o Hibernate para a camada de persistência. Isso sem falar que dentro de uma aplicação, na camada de negócio, pode-se incluir o Spring como apoio para injeção de dependência e inversão de controle (*IoC*) ou o próprio EJB 3 aliado ao CDI para obter a mesma eficiência do Spring na parte de *IoC*.

Esses frameworks acabam “mascarando” boa parte da complexidade do que é programar orientado a objetos, fazendo com que o desenvolvedor se limite a implementar métodos, muitas vezes apenas encaixando-os nos pontos certos, ligando uma ponta à outra. É isso mesmo. Ele deixa de ser um desenvolvedor e se torna um implementador.

O trecho a seguir toma uma afirmação do próprio livro (*Design Patterns: Elements of Reusable Object-Oriented Software*), coincidentemente já quase que prevendo um fato que ocorre nos dias de hoje:

*“Um framework dita a arquitetura da sua aplicação. Ele define a arquitetura de forma geral, sua subdivisão em classes e objetos, a responsabilidade entre eles, como os objetos colaboram. Um framework predefine esses padrões de desenvolvimento para você. O desenvolvedor/implementador pode se concentrar em detalhes da aplicação... GoF Pág. 26.”.*

A intenção do uso de um framework é sem dúvida alguma acelerar o desenvolvimento e facilitar a manutenção do sistema, mas, por outro lado, muitos desenvolvedores acabam se tornando especialistas em determinado framework, mas inexperientes em programar orientado a objetos em sua plenitude.

A consequência disso é que quando surge um problema que requer o mínimo de elegância na solução, acaba-se adotando uma estratégia procedural dentro de uma aplicação que deveria ser por si só totalmente orientada a objetos.

Com base nisso, a ideia deste artigo é mostrar como identificar um problema e reconhecer o design pattern que melhor se encaixa como solução. O primeiro tópico apresentará o design pattern *Chain Of Responsibility*. Esse padrão possui um alto grau de polimorfismo, é capaz de automatizar um processo e representa muito bem o quarto princípio da programação orientada a objetos: o Princípio da Segregação de Interfaces. Em seguida será apresentado o design pattern *Strategy*. Esse padrão é capaz de impor um baixo acoplamento e representa muito bem a boa prática de programar voltado a interface. Você notará que em nenhum dos tópicos apresentados será cogitado o uso de qualquer framework. O objetivo disso é ensinar ao leitor como identificar um problema e mostrar alternativas elegantes para solucioná-lo.

## Chain Of Responsibility

É o primeiro na lista dos padrões comportamentais, sendo extremamente fácil de implementar e que pode ser muito útil como solução adotada dentro de um sistema corporativo, pois impõe um alto grau de coerência ao código, força um baixo acoplamento, utiliza muito bem recursos de polimorfismo e é capaz de automatizar um processo sem encadeamentos desnecessários.

Esse design pattern promete acabar com o encadeamento excessivo, que é visto como uma das principais causas da ocorrência de alto acoplamento e até mesmo criando “baixa coerência” no código por conta de encadeamentos desnecessários e sem sentido.

O design pattern *Chain Of Responsibility* permite determinar quem será o objeto que irá tratar uma requisição em tempo de execução. Dessa forma, cada objeto pode tratar ou passar a responsabilidade para o próximo, construindo uma *cadeia de responsabilidades*.

O trecho a seguir toma a definição do livro *Patterns: Elements of Reusable Object-Oriented Software* em relação à intenção desse padrão: “*Evitar o acoplamento do remetente de uma solicitação. Encadear os objetos receptores, passando a solicitação ao longo da cadeia até que um objeto a trate... GoF Pág. 212.*”

Essa é a grande motivação para o uso desse pattern: acabar com as estruturas de decisão, criando uma cadeia de objetos. A responsabilidade é passada entre eles até finalmente ser tratada por quem tiver a devida competência.

## Características do Chain Of Responsibility

Em um sistema orientado a objetos, estes interagem entre si. Como já foi ressaltada anteriormente, a intenção do pattern é fazer com que o sistema determine qual o objeto que irá tratar uma requisição. Mas como o programador poderá identificar qual o melhor momento para a adoção do *Chain of Responsibility*?

Existem algumas características que podem ajudar a identificar se há necessidade de adoção ao uso do pattern para atender como solução de um determinado problema, a saber:

- Mais de um objeto pode tratar uma solicitação e o objeto que a tratará não precisa ser conhecido;
- O objeto que trata a solicitação deve ser escolhido automaticamente;
- Deve-se emitir uma solicitação para um dentre vários objetos, sem especificar explicitamente o receptor;
- O conjunto de objetos que pode tratar uma solicitação deve ser especificado dinamicamente.

Uma excelente maneira de se entender a motivação para o uso de determinado pattern é abordando um exemplo de uso real.

Quero uma cadeira...

O texto apresentado é fictício e foi elaborado como uma forma de simular uma situação em que a abordagem ao *Chain Of Responsibility* atenderia como solução de um problema.

Durante os meses de Outubro e Novembro é muito comum a procura por cursinhos pré-vestibular. Dessa forma é previsível que as salas estejam sempre lotadas e em algum momento ocorra uma falta de cadeiras para aqueles alunos que chegarem atrasados. Isso causa um problema que precisa de solução.

Um aluno chega atrasado à aula de história e logo se dá conta de que a sala está lotada e não há mais cadeiras para que ele possa sentar e acompanhar o curso. O aluno decide então pedir ao professor de história que consiga uma cadeira para ele. O professor de história tem a responsabilidade de lecionar história geral e não possui a responsabilidade de conseguir cadeiras novas. Vai à sala ao lado e pede ao professor de literatura que consiga uma cadeira nova. O professor de literatura tem a responsabilidade de lecionar literatura e também não possui a responsabilidade de conseguir cadeiras novas. Portanto vai ao corredor e pede ao inspetor de alunos que consiga uma cadeira nova para o professor de história. Já a responsabilidade do inspetor de alunos é de orientar sobre a localização das salas de aula. Ele não possui a responsabilidade de conseguir cadeiras novas. Dessa forma o inspetor de alunos se dirige ao diretor do cursinho e pede que consiga uma cadeira nova para o professor de história. O diretor, dentre várias responsabilidades, possui a responsabilidade de comprar cadeiras novas. Finalmente a cadeira é comprada e entregue ao professor de história para que o aluno consiga acompanhar a aula.

No texto apresentado, uma *cadeia* é formada e somente finalizada quando alguém que atende ao requisito esperado encerra o ciclo. Mas qual a melhor forma de representar o texto programaticamente? Aí começa o problema. Na maioria das vezes o programador esquece o que é programar OO e decide pela solução mais simples, porém às vezes mais custosa em termos de manutenção de código ou mesmo desempenho do sistema, tornando-se o que chamamos na orientação a objetos de *Anti-pattern*.

Um *Anti-pattern*, por sua vez, é o oposto a determinado padrão de projetos. Considerados como má prática, devem ser evitados como solução de contorno. São ineficientes, pouco produtivos e geram impactos muitas vezes imprevisíveis durante o ciclo de vida de um sistema.

A **Listagem 1** apresenta uma solução comum, porém longe de ser elegante. Um perfeito exemplo de Anti-pattern.

#### **Listagem 1.** Cadeia de responsabilidades – Solução sem elegância.

```

1 package br.com.javamagazine.test;
2
3 public class CadeiaDeResponsabilidades {
4
5     public String pesquisaResponsavel(String atendimento){
6
7         if(atendimento.equalsIgnoreCase("Professor de História")){
8             return "O professor de História explicou sobre a guerra dos Canudos.";
9         }else if(atendimento.equalsIgnoreCase("Professor de Literatura")){
10            return "O professor de literatura explicou sobre o barroco.";
11        }else if(atendimento.equalsIgnoreCase("Inspetor de Alunos")){
12            return "O inspetor de alunos indicou a localização de uma determinada sala de aula.";
13        }else if(atendimento.equalsIgnoreCase("Diretor")){
14            return "O diretor comprou uma cadeira nova.";
15        }else{
16            return null;
17        }
18    }
19
20 }
```

Pode-se observar que, na linha 5, o cliente possui o conhecimento sobre quem deve ser o responsável para atender ao pedido por uma nova cadeira. Entende-se que o parâmetro recebido no método é a definição desse responsável.

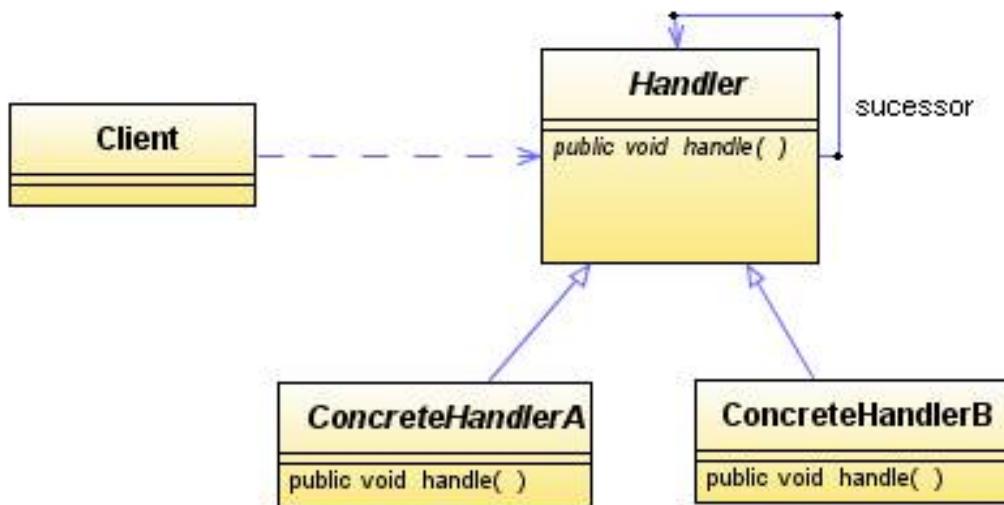
É possível notar que existe um encadeamento de condições no método. Note que são feitas várias tentativas até que se encontre o responsável correto, e se não encontrar ninguém que atenda ao parâmetro de entrada, o retorno é nulo, como mostrado na linha 16.

Quais os problemas encontrados nesse método? Podem-se listar alguns, como por exemplo, o fato de que ele só tende a crescer infinitamente. Sempre que outra responsabilidade for incluída na cadeia, uma nova condição será também incluída.

Pode-se citar também o fato de que as regras de negócio relativas a cada responsabilidade estariam incorporadas dentro do mesmo método. Isso é péssimo para manutenção, pois causa “alto acoplamento”, e, principalmente, esse método não deveria fazer nada além de “delegar” a responsabilidade para quem lhe for devido. Para o leitor mais atento, é possível reconhecer uma falha dentro do próprio entendimento do negócio. O estudante não sabe quem vai comprar a cadeira nova. Na verdade pouco importa para ele. O estudante tem uma necessidade, e é essa necessidade que deve ser a requisição. O objeto responsável por atender a essa necessidade não deveria ser conhecido. Dentro de um escopo correto, ele deveria ser requisitado, tratar a requisição e devolver a resposta ao cliente. Enfim, a estratégia apresentada na **Listagem 1** é extremamente “Procedural”.

A **Figura 1** apresenta um diagrama de classes que representa corretamente o que é o padrão *Chain Of Responsibility*. Observe que o diagrama é composto por três elementos (**Handler**, **ConcreteHandler** e **Client**) que são fundamentais dentro da estrutura e para o comportamento correto do pattern:

- **Handler:** Define uma interface de tratamento de solicitações;
- **ConcreteHandler:** Caracteriza as implementações que tratarão as requisições;
- **Client:** Quem inicia a chamada na cadeia de objetos.



**Figura 1.** Diagrama de classes do padrão *Chain Of Responsibility*.

## Desacoplando e padronizando o exemplo

Após entendido todo o conceito, somado ao diagrama de classes exibido, o objetivo agora é tornar o exemplo apresentado na **Listagem 1** um verdadeiro modelo do design pattern *Chain Of Responsibility*.

Como afirmado anteriormente, o cliente envia a necessidade como requisição. Ele não conhece quem, dentro de uma cadeia de responsabilidades, irá atendê-lo. Observe que a **Listagem 2** apresenta um **Enum**. Esse **Enum**, chamado de **Requisicao**, representará as responsabilidades que o nosso sistema dispõe.

O objetivo deste artigo é ensinar o leitor a resolver um problema específico e muitas vezes comum utilizando um design pattern. Para auxiliar esse estudo a IDE adotada nos exemplos foi o Eclipse na versão Luna 4.4.0, mas fica a critério do leitor qual IDE adotar, assim como a versão do Java a ser empregada.

#### **Listagem 2.** Cadeia de responsabilidades – Requisicao.

```

1 package br.com.javamagazine.enuns;
2
3 public enum Requisicao {
4
5     DUVIDA_SOBRE_A_GUERRA_DOS_CANUDOS(0,"Dúvida sobre a guerra dos Canudos"),
6     DUVIDA_SOBRE_BARROCO(1,"Dúvida sobre o Barroco."),
7     INFORMACAO_DE_LOCALIZACAO_DE_SALA(2,"Informação sobre localização de salas de aula."),
8     AQUISICAO_DE_CADEIRA(3,"Aquisição de cadeiras novas.");
9
10    private Integer id;
11    private String descricao;
12
13    private Requisicao(Integer id, String descricao) {
14        this.id = id;
15        this.descricao = descricao;
16    }
17
18    //GETTERS E SETTERS OMITIDOS

```

## Definindo os objetos

Para desacoplar completamente o código apresentado na **Listagem 1**, será necessário definir os objetos que irão interagir entre si, sendo que cada objeto dentro da *cadeia de responsabilidades* deverá ter uma responsabilidade.

Lendo novamente o texto e analisando o código da mesma listagem, é possível identificar quatro objetos que precisarão ser criados. De acordo com o texto, o professor de História é o primeiro elemento a ser acionado; logo, ele será um dos nossos objetos com a sua competência definida. Em seguida, o texto informa sobre a interação do professor de História com o professor de Literatura e esse com o inspetor de alunos, finalizando com o diretor do curso.

As **Listagens 3, 4, 5 e 6** apresentam o código dessas classes.

#### **Listagem 3.** Cadeia de responsabilidades – ProfessorDeHistoriaBean.

```

1 package br.com.javamagazine.bean;
2
3 public class ProfessorDeHistoriaBean {
4
5     private String competencia;
6
7     private ProfessorDeHistoriaBean(String competencia) {
8         this.competencia = competencia;

```

```

9      }
10
11     public String getCompetencia() {
12         return competencia;
13     }
14 }
```

**Listagem 4.** Cadeia de responsabilidades – ProfessorDeLiteraturaBean.

```

1 package br.com.javamagazine.bean;
2
3 public class ProfessorDeLiteraturaBean {
4
5     private String competencia;
6
7     private ProfessorDeLiteraturaBean(String competencia) {
8         this.competencia = competencia;
9     }
10
11    public String getCompetencia() {
12        return competencia;
13    }
14 }
```

**Listagem 5.** Cadeia de responsabilidades – InspetorDeAlunosBean.

```

1 package br.com.javamagazine.bean;
2
3 public class InspetorDeAlunosBean {
4
5     private String competencia;
6
7     private InspetorDeAlunosBean(String competencia) {
8         this.competencia = competencia;
9     }
10
11    public String getCompetencia() {
12        return competencia;
13    }
14 }
```

**Listagem 6.** Cadeia de responsabilidades – DiretorBean.

```

1 package br.com.javamagazine.bean;
2
3 public class DiretorBean {
4
5     private String competencia;
6
7     private DiretorBean(String competencia) {
8         this.competencia = competencia;
9     }
10
11    public String getCompetencia() {
12        return competencia;
13    }
14 }
```

## Criando uma composição

Nesse momento, os objetos necessários para a definição da cadeia de responsabilidades já estão criados. Como boa prática é interessante agora criar um novo objeto e compor com os quatro criados anteriormente. Essa estratégia é conhecida como *composição*, e nos permitirá trabalhar com um único objeto composto por todos. Repare na **Listagem 7** que a própria denominação da classe já define o objetivo dessa composição. Ela possui todos os objetos com alguma responsabilidade dentro do nosso negócio. Um desses objetos, se possuir a devida competência, tratará a requisição.

**Listagem 7.** Cadeia de responsabilidades – Composição.

```

1 package br.com.javamagazine.bean;
2
3 public class Responsavel {
4
5     private ProfessorDeHistoriaBean professorDeHistoriaBean;
6     private ProfessorDeLiteraturaBean professorDeLiteraturaBean;
7     private InspetorDeAlunosBean inspetorDeAlunosBean;
8     private DiretorBean diretorBean;
9
10    private Responsavel(ProfessorDeHistoriaBean professorDeHistoriaBean,
11                        ProfessorDeLiteraturaBean professorDeLiteraturaBean,
12                        InspetorDeAlunosBean inspetorDeAlunosBean, DiretorBean diretorBean) {
13        this.professorDeHistoriaBean = professorDeHistoriaBean;
14        this.professorDeLiteraturaBean = professorDeLiteraturaBean;
15        this.inspetorDeAlunosBean = inspetorDeAlunosBean;
16        this.diretorBean = diretorBean;
17    }
18
19    //GETTERS OMITIDOS

```

Um ponto forte do pattern *Chain Of Responsibility* é o bom aproveitamento de polimorfismo, permitindo uma grande versatilidade e reuso dos objetos de negócio por outras funcionalidades do sistema, e não apenas a uma necessidade específica. A **Listagem 8** representa exatamente o elemento “*Handler*” no diagrama de classes apresentado na **Figura 1**. Toda a solicitação de atendimento pelos objetos de negócio envolvidos deverá passar por essa Interface, isolando totalmente o acesso à camada de negócio.

Para a necessidade do nosso negócio, a interface possuirá um método que servirá não apenas como acesso a quem delegará a responsabilidade aos objetos requisitados, mas também *automatizará* a nossa cadeia de responsabilidades, como veremos mais à frente.

**Listagem 8.** Cadeia de responsabilidades – Interface.

```

1 package br.com.javamagazine.business;
2
3 import br.com.javamagazine.bean.Responsavel;
4
5 public interface Atendimento {
6
7     Responsavel atende(final Responsavel responsavel);
8 }

```

A primeira parte da estratégia está pronta, mas ainda falta definir as responsabilidades para cada objeto. Conforme abordado, regras de negócio devem sempre ficar isoladas na sua própria camada. Isto facilita a manutenção, evita o alto acoplamento e permite uma alta coerência no código.

A definição das responsabilidades de cada objeto é mostrada nas **Listagens 9 e 10**. Os objetos de negócio 1 e 2 contemplam a regra de negócio necessária para que o responsável pelo atendimento seja identificado.

**Listagem 9.** Cadeia de responsabilidades – Objeto de negócio 1.

```

1 package br.com.javamagazine.business.impl;
2
3 import br.com.javamagazine.bean.Responsavel;
4 import br.com.javamagazine.business.Atendimento;
5 import br.com.javamagazine.enuns.Requisicao;
6
7 public class ProfessorDeHistoriaBO implements Atendimento{
8
9     @Override
10    public Responsavel atende(Responsavel atendimento) {
11        if(atendimento.getProfessorDeHistoriaBean().getCompetencia().equals
12            (Requisicao.DUVIDA_SOBRE_A_GUERRA_DOS_CANUDOS.getDescricao())){
13            System.out.println("O professor de História respondeu a dúvida " +
14                "sobre a guerra dos Canudos.");
15        }
16        return null;
17    }
18 }
```

**Listagem 10.** Cadeia de responsabilidades – Objeto de negócio 2.

```

1 package br.com.javamagazine.business.impl;
2
3 import br.com.javamagazine.bean.Responsavel;
4 import br.com.javamagazine.business.Atendimento;
5 import br.com.javamagazine.enuns.Requisicao;
6
7 public class ProfessorDeLiteraturaBO implements Atendimento{
8
9     @Override
10    public Responsavel atende(Responsavel atendimento) {
11        if(atendimento.getProfessorDeLiteraturaBean().getCompetencia().equals
12            (Requisicao.DUVIDA_SOBRE_BARROCO.getDescricao())){
13            System.out.println("O professor de História respondeu a dúvida " +
14                "sobre o barroco.");
15        }
16        return null;
17    }
18 }
```

Uma vez construídos pelo menos dois dos objetos de negócio necessários, já é possível *refatorar* a classe **CadeiaDeResponsabilidades** apresentada na **Listagem 1**.

Na **Listagem 11** a Interface **Responsavel** agora é utilizada para instanciar o objeto de negócio. Dessa forma, se alguma regra for alterada, pouco fará diferença para essa classe, que só tem a obrigação de “*delegar*” a requisição.

O cliente, representado pela classe **CadeiaDeResponsabilidades**, invoca o primeiro objeto de negócio (no texto seria representado pelo professor de História). Caso esse objeto de negócio não atenda a expectativa, conforme observado na linha 12, o próximo objeto de negócio (representado no texto pelo professor de Literatura) é chamado. Caso esse objeto também não retorne resultado, encerra-se a rotina, como mostrado na linha 13.

**Listagem 11.** Cadeia de responsabilidades – Refatoração.

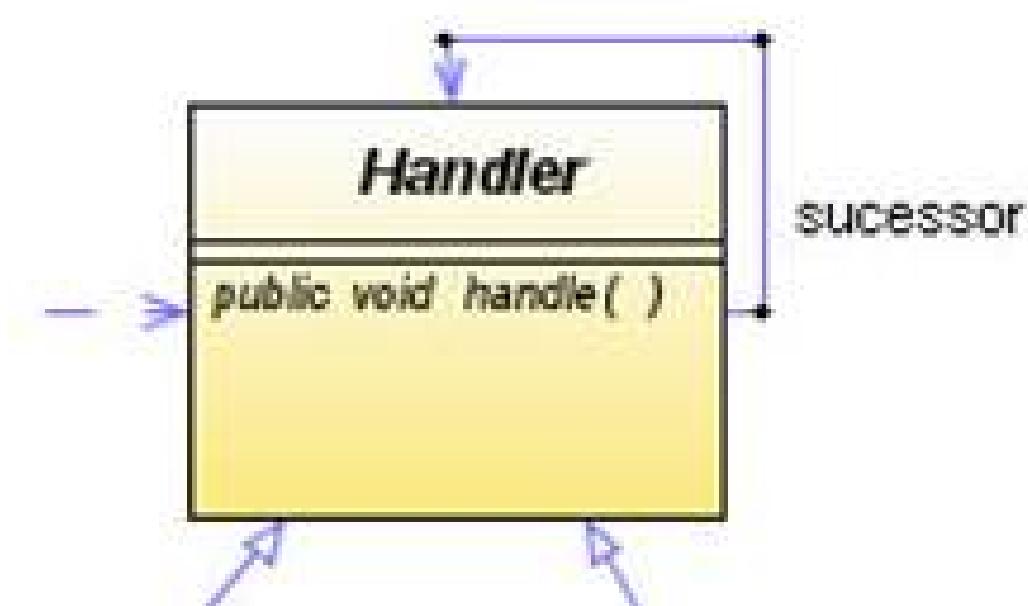
```

1 package br.com.javamagazine.business.impl;
2
3 import br.com.javamagazine.bean.Responsavel;
4 import br.com.javamagazine.business.impl.ProfessorDeHistoriaBO;
5 import br.com.javamagazine.business.impl.ProfessorDeLiteraturaBO;
6
7 public class CadeiaDeResponsabilidades {
8
9     public Responsavel solicitaAtendimento(Responsavel responsavel){
10         Responsavel atendimento = new ProfessorDeHistoriaBO().atende(responsavel);
11         if(atendimento==null)atendimento= new ProfessorDeLiteraturaBO().atende(responsavel);
12         return null;
13     }
14 }
```

Observando novamente o diagrama de classes apresentado na **Figura 1**, pode-se concluir que o refatoramento “quase” atende ao modelo do design pattern, mas ainda não está pronto.

O leitor mais atento vai perceber que o método de fato não contém regra de negócio, mas vai continuar crescendo toda vez que uma nova responsabilidade for adicionada na cadeia. Além disso, existe outro problema visível: o objeto de negócio, quando invocado, caso não atenda ou não possua a responsabilidade esperada, *deveria* invocar o *próximo* objeto de negócio. Essa é a principal motivação ao uso do pattern.

Revendo o trecho do diagrama de classes na **Figura 2** com atenção é possível entender o elemento que falta para continuar a refatoração.



**Figura 2.** Diagrama de Classes – sucessor.

*“O objeto que trata a solicitação deve ser escolhido automaticamente;”.* Como o leitor deve se lembrar, essa é uma das definições apresentadas como característica do pattern no início do artigo. O texto “sucessor” no trecho do diagrama de classes apresentado, representa exatamente a situação em que o objeto de negócio escolhido não tem a responsabilidade esperada e invoca o próximo objeto *automaticamente*.

Um novo método é adicionado na Interface **Atendimento** com o objetivo de invocar o próximo objeto de negócio dentro da cadeia de responsabilidades. O parâmetro recebido por esse método será exatamente o próximo objeto de negócio dentro da cadeia de responsabilidades. Esse método deverá ser invocado sempre que o objeto de negócio atual não atender a expectativa do negócio. Na linha 9 da **Listagem 12** é apresentado o novo método que representará exatamente o elemento que faltava no trecho do diagrama de classes da **Figura 2**.

**Listagem 12.** Cadeia de responsabilidades – Invoca o próximo atendente.

```

1 package br.com.javamagazine.business;
2
3 import br.com.javamagazine.bean.Responsavel;
4
5 public interface Atendimento {
6
7     Responsavel atende(final Responsavel responsavel);
8     //Invoca o próximo objeto de negócio
9     void setSucessor(final Atendimento sucessor);
10 }
```

Analizando as **Listagens 13 e 14**, o novo método (**setSucessor()**) receberá como parâmetro de entrada a interface **Atendimento** (declarada na linha 9 como atributo do objeto de negócio), como pode-se observar na linha 21 de ambas as listagens. Desta forma, o retorno do método, quando a condição não atender ao negócio, será a chamada ao próximo objeto de negócio de forma automática, conforme declarado na linha 18 de ambas as listagens.

**Listagem 13.** Cadeia de responsabilidades – Objeto de negócio 1 invoca o sucessor no atendimento da requisição.

```

1 package br.com.javamagazine.business.impl;
2
3 import br.com.javamagazine.bean.Responsavel;
4 import br.com.javamagazine.business.Atendimento;
5 import br.com.javamagazine.enuns.Requisicao;
6
7 public class ProfessorDeHistoriaBO implements Atendimento{
8
9     private Atendimento sucessor;
10
11    @Override
12    public Responsavel atende(Responsavel atendimento) {
13        if(atendimento.getProfessorDeHistoriaBean().getCompetencia().equals
14            (Requisicao.DUVIDA_SOBRE_A_GUERRA_DOS_CANUDOS.getDescricao())){
15            System.out.println("O professor de História respondeu a dúvida " +
16                "sobre a guerra dos Canudos.");
17        }
18        return sucessor.atende(atendimento);
19    }
}
```

```

20     @Override
21     public void setSucessor(Atendimento sucessor) {
22         this.sucessor = sucessor;
23     }
24 }
```

**Listagem 14.** Cadeia de responsabilidades – Objeto de negócio 2 invoca o sucessor no atendimento da requisição.

```

1 package br.com.javamagazine.business.impl;
2
3 import br.com.javamagazine.bean.Responsavel;
4 import br.com.javamagazine.business.Atendimento;
5 import br.com.javamagazine.enuns.Requisicao;
6
7 public class ProfessorDeLiteraturaBO implements Atendimento{
8
9     private Atendimento sucessor;
10
11    @Override
12    public Responsavel atende(Responsavel atendimento) {
13        if(atendimento.getProfessorDeLiteraturaBean().getCompetencia().equals
14            (Requisicao.DUVIDA_SOBRE_BARROCO.getDescricao())){
15            System.out.println("O professor de História respondeu a dúvida " +
16                "sobre o barroco.");
17        }
18        return sucessor.atende(atendimento);
19    }
20    @Override
21    public void setSucessor(Atendimento sucessor) {
22        this.sucessor = sucessor;
23    }
24 }
```

Nesse momento, podemos criar os dois últimos objetos de negócio previstos dentro da cadeia de responsabilidades, pois o comportamento esperado para cada objeto está entendido e pode ser finalizado. Se o objeto de negócio 1, apresentado na **Listagem 13**, representado no texto pelo professor de história não possuir a competência necessária para adquirir uma cadeira nova para o aluno, ele automaticamente invocará o objeto 2, apresentado na **Listagem 14** e representado no texto pelo professor de literatura. Caso o objeto 2 também não tenha essa competência, ele invocará o objeto 3, apresentado na **Listagem 15**, que, como vimos no texto, é representado pelo inspetor de alunos. Finalmente, caso esse objeto não atenda a expectativa do negócio, invocará o último objeto da cadeia de responsabilidades, representado pelo diretor, apresentado na **Listagem 16**, que, caso atenda a expectativa do negócio, devolverá a resposta esperada pelo cliente (aluno).

**Listagem 15.** Cadeia de responsabilidades – Objeto de negócio 3.

```

1 package br.com.javamagazine.business.impl;
2
3 import br.com.javamagazine.bean.Responsavel;
4 import br.com.javamagazine.business.Atendimento;
5 import br.com.javamagazine.enuns.Requisicao;
6
7 public class InspetorDeAlunosBO implements Atendimento{
8
9     private Atendimento sucessor;
```

```

10
11     @Override
12     public Responsavel atende(Responsavel atendimento) {
13         if(atendimento.getInspetorDeAlunosBean().getCompetencia()
14             .equals(Requisicao.INFORMACAO_DE_LOCALIZACAO_DE_SALA.getDescricao())){
15             System.out.println("O Inspetor de alunos localizou a sala de aula.");
16         }
17         return sucessor.atende(atendimento);
18     }
19     @Override
20     public void setSucessor(Atendimento sucessor) {
21         this.sucessor = sucessor;
22     }
23 }
```

**Listagem 16.** Cadeia de responsabilidades – Objeto de negócio 4.

```

1 package br.com.javamagazine.business.impl;
2
3 import br.com.javamagazine.bean.Responsavel;
4 import br.com.javamagazine.business.Atendimento;
5 import br.com.javamagazine.enums.Requisicao;
6
7 public class DiretorBO implements Atendimento{
8
9     private Atendimento sucessor;
10
11    @Override
12    public Responsavel atende(Responsavel atendimento) {
13        if(atendimento.getDiretorBean().getCompetencia()
14            .equals(Requisicao.AQUISICAO_DE_CADEIRA.getDescricao())){
15            System.out.println("O diretor comprou uma cadeira nova.");
16        }
17        return sucessor.atende(atendimento);
18    }
19    @Override
20    public void setSucessor(Atendimento sucessor) {
21        this.sucessor = sucessor;
22    }
23 }
```

Conforme veremos, a classe **CadeiaDeResponsabilidades** ganhou uma nova refatoração. A interface continua instanciando os objetos de negócio, porém, o retorno do método será sempre o próximo objeto de negócio, conforme observado na linha 19. O elemento “sucessor” é uma implementação da interface **Atendimento** e atende exatamente ao modelo de design pattern apresentado no diagrama de classes da **Figura 1**. Deste modo, caso o primeiro objeto de negócio não atenda ao requisito, o segundo será chamado, e assim sucessivamente, conforme apresentado nas linhas 15,16 e 17.

**Listagem 17.** Cadeia de responsabilidades – Ciclo de atendimento refatorado.

```

1 package br.com.java.magazine.business.impl;
2
3 import br.com.java.magazine.bean.Responsavel;
4 import br.com.java.magazine.business.Atendimento;
5
6 public class CadeiaDeResponsabilidades {
7 }
```

```

8     public Responsavel solicitaAtendimento(Responsavel atendimento){
9
10    Atendimento professorDeHistoria = new ProfessorDeHistoriaBO();
11    Atendimento professorDeLiteratura = new ProfessorDeLiteraturaBO();
12    Atendimento inspetorDeAlunos = new InspetorDeAlunosBO();
13    Atendimento diretor = new DiretorBO();
14
15    professorDeHistoria.setSucessor(professorDeLiteratura);
16    professorDeLiteratura.setSucessor(inspetorDeAlunos);
17    inspetorDeAlunos.setSucessor(diretor);
18
19    return professorDeHistoria.atende(atendimento);
20  }
21 }
```

## Quando o ciclo termina?

Analisando a classe **CadeiaDeResponsabilidades** é possível concluir que existe uma falha na estrutura da implementação apresentada até o momento, e que por sinal é bem comum em um ciclo de desenvolvimento. Sabe-se que existem quatro objetos de negócio que representam os personagens do texto apresentado, sendo que, o objeto de negócio 4, representado no texto pelo diretor, possui a competência necessária para atender a expectativa do negócio e encerrar o ciclo da nossa cadeia de responsabilidades. Mas, e se o objeto de negócio 4 também não fosse capaz de atender a necessidade do cliente? Quando o ciclo da cadeia terminaria? E se nenhum dos nossos objetos de negócio fosse capaz de atender a expectativa? O texto apresentado no início do artigo fala de um aluno de algum cursinho preparatório buscando por uma cadeira, mas, imagine que esse aluno quisesse, por exemplo, um remédio para dor de cabeça? A maioria das escolas possui uma enfermeira, mas dentro da nossa cadeia não temos ninguém com essa responsabilidade. O sistema na forma como está, iria continuar procurando por um responsável infinitamente. Portanto, seria criado um *loop* infinito na nossa *cadeia de responsabilidades*.

Para resolver o problema, um novo objeto de negócio será criado, porém, conforme observado na **Listagem 18** (linha 10) o método que pesquisa pelo próximo responsável retornará **null**. Isso mesmo. Essa classe não deverá fazer absolutamente nada, mas será de grande importância para o encerramento correto do ciclo. Na realidade, ela é o ponto final da cadeia de responsabilidades.

**Listagem 18.** Cadeia de responsabilidades – Objeto de negócio 5.

```

1 package br.com.java.magazine.business.impl;
2
3 import br.com.java.magazine.bean.Responsavel;
4 import br.com.java.magazine.business.Atendimento;
5
6 public class SemAtendimento implements Atendimento {
7
8     @Override
9     public Responsavel atende(Responsavel responsavel) {
10        return null;
11    }
12    @Override
13    public void setSucessor(Atendimento sucessor) {
14    }
15 }
```

Agora a classe **CadeiaDeResponsabilidades** atende por completo as características do pattern *Chain Of Responsibility*. Como é possível verificar na **Listagem 19**, as regras de negócio encontram-se completamente isoladas e a única responsabilidade dessa classe passa a ser única e exclusivamente *delegar* responsabilidades. Se um requisito não for atendido, o ciclo será encerrado, conforme apresentado nas linhas 14 e, posteriormente, 20.

**Listagem 19.** Cadeia de responsabilidades – concluída.

```

1 package br.com.java.magazine.business.impl;
2
3 import br.com.java.magazine.bean.Responsavel;
4 import br.com.java.magazine.business.Atendimento;
5
6 public class CadeiaDeResponsabilidades {
7
8     public Responsavel solicitaAtendimento(Responsavel atendimento){
9
10        Atendimento professorDeHistoria = new ProfessorDeHistoriaBO();
11        Atendimento professorDeLiteratura = new ProfessorDeLiteraturaBO();
12        Atendimento inspetorDeAlunos = new InspetorDeAlunosBO();
13        Atendimento diretor = new DiretorBO();
14        Atendimento semAtendimento = new SemAtendimento();
15
16        professorDeHistoria.setSucessor(professorDeLiteratura);
17        professorDeLiteratura.setSucessor(inspetorDeAlunos);
18        inspetorDeAlunos.setSucessor(diretor);
19        //Encerra o ciclo se o objeto anterior não tiver a competência esperada
20        diretor.setSucessor(semAtendimento);
21
22        return professorDeHistoria.atende(atendimento);
23    }
24 }
```

Resta agora testar a implementação. Para isso, a **Listagem 20** apresenta uma classe de teste que representará, dentro do entendimento do negócio, o apelo do aluno pela cadeira e o atendimento aguardado.

No construtor da classe **Responsavel** será passado como parâmetro exatamente a necessidade do aluno, que é adquirir uma cadeira. Repare que ele (**aluno**) não sabe quem vai atendê-lo. Para ele tanto faz. Deste modo, o mesmo questionamento será feito para todos os personagens envolvidos no texto e agora representados pelos parâmetros de entrada dessa classe. Lembre-se que a classe **Responsavel** é uma composição, e como tal, o construtor da classe possui os quatro objetos da cadeia de responsabilidades como parâmetro de entrada. Cada um desses objetos também possui um parâmetro de entrada, que será justamente o questionamento do aluno citado anteriormente, como é possível observar da linha 19 à linha 24. A solicitação de atendimento é enviada na linha 26. Se um dos quatro objetos atender a expectativa do negócio (aquisição de uma cadeira nova), uma mensagem será exibida na console da sua IDE.

Em nosso exemplo, sabemos que o objeto de negócio 4, representado no texto pelo diretor, possui a competência necessária para atender a necessidade do aluno, e nesse caso a mensagem exibida será “O diretor comprou uma cadeira nova”. Se nenhum dos nossos objetos de negócio tiver a competência esperada, o processo se encerra e nada será exibido no console.

Repare que ao rodar a classe **TestaAtendimento**, o objeto que tratará a requisição e atenderá a expectativa do negócio será identificado de forma automática. Se um objeto não atender a expectativa do negócio, o próximo objeto será imediatamente invocado, porém sem se conhecerem, pois estão totalmente desacoplados um do outro.

**Listagem 20.** Cadeia de responsabilidades – Teste.

```

1 package br.com.java.magazine.test;
2
3 import br.com.java.magazine.bean.DiretorBean;
4 import br.com.java.magazine.bean.InspetorDeAlunosBean;
5 import br.com.java.magazine.bean.ProfessorDeHistoriaBean;
6 import br.com.java.magazine.bean.ProfessorDeLiteraturaBean;
7 import br.com.java.magazine.bean.Requisicao;
8 import br.com.java.magazine.bean.Responsavel;
9 import br.com.java.magazine.business.impl.CadeiaDeResponsabilidades;
10
11 public class TestaAtendimento {
12
13     public static void main(String[] args) {
14
15         CadeiaDeResponsabilidades atendimento = new CadeiaDeResponsabilidades();
16
17         Responsavel responsavel = new Responsavel(
18                 new ProfessorDeHistoriaBean
19                         (Requisicao.AQUISICAO_DE_CADEIRA.getDescricao()),
20                 new ProfessorDeLiteraturaBean
21                         (Requisicao.AQUISICAO_DE_CADEIRA.getDescricao()),
22                 new InspetorDeAlunosBean
23                         (Requisicao.AQUISICAO_DE_CADEIRA.getDescricao()),
24                 new DiretorBean(Requisicao.AQUISICAO_DE_CADEIRA.getDescricao()));
25
26         atendimento.solicitaAtendimento(responsavel);
27     }
28 }
```

O exemplo abordado é simples e cotidiano, com o objetivo de deixar o mais claro possível quando e como utilizar o pattern *Chain Of Responsibility*. Existem outros exemplos que podem ser citados em que a adoção do *Chain Of Responsibility* como solução faria sentido. Casos mais complexos como aprovações de planos de saúde, por exemplo. Neste caso, imagine que um corretor de seguros pretende realizar a venda de uma apólice para um possível asssegurado e reúne as informações necessárias, porém não é capaz de definir a aprovação da aquisição para o solicitante, pois ele não possui essa responsabilidade.

Na realidade, dentro de um sistema corporativo o corretor de seguros seria o cliente. Como não possui a responsabilidade de aprovar a aquisição de uma apólice, ele encaminha a solicitação para que outras alçadas o façam. De fato, ele nem sabe quem aprovará e pouco importa para ele. De acordo com o grau de complexidade das informações do solicitante, a aprovação ainda pode ser “*delegada*” para alçadas maiores. Esta é uma excelente alternativa dentro de um sistema corporativo para atender ao que é conhecido como *tomada de decisão*. Podemos ter vários objetos de negócio, totalmente desacoplados uns dos outros, contendo cada qual a sua devida regra de negócio e a capacidade de tomar uma decisão de acordo com a sua competência (Responsabilidade). Assim, se um desses objetos precisar sofrer qualquer alteração na regra, ou se uma nova regra precisar ser criada, os outros objetos não terão qualquer conhecimento. E se um objeto não for capaz de tratar e tomar uma decisão, o próximo

objeto será chamado imediatamente sem qualquer intervenção interna ou externa. Será automático.

Ademais, o design pattern *Chain of Responsibility* é um recurso que pode auxiliar outros frameworks como o Spring, por exemplo, que trabalha muito bem com injeção de dependência e inversão de controle (*IoC*), mas que precisa de uma implementação de código inteligente e elegante para solucionar problemas como o citado anteriormente. Sendo assim, ao se deparar com situações como as analisadas, não deixe de considerar esse padrão de projeto como uma ótima opção para o seu código.

## Links

### **OODesign.com – Object Oriented Design.**

<http://www.oodesign.com/>

### **Chain of Responsibility.**

[http://sourcemaking.com/design\\_patterns/chain\\_of\\_responsibility](http://sourcemaking.com/design_patterns/chain_of_responsibility)

### **Chain of Responsibility in Java: Before and after.**

[http://sourcemaking.com/design\\_patterns/chain\\_of\\_responsibility/java/1](http://sourcemaking.com/design_patterns/chain_of_responsibility/java/1)

### **Chain of Responsibility Pattern.**

[http://www.tutorialspoint.com/design\\_pattern/chain\\_of\\_responsibility\\_pattern.htm](http://www.tutorialspoint.com/design_pattern/chain_of_responsibility_pattern.htm)

### **Java Design Patterns.**

<http://www.javaworld.com/blog/java-design-patterns>

### **GoF Design Patterns - with examples using Java and UML2.**

<http://www.usp.br/thienne/coo/material/GoFDesignPatterns.pdf>



Arthur Gomes

É formado em Análise de Sistemas pela UNIESP, trabalha com desenvolvimento de sistemas faz aproximadamente dez anos, é especialista em plataforma Java, certificado em várias soluções IBM e Oracle como o InfoSphere DataStage 8.5, W [...]