

# CENTRO UNIVERSITÁRIO FACENS

## CURSOS TECNOLÓGICOS – AS014TGN1



# PROGRAMAÇÃO ORIENTADA A OBJETOS

# INTRODUÇÃO

Na POO um **método**, responsável por **realizar uma** tarefa, pode receber **parâmetros** que serão os dados utilizados nessa tarefa. Há situações onde a **resolução** do método pode ser de **uma forma** ou **de outra, dependendo dos dados** informados.

Por exemplo, imagine uma tabela de produtos e um método inserir que recebe 3 parâmetros: **nome**, **preço** e **data de fabricação**. algo como:

```
public void inserir(String nome, double preco, Date data_fab) {  
    // ...  
}
```

Considere ainda que desejamos omitir a data de fabricação para que esta seja inserida como a data do dia. Teremos então um método com 2 parâmetros: **nome** e **preço**. algo como:

```
public void inserir(String nome, double preco) {  
    // ...  
}
```

# CONTEXTO

```
public class ExemploOverloading {  
    public void inserir(String nome, double preco, Date data_fab) {  
        // ...  
    }  
  
    public void inserir(String nome, double preco) {  
        // ...  
    }  
}
```

É possível ter dois métodos com o mesmo nome na mesma classe?

Sim, desde que tenham parâmetros diferentes.

# SOBRECARGA

Sobrecarregar (ou *overload*) é um conceito em Java que permite que várias funções ou métodos tenham o mesmo nome, mas com diferentes lista de parâmetros. A sobrecarga de métodos é um recurso importante que ajuda a criar métodos com o mesmo nome para realizar operações semelhantes, mas com diferentes tipos e quantidades de parâmetros.

```
public class ExemploOverloading {  
    public void inserir(String nome, double preco, Date data_fab) {  
        // ...  
    }  
  
    public void inserir(String nome, double preco) {  
        // ...  
    }  
}
```

As assinaturas são:

`inserir(String, double, Date)`

`inserir(String, double)`

# CONTEXTO

**Overloading** pode se dar de 2 formas: Pela **quantidade** de parâmetros e pelo **tipo** dos parâmetros.

```
public class Matematica {  
    public int soma(int a, int b) {  
        return a + b;  
    }  
    public int soma(int a, int b, int c) {  
        return a + b + c;  
    }  
    public int soma(int[] lista) {  
        int result = 0;  
  
        for(int n : lista)  
            result += n;  
  
        return result;  
    }  
}
```

The diagram illustrates method overloading in Java. It shows a class `Matematica` with three `soma` methods. Red curly braces group the parameters of each method: `(int a, int b)` for the first, `(int a, int b, int c)` for the second, and `(int[] lista)` for the third. Blue callout boxes with arrows point to these parameter groups, providing descriptions: "2 parâmetros do tipo int" for the first method, "3 parâmetros do tipo int" for the second, and "1 parâmetro do tipo array" for the third.

# CONTEXTO

Agora devemos instanciar a classe e averiguar os resultados:

```
Matematica mat = new Matematica();
```

```
int[] array = {1,2,3,4,5,6,7,8,9};
```

```
int x = mat.soma(1, 2);
```

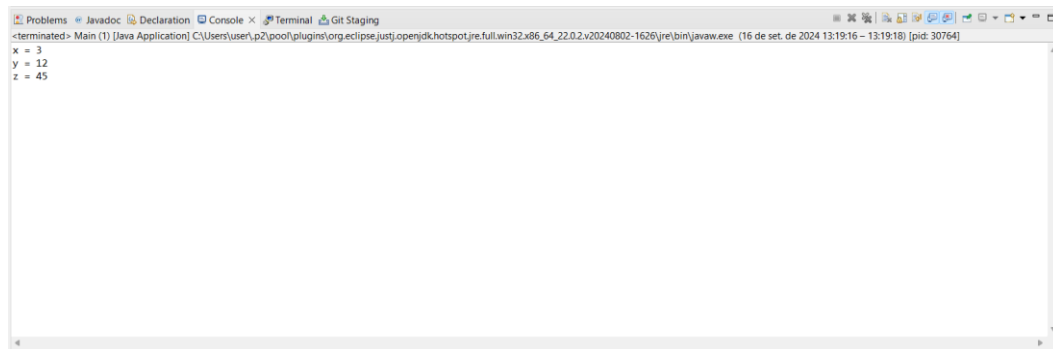
```
int y = mat.soma(3, 4, 5);
```

```
int z = mat.soma(array);
```

```
System.out.println("x = " + x);
```

```
System.out.println("y = " + y);
```

```
System.out.println("z = " + z);
```



The screenshot shows a terminal window from an IDE. The title bar includes tabs for 'Problems', 'Javadoc', 'Declaration', 'Console', 'Terminal', and 'Git Staging'. The terminal content shows the output of the Java program: 'x = 3', 'y = 12', and 'z = 45'. The status bar at the bottom indicates the application is terminated.

```
<terminated> Main (1) [Java Application] C:\Users\user\p2\pool\plugins\org.eclipse.justi.openjdk hotspot\jre.full.win32.x86_64_22.0.2.v20240802-1626\jre\bin\javaw.exe (16 de set. de 2024 13:19:16 - 13:19:18) [pid: 30764]
```

```
x = 3  
y = 12  
z = 45
```

# CONTEXTO

Vamos executar um novo exemplo, para isso crie um novo projeto:

```
public class OperacoesComArrays {  
    public void printArray(int[] array) {  
        for(int i=0; i<array.length; i++ ) {  
            System.out.print(array[i] + " ");  
        }  
        System.out.println();  
    }  
  
    public void printArray(int[] array, int qtd) {  
        for(int i=0; i<qtd; i++ ) {  
            System.out.print(array[i] + " ");  
        }  
        System.out.println();  
    }  
  
    public void printArray(int[] array, int inicio, int fim) {  
        for(int i=inicio; i<fim; i++ ) {  
            System.out.print(array[i] + " ");  
        }  
        System.out.println();  
    }  
}
```

# CONTEXTO

Após a criação da classe OperacoesComArrays precisamos instanciar a classe no projeto:

```
package main;

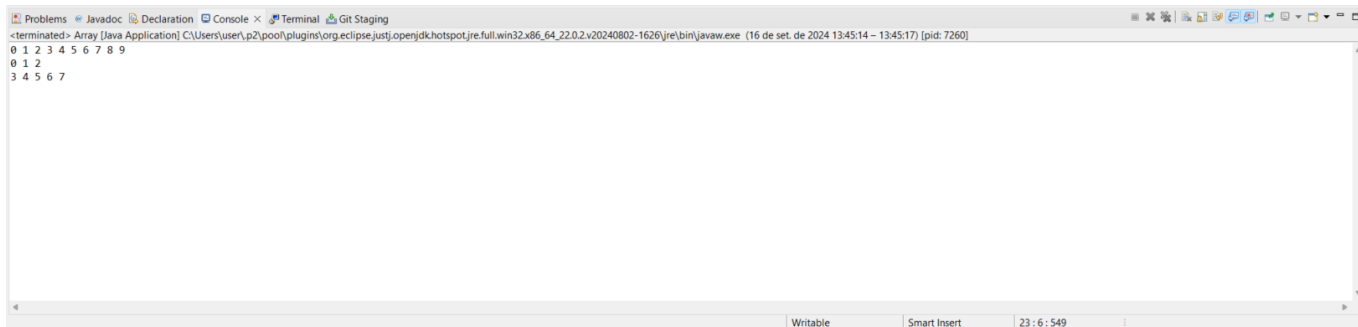
public class Main {

    public static void main(String[] args) {

        int[] a = {0,1,2,3,4,5,6,7,8,9};

        OperacoesComArrays op = new OperacoesComArrays();

        op.printArray(a);
        op.printArray(a, 3);
        op.printArray(a, 3, 8);
    }
}
```



```
<terminated> Array [Java Application] C:\Users\user\p2\pool\plugins\org.eclipse.justf.openjdk.hotspot.jre.full.win32.x86_64.22.0.2.v20240802-1626\jre\bin\javaw.exe (16 de set. de 2024 13:45:14 - 13:45:17) [pid: 7260]
0 1 2 3 4 5 6 7 8 9
0 1 2
3 4 5 6 7
```



# SOBRECARGA DE CONSTRUTORES

Como todo método, os **construtores** podem ser **sobrescritos**. Vamos ver no exemplo a seguir:

```
public class Pessoa {  
    String nome;  
    double altura;  
    String email;  
  
    public Pessoa() {  
    }  
  
    public Pessoa(String nome, double altura, String email) {  
        this.nome = nome;  
        this.altura = altura;  
        this.email = email;  
    }  
  
    public Pessoa(String nome) {  
        this.nome = nome;  
        this.altura = 0;  
        this.email = emailPadrao();  
    }  
  
    private String emailPadrao() {  
        String[] nomes = this.nome.split("");  
        return nomes[0]+"."+nomes[nomes.length-1]+ "@mail.com";  
    }  
}
```

# SOBRECARGA DE CONSTRUTORES

Como todo método, os **construtores** podem ser **sobrescritos**. Vamos ver no exemplo a seguir:

```
Pessoa p1 = new Pessoa();  
p1.setNome("Charles Darwin");  
p1.setAltura(1.70);  
p1.setEmail("dotheevolution@mail.com");  
  
Pessoa p2 = new Pessoa("Alan Turing", 1.6);  
  
System.out.println(p1);  
System.out.println(p2);
```



```
<terminated> Main (3) [Java Application] /usr/lib/jvm/java-17-openjdk-amd64/bin/java (Aug 31, 2022, 4:09:39 PM - 4:09:39 PM)  
Nome: Charles Darwin  
Altura: 1.7  
E-mail: dotheevolution@mail.com  
  
Nome: Alan Turing  
Altura: 1.6  
E-mail: alan.turing@mail.com
```

# RESUMO

A sobrecarga é um conceito fundamental em programação orientada a objetos e ajuda a criar um código mais flexível e reutilizável.

## Regras de Sobrecarga

**1.Nome do Método:** O nome do método deve ser o mesmo para todos os métodos sobrecarregados.

**2.Lista de Parâmetros:** Cada método sobrecarregado deve ter uma lista de parâmetros diferente. Isso pode incluir:

1. Número de parâmetros diferentes
2. Tipos de parâmetros diferentes
3. Ordem dos parâmetros diferentes

**3.Tipo de Retorno:** O tipo de retorno não é suficiente para diferenciar métodos sobrecarregados. Portanto, dois métodos com o mesmo nome e lista de parâmetros, mas tipos de retorno diferentes, não são considerados sobrecarregados.

# RESUMO

## Por Que Usar Sobrecarga?

**1.Legibilidade:** Métodos com o mesmo nome que realizam operações similares são mais fáceis de entender e usar.

**2.Convenção:** Permite que você use o mesmo nome de método para operações que têm propósitos semelhantes, mas que operam em diferentes tipos de dados ou quantidades de dados.

# RESUMO

The screenshot displays the Eclipse IDE interface. The Package Explorer on the left shows a project named 'Refatoracao\_Sobrecarga' with a source folder 'src' containing the file 'SobrecargaExemplo.java'. The main editor window shows the code for 'SobrecargaExemplo.java'.

```
1 |
2 public class SobrecargaExemplo {
3
4     // Método que soma dois inteiros
5     public int soma(int a, int b) {
6         return a + b;
7     }
8
9     // Método que soma três inteiros
10    public int soma(int a, int b, int c) {
11        return a + b + c;
12    }
13
14    // Método que soma dois números de ponto flutuante
15    public double soma(double a, double b) {
16        return a + b;
17    }
18
19    public static void main(String[] args) {
20        SobrecargaExemplo exemplo = new SobrecargaExemplo();
21
22        // Chamadas para métodos sobrecarregados
23        System.out.println(exemplo.soma(10, 20)); // Chama soma(int a, int b)
24        System.out.println(exemplo.soma(10, 20, 30)); // Chama soma(int a, int b, int c)
25        System.out.println(exemplo.soma(10.5, 20.5)); // Chama soma(double a, double b)
26    }
27 }
28
```

The bottom panel shows the console output:

```
<terminated> SobrecargaExemplo [Java Application] C:\Users\user\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.22.0.2.v20240802-1626\jre\bin\javaw.exe (16 de set. de 2024 14:04:43 - 14:04:44) [pid: 13000]
30
60
31.0
```

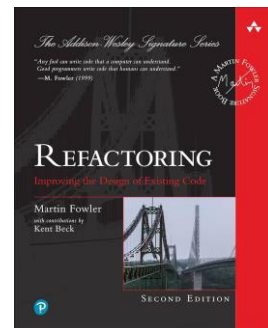
The status bar at the bottom indicates 'Writable', 'Smart Insert', and '1 : 1 : 0'.

# REFATORAÇÃO

Refatoração de código é o processo de modificar a estrutura interna do código-fonte de um software, sem alterar seu comportamento externo, com o objetivo de melhorar sua legibilidade, organização e manutenção. A refatoração visa tornar o código mais limpo, eficiente e fácil de entender, facilitando a adição de novas funcionalidades e a correção de bugs.

## Origem da Refatoração

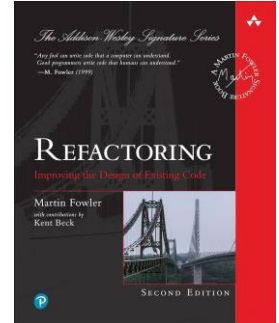
A prática de refatoração tem suas raízes na programação orientada a objetos e na metodologia de desenvolvimento ágil. A palavra "refatoração" foi popularizada pelo livro *"Refactoring: Improving the Design of Existing Code"* de Martin Fowler, publicado em 1999.



# REFATORAÇÃO

## Objetivos da Refatoração

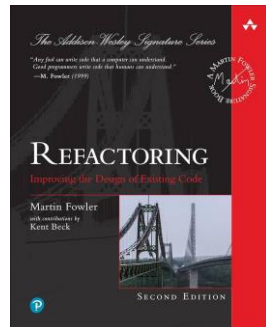
- 1. Melhorar a Legibilidade:** Tornar o código mais fácil de ler e entender por outros desenvolvedores ou mesmo pelo próprio desenvolvedor no futuro.
- 2. Facilitar a Manutenção:** Reduzir a complexidade e a duplicação de código para facilitar a manutenção e evolução do software.
- 3. Aumentar a Reusabilidade:** Criar código modular que pode ser reutilizado em diferentes partes do projeto ou em projetos diferentes.
- 4. Reduzir a Complexidade:** Simplificar a lógica e a estrutura do código para torná-lo mais fácil de testar e modificar.



# REFATORAÇÃO

## Exemplos de Refatoração

- **Renomear Variáveis e Métodos:** Dar nomes mais descritivos e intuitivos a variáveis e métodos para tornar o código mais compreensível.
- **Extraír Métodos:** Dividir métodos grandes em métodos menores e mais específicos para melhorar a clareza.
- **Eliminar Código Duplicado:** Consolidar trechos de código duplicado em métodos ou classes reutilizáveis.
- **Reorganizar Classes e Pacotes:** Organizar classes e pacotes de maneira que reflitam melhor a estrutura e a funcionalidade do sistema.





# REFATORAÇÃO

The screenshot displays the Eclipse IDE interface. The Package Explorer on the left shows a project named 'Refatoracao\_Sobrecarga' with a source folder 'src' containing the file 'SobrecargaExemplo.java'. The main editor window shows the code for 'SobrecargaExemplo.java'.

```
1 |
2 public class SobrecargaExemplo {
3
4     // Método que soma dois inteiros
5     public int soma(int a, int b) {
6         return a + b;
7     }
8
9     // Método que soma três inteiros
10    public int soma(int a, int b, int c) {
11        return a + b + c;
12    }
13
14    // Método que soma dois números de ponto flutuante
15    public double soma(double a, double b) {
16        return a + b;
17    }
18
19    public static void main(String[] args) {
20        SobrecargaExemplo exemplo = new SobrecargaExemplo();
21
22        // Chamadas para métodos sobrecarregados
23        System.out.println(exemplo.soma(10, 20)); // Chama soma(int a, int b)
24        System.out.println(exemplo.soma(10, 20, 30)); // Chama soma(int a, int b, int c)
25        System.out.println(exemplo.soma(10.5, 20.5)); // Chama soma(double a, double b)
26    }
27 }
28
```

The bottom panel shows the console output:

```
<terminated> SobrecargaExemplo [Java Application] C:\Users\user\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.22.0.2.v20240802-1626\jre\bin\javaw.exe (16 de set. de 2024 14:04:43 - 14:04:44) [pid: 13000]
30
60
31.0
```

The status bar at the bottom indicates 'Writable', 'Smart Insert', and '1 : 1 : 0'.

# REFATORAÇÃO

The screenshot displays the Eclipse IDE interface. The Package Explorer on the left shows the project structure: Main, Main\_Array, Refatoracao\_Sobrecarga, JRE System Library [jre], src, and a default package containing SobrecargaExemplo.java. The main editor window shows the source code of SobrecargaExemplo.java.

```
1 public class SobrecargaExemplo {
2
3     // Método que soma dois inteiros
4     public int soma(int a, int b) {
5         return a + b;
6     }
7
8     // Método que soma três inteiros
9     public int soma(int a, int b, int c) {
10        return a + b + c;
11    }
12
13    // Método que soma dois números de ponto flutuante
14    public double soma(double a, double b) {
15        return a + b;
16    }
17
18    // Método principal
19    public static void main(String[] args) {
20        SobrecargaExemplo exemplo = new SobrecargaExemplo();
21
22        // Chamadas para métodos sobrecarregados
23        System.out.println(formatarResultado(exemplo.soma(10, 20))); // Chama soma(int a, int b)
24        System.out.println(formatarResultado(exemplo.soma(10, 20, 30))); // Chama soma(int a, int b, int c)
25        System.out.println(formatarResultado(exemplo.soma(10.5, 20.5))); // Chama soma(double a, double b)
26    }
27
28    // Método para formatar a saída do resultado
29    private static String formatarResultado(double resultado) {
30        return "Resultado: " + resultado;
31    }
32 }
33
34
```

The console output at the bottom shows the results of the program execution:

```
<terminated> SobrecargaExemplo [Java Application] C:\Users\user\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.22.0.2.v20240802-1626\jre\bin\javaw.exe (16 de set. de 2024 14:14:46 – 14:14:48) [pid: 27564]
Resultado: 30.0
Resultado: 60.0
Resultado: 31.0
```

The status bar at the bottom indicates the file is Writable, in Smart Insert mode, and the cursor is at line 33, column 1 of 1082 characters.

# REFATORAÇÃO

- Método `formatarResultado`: Adicionado um método separado para formatar a saída dos resultados. Isso facilita a modificação da forma como os resultados são exibidos sem alterar a lógica de cálculo.
- Separação de Lógica e Apresentação: Ao mover a lógica de impressão para um método separado, você torna o código principal (main) mais limpo e focado na lógica de negócios.
- Comentários: Incluídos para descrever o propósito dos métodos, tornando o código mais fácil de entender para outros desenvolvedores.

Observe o próximo exemplo de uma calculadora de soma e subtração.

# REFATORAÇÃO

The screenshot shows an IDE with the following components:

- Package Explorer:** Shows a project named 'Calculadora' with a source folder 'src' containing a file 'Calculadora.java'.
- Editor:** Displays the code for 'Calculadora.java'. The code includes a class 'Calculadora' with a 'main' method and two static methods: 'soma' and 'subtracao'. The 'main' method initializes variables 'a' and 'b', calculates their sum and difference, and prints the results. Comments in the code indicate refactoring steps: 'renomear para soma', 'renomear para subtracao', and 'renomear para soma'.
- Console:** Shows the output of the program: 'Soma: 15', 'Subtração: 5', 'Nova Soma: 35', and 'Nova Subtração: 5'.

```
1 public class Calculadora {
2
3     public static void main(String[] args) {
4         int a = 10;
5         int b = 5;
6         int resultadoSoma;
7         int resultadoSubtracao;
8
9         resultadoSoma = soma(a, b);
10        System.out.println("Soma: " + resultadoSoma);
11
12        resultadoSubtracao = subtracao(a, b);
13        System.out.println("Subtração: " + resultadoSubtracao);
14
15        resultadoSoma = soma(20, 15);
16        System.out.println("Nova Soma: " + resultadoSoma);
17
18        resultadoSubtracao = subtracao(20, 15);
19        System.out.println("Nova Subtração: " + resultadoSubtracao);
20    }
21
22    public static int soma(int x, int y) {
23        return x + y;
24    }
25
26    public static int subtracao(int x, int y) {
27        return x - y;
28    }
29 }
30
```

Problems | Javadoc | Declaration | Console | Terminal | Git Staging

<terminated> Calculadora [Java Application] C:\Users\user\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86\_64\_22.0.2.v20240802-1626\jre\bin\javaw.exe (16 de set. de 2024 14:27:09 – 14:27:11) [pid: 28692]

Soma: 15  
Subtração: 5  
Nova Soma: 35  
Nova Subtração: 5

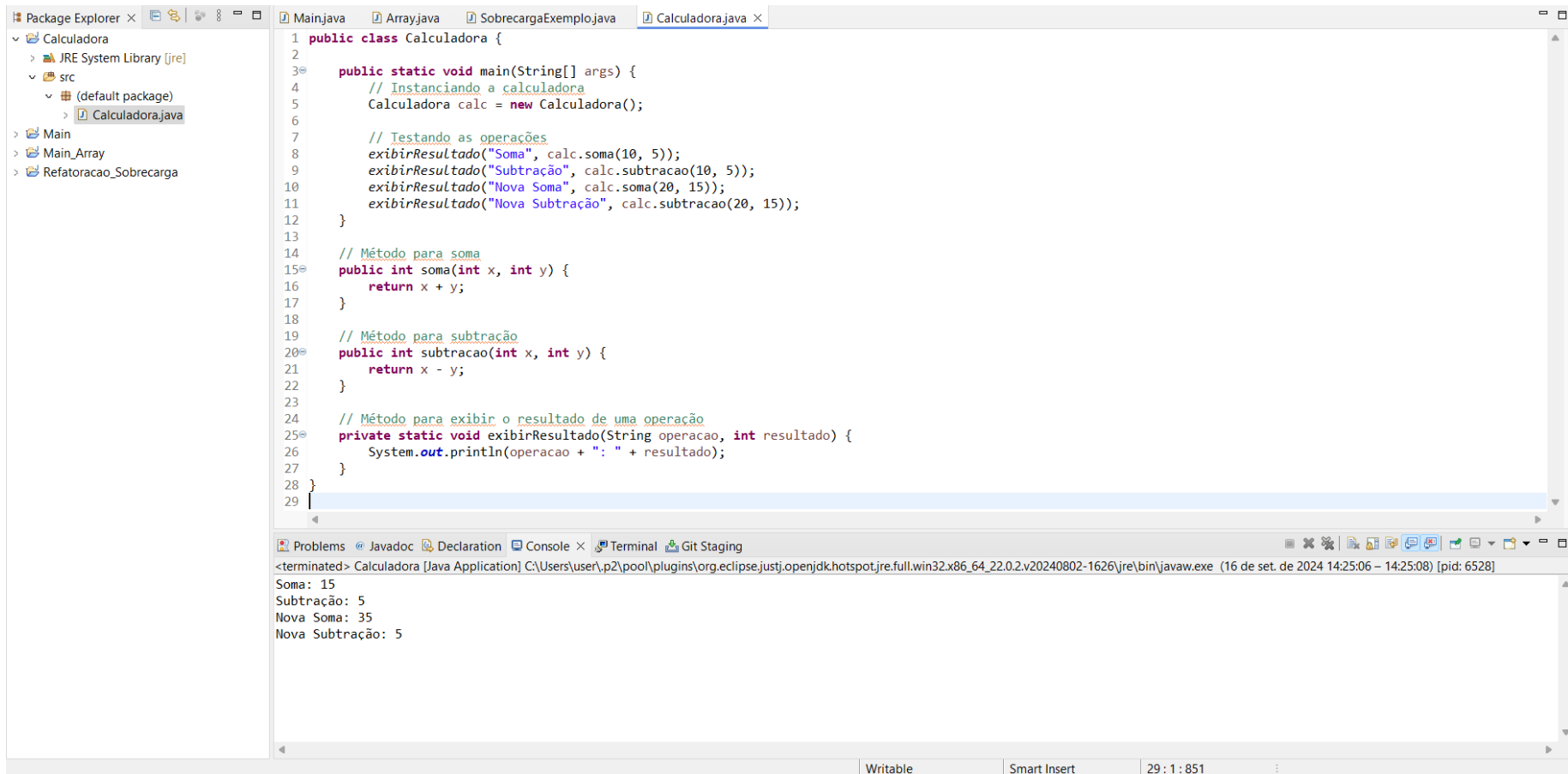
Writable | Smart Insert | 30 : 1 : 846

# REFATORAÇÃO

## Pontos a Serem Refatorados

- 1. Duplicação de Código:** O código para realizar soma e subtração é repetido com diferentes valores.
- 2. Métodos Específicos:** A lógica de soma e subtração é simples, mas poderia ser centralizada para facilitar a expansão futura.
- 3. Código de Impressão:** A lógica de impressão dos resultados pode ser melhorada para evitar repetição e tornar o código mais limpo.

# REFATORAÇÃO



The screenshot displays the Eclipse IDE interface. On the left, the Package Explorer shows a project named 'Calculadora' with a source folder 'src' containing the file 'Calculadora.java'. The main editor window shows the code for 'Calculadora.java'.

```
1 public class Calculadora {
2
3     public static void main(String[] args) {
4         // Instanciando a calculadora
5         Calculadora calc = new Calculadora();
6
7         // Testando as operações
8         exibirResultado("Soma", calc.soma(10, 5));
9         exibirResultado("Subtração", calc.subtracao(10, 5));
10        exibirResultado("Nova Soma", calc.soma(20, 15));
11        exibirResultado("Nova Subtração", calc.subtracao(20, 15));
12    }
13
14    // Método para soma
15    public int soma(int x, int y) {
16        return x + y;
17    }
18
19    // Método para subtração
20    public int subtracao(int x, int y) {
21        return x - y;
22    }
23
24    // Método para exibir o resultado de uma operação
25    private static void exibirResultado(String operacao, int resultado) {
26        System.out.println(operacao + ": " + resultado);
27    }
28 }
29
```

The bottom of the IDE shows the Console window with the following output:

```
<terminated> Calculadora [Java Application] C:\Users\user\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_22.0.2.v20240802-1626\jre\bin\javaw.exe (16 de set. de 2024 14:25:06 - 14:25:08) [pid: 6528]
Soma: 15
Subtração: 5
Nova Soma: 35
Nova Subtração: 5
```

The status bar at the bottom indicates 'Writable', 'Smart Insert', and '29 : 1 : 851'.

# REFATORAÇÃO

**Método `exibirResultado`:** Adicionado um método auxiliar para exibir o resultado, evitando a repetição do código de impressão.

**Instanciação da `Calculadora`:** O código agora usa uma instância da classe `Calculadora`, o que facilita a extensão do código se mais funcionalidades forem adicionadas no futuro.

**Modularidade:** A lógica de soma e subtração é mantida separada da lógica de exibição dos resultados, melhorando a clareza e a manutenção do código.

**Comentários:** Incluídos para descrever o propósito dos métodos, tornando o código mais fácil de entender para outros desenvolvedores.

Mas podemos melhorar ainda mais o projeto, observe como:

# REFATORAÇÃO

The screenshot shows the Eclipse IDE interface. The Package Explorer on the left lists the project 'Calculadora' with its source files. The main editor displays the code for 'CalculadoraApp.java'.

```
1 public class CalculadoraApp {
2
3     public static void main(String[] args) {
4         // Instanciando a calculadora
5         Calculadora calc = new Calculadora();
6
7         // Testando as operações
8         exibirResultado("Soma", calc.soma(10, 5));
9         exibirResultado("Subtração", calc.subtracao(10, 5));
10        exibirResultado("Nova Soma", calc.soma(20, 15));
11        exibirResultado("Nova Subtração", calc.subtracao(20, 15));
12    }
13
14    // Método para exibir o resultado de uma operação
15    private static void exibirResultado(String operacao, int resultado) {
16        System.out.println(operacao + ": " + resultado);
17    }
18 }
19
20
```

The console output shows the results of the calculations:

```
<terminated> CalculadoraApp [Java Application] C:\Users\user\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_22.0.2.v20240802-1626\jre\bin\javaw.exe (16 de set. de 2024 14:36:05 - 14:36:05) [pid: 9816]
Soma: 15
Subtração: 5
Nova Soma: 35
Nova Subtração: 5
```



# REFATORAÇÃO

The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer displays the project structure for 'Calculadora', including 'JRE System Library [jre]', 'src', and 'CalculadoraApp.java'. The main editor window shows the 'Calculadora.java' file with the following code:

```
1 public class Calculadora {
2
3     // Método para soma
4     public int soma(int x, int y) {
5         return x + y;
6     }
7
8     // Método para subtração
9     public int subtracao(int x, int y) {
10        return x - y;
11    }
12 }
13
14
```

The bottom console shows the output of the application:

```
<terminated> CalculadoraApp [Java Application] C:\Users\user\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_22.0.2.v20240802-1626\jre\bin\javaw.exe (16 de set. de 2024 14:36:05 – 14:36:05) [pid: 9816]
Soma: 15
Subtração: 5
Nova Soma: 35
Nova Subtração: 5
```

# REFATORAÇÃO

## Explicação da Refatoração

### Classe Calculadora:

- Contém apenas a lógica para as operações matemáticas.
- É uma classe independente que pode ser reutilizada em diferentes contextos sem depender de como os resultados são exibidos.

### Classe CalculadoraApp:

- Contém o método main, que é o ponto de entrada do programa.
- Cria uma instância da classe Calculadora e usa seus métodos.
- Contém a lógica de exibição dos resultados, centralizada no método exibirResultado.

# REFATORAÇÃO

## Benefícios da Refatoração

- **Separação de Preocupações:** A lógica de cálculos e a lógica de exibição estão separadas. Isso facilita a manutenção e a atualização de cada parte independentemente.
- **Modularidade:** A classe Calculadora pode ser reutilizada em outros projetos ou contextos sem depender da implementação de exibição.
- **Escalabilidade:** Se precisar adicionar novas operações matemáticas ou mudar a forma como os resultados são exibidos, você pode fazer isso de maneira isolada, sem afetar a outra parte do código.

MUITO OBRIGADO!!!



[daniel.ohata@facens.br](mailto:daniel.ohata@facens.br)