1 Sobrecarga de métodos

É possível que uma determinada classe possua métodos de nome igual. O que torna isso possível é um mecanismo conhecido como **sobrecarga de métodos**. Para começar a entendê-lo, iremos implementar uma classe que representa calculadoras.

1.1 (Criando um novo projeto) Crie um novo projeto com um nome conveniente, que te ajude a lembrar depois a que se refere essa solução.

1.2 (Criando a classe para representar calculadoras) Crie uma nova classe. Será preciso escolher um nome para ela. Dado que ela serve para representar calculadoras, um bom nome poderia ser **Calculadora**. Veja seu código na Listagem 1.2.1.

Listagem 1.2.1

```
public class Calculadora {

}
```

1.3 (Escrevendo métodos para as operações aritméticas elementares) Nossa calculadora, inicialmente, será capaz de realizar as quatro operações elementares envolvendo números inteiros. A Listagem 1.3.1 mostra os métodos que as implementam. Cada método recebe dois parâmetros, realiza a operação desejada e, a seguir, devolve o resultado para seu cliente (o método que o chamou).

Listagem 1.3.1

```
public class Calculadora {
	public int soma (int a, int b) {
		//usando uma variável para o resultado
		int resultado;
		resultado = a + b;
		return resultado;
	}
	public int subtracao (int a, int b) {
		//sem usar uma variável para o resultado
		return a - b;
	}
	public int multiplicacao (int a, int b) {
		return a * b;
	}
	public int divisao (int a, int b) {
		//note que aqui ocorre a divisão inteira
		return a / b;
	}
}
```

1.4 (Testando a classe calculadora) Agora vamos criar uma nova classe (por conta do princípio chamado **alta coesão**) para testar a classe Calculadora. Instanciaremos uma calculadora e deixaremos o usuário decidir qual operação deseja realizar. Veja a Listagem 1.4.1.

Listagem 1.4.1

```
import javax.swing.JOptionPane;
public class TesteCalculadora {
	public static void main(String[] args) {
		//construindo uma  calculadora
		Calculadora c = new Calculadora();
		//o que o usuário quer?
		int opcao = Integer.parseInt(JoptionPane.showInputDialog("1-Soma\n2-
Subtracao\n3-Multiplicacao\n4-Divisao"));
		//quais os operandos?
		int operando1 = Integer.parseInt(JOptionPane.showInputDialog("Qual o
primeiro operando?"));
		int operando2 = Integer.parseInt(JOptionPane.showInputDialog("Qual o
segundo operando?"));
		//para guardar o resultado
		int resultado;
		if (opcao == 1) {
			resultado = c.soma(operando1, operando2);
		}
		else if (opcao == 2) {
			resultado = c.subtracao(operando1, operando2);
		}
		else if (opcao == 3) {
			resultado = c.multiplicacao(operando1, operando2);
		}
		else {
			resultado = c.divisao(operando1, operando2);
		}
		//mostrando o resultado
		JOptionPane.showMessageDialog(null, "Resultado: " + resultado);
	}
}
```

1.5 (Sobrecarga para o método de adição, lidando com reais) Agora suponha que desejamos que nossa calculadora seja capaz de somar números reais também. Poderíamos criar um método com nome diferente para isso, mas do ponto de visto do cliente (o código que usa o método) é muito mais cômodo usar o mesmo método. Pelo menos é que vai parecer para ele, quando fizermos uma sobrecarga para o método de adição, como na Listagem 1.5.1.

Listagem 1.5.1

```
//uma sobrecarga para o método soma
public float soma (float a, float b) {
	return a + b;
}
```

Perceba que o novo método também se chama soma. Porém ele tem **lista de parâmetros diferente**. Esse exemplo ilustra o mecanismo conhecido como **sobrecarga de métodos**. Ele permite que métodos de nome igual e mesmo escopo (declarados na mesma classe, por exemplo) coexistam. Isso é possível devido à lista de parâmetros ser diferente.

1.6 (Sobrecarga para o método de adição, lidando com strings) Agora vamos fazer uma sobrecarga para o método soma que recebe duas strings, as converte para inteiro, faz a soma e devolve o resultado. Veja a Listagem 1.6.1.

Listagem 1.6.1

```
public int soma (String a, String b) {
                    int n1 = Integer.parseInt(a);
                    int n2 = Integer.parseInt(b);
                    return n1 + n2;
}
```

Note que essa versão é diferente das demais, pois sua lista de parâmetros é diferente.

1.7 (Sobrecarga para o método de adição, lidando com três inteiros) A próxima sobrecarga permitirá a soma de três números inteiros. Veja a Listagem 1.7.1.

Listagem 1.7.1

```
public int soma (int a, int b, int c) {
                    return a + b + c;
}
```

1.8 (Sobrecarga para o método de adição, lidando com string e int) Digamos que queremos somar um inteiro com uma string (convertendo-a primeiro para int). Se desejarmos, podemos escrever duas sobrecargas, invertendo a ordem dos parâmetros. Veja a Listagem 1.8.1.

Listagem 1.8.1

```
public int soma (String a, int b) {
                    return Integer.parseInt(a) + b;
}
public int soma (int a, String b) {
          return a + Integer.parseInt(b);
}
```

1.9 (Testando as novas sobrecargas do método de adição) A classe da Listagem 1.9.1 mostra como usar as novas sobrecargas. Pelo princípio da alta coesão, crie uma nova classe para esse novo teste.

Listagem 1.9.1

```
public class TesteSobrecargaSoma {
	public static void main(String[] args) {
		Calculadora c = new Calculadora();
		//operandos
		int x = 1, y = 2, z = 3;
		String s1 = "50", s2 = "60";
		float f1 = 56.7f, f2 = 0.3f;

		//somando dois números reais
		System.out.println(c.soma(f1, f2));

		//somando dois inteiros
		System.out.println(c.soma(x, y));

		//somando inteiro e string
		System.out.println(c.soma(x, s1));

		//somando string e inteiro
		System.out.println(c.soma(s1, x));

		//somando string e string
		System.out.println(c.soma(s1, s2));

		//somando três inteiros
		System.out.println(c.soma(x, y, z));
	}
}
```

1.10 (Entendendo os critérios para que a sobrecarga de métodos funcione) Quando utilizamos a sobrecarga de métodos, uma pergunta é muito natural: como o compilador faz para diferenciar um método do outro, dado que todos têm o mesmo nome e mesmo escopo. Por exemplo, quando chamamos soma passando dois inteiros, como o compilador sabe que não queremos a versão que recebe duas strings? Na verdade, essa pergunta já contém a resposta. O compilador diferencia os métodos uns dos outros por meio de sua lista de parâmetros. O método a ser chamado será aquele cuja assinatura está de acordo com a lista de argumentos passados no momento da chamada ao método. Porém, é necessário entender as regras precisamente. Quais critérios o compilador utiliza para dizer que uma lista de parâmetros é diferente da outra. Eles são três:

- **Quantidade de parâmetros**: Dois métodos que recebem quantidade diferente de parâmetros são diferentes. Por esse critério, os métodos da Tabela 1.10.1 são diferentes. A segunda linha mostra como o cliente os chama.

Tabela 1.10.1

public int soma (**int** a, **int** b) {
return a + b;
}
	public int soma (**int** a, **int** b, **int** c) {
	return a + b + c;
	}

c.soma (1, 2);	c.soma (1, 2, 3);

- **Tipo dos parâmetros**: Se dois métodos possuem a mesma quantidade de parâmetros de tipos diferentes, para o compilador eles são diferentes. Por esse critério, os métodos da Tabela 1.10.2 são diferentes. A segunda linha mostra como o cliente os chama.

Tabela 1.10.2

public int soma (String a, String b) {
int n1 = Integer.*parseInt*(a);
int n2 = Integer.*parseInt*(b);
return n1 + n2;
}
	public float soma (**float** a, **float** b) {
	return a + b;
	}

c.soma (“1”, “2”);	c.soma (5f, 2f);

- **Ordem dos parâmetros:** Se dois métodos possuem a mesma quantidade de parâmetros e eles têm o mesmo tipo, porém a ordem de aparição na lista é diferente, para o compilador eles são diferentes. Por esse critério, os métodos da Tabela 1.10.3 são diferentes. A segunda linha mostra como o cliente os chama.

Tabela 1.10.3

public int soma (String a, **int** b) {
return Integer.*parseInt*(a) + b;
}
	public int soma (**int** a, String b) {
	return a + Integer.*parseInt*(b);
	}

c.soma (“1”, 2);	c.soma (1, “2”);

1.11 (Tentando diferenciar métodos pelo nome dos seus parâmetros) Considere os métodos da Tabela 1.11.1. Embora seus parâmetros tenham nomes diferentes, para o compilador eles são iguais e, portanto, **não podem coexistir**. Isso ocorre pois o cliente, ao especificar os argumentos no momento da chamada ao método, não tem meios de informar o nome do parâmetro a que cada argumento deve ser atribuído. A segunda linha ilustra isso.

Tabela 1.11.1

public int soma (**int** a, String b) {
return a + Integer.*parseInt*(b);
}
	public int soma (**int** x, String y) {
	return x + Integer.*parseInt*(y);
	}

c.soma (2, “3”);	c.soma (1, “2”);

Exercícios

1 Escreva uma classe chamada Impressora. Essa classe tem como finalidade imprimir conteúdos diversos na tela. Ela deve possuir diversos métodos chamados **exibir**. Eles poderão coexistir graças ao mecanismo conhecido como sobrecarga de métodos. Escreva versões do método exibir que tenham as seguintes listas de parâmetros.

1.1 um único float.
1.2 dois floats.
1.3 um float e uma String, nessa ordem.
1.4 uma String e um float, nessa ordem.
1.5 três Strings.
1.6 dois ints e uma String, nessa ordem.

Em todo caso, cada método exibir deve exibir todos os valores usando System.out.println, separando cada um por vírgula.

2 Escreva uma classe de teste que chama cada um dos métodos que você escreveu no exercício 1.

3 Escreva uma classe chamada CalculadoraCientifica. Ela deve fazer as operações de potenciação e radiciação com inteiros e reais.

3.1 Escreva um método chamado raiz. Utilizando sobrecarga de métodos, ele terá três versões:

3.1.1 Lista de parâmetros com um inteiro
3.1.2 Lista de parâmetros com um double
3.1.3 Lista de parâmetros com uma String (internamente, ele converte a String para Double com Double.parseDouble antes de fazer a operação).

Dica 1: Use o método Math.sqrt para fazer o cálculo da raiz.
https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html#sqrt-double-

Dica 2: Todos os métodos têm tipo de retorno double.

3.2 Escreva um método chamado potência. Utilizando a sobrecarga de métodos, ele terá três versões:
3.2.1 Lista de parâmetros com dois bytes a e b. Ele calcula e devolve a elevado a b.
3.2.2 Lista de parâmetros com duas Strings s1 e s2. Ele converte ambas para double, calcula e devolve s1 elevado a s2.
3.2.3 Lista de parâmetros com um inteiro a e um double b. Ele calcula e devolve a elevado a b.

Dica 1: Use o método Math.pow para calcular a potência.
https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html#sqrt-pow
Dica 2: Todos os métodos têm tipo de retorno double.

4 Escreva uma classe de teste para testar todos os métodos da sua calculadora científica.

Referências

DEITEL, P. e DEITEL, H. **Java Como Programar**. 8ª Edição. São Paulo, SP: Pearson, 2010.

LOPES, A. e GARCIA, G. **Introdução à Programação – 500 Algoritmos Resolvidos**. 1ª Edição. São Paulo, SP: Elsevier, 2002.