1 Polimorfismo

Polimorfismo significa **múltiplas formas** e é um recurso que aparece na linguagem Java de três formas diferentes.

- A **sobrecarga de métodos** é um tipo de polimorfismo pois, do ponto de vista do código cliente, um mesmo método pode ser chamado de formas diferentes, embora saibamos que, na verdade, trata-se de vários métodos com nome igual. Esse tipo de polimorfismo é, geralmente, chamado de **polimorfismo estático**. Estático no sentido de ser realizado pelo compilador.

- O recurso conhecido como **Generics** do java permite um tipo de polimorfismo, pois ele permite que uma classe (ou método) se comporte de forma diferente de acordo com um ou mais tipos de dados que são especificados em tempo de compilação. Esse tipo de polimorfismo é conhecido como **polimorfismo paramétrico**.

- A **sobrescrita de métodos** é um tipo de polimorfismo em que, em tempo de execução, a JVM decide qual método colocar em execução de acordo com o tipo do objeto utilizado. Objetos diferentes recebem a mesma mensagem (ou seja, um nome de nome igual é chamado sobre eles) e, de acordo com seu tipo, apresentam comportamento diferente. Esse tipo de polimorfismo é conhecido como **polimorfismo dinâmico**. Dinâmico no sentido de ser realizado pela JVM, ou seja, em tempo de execução.

Quando se fala somente de polimorfismo, é comum que se esteja referindo ao polimorfismo dinâmico. Esse é o assunto da aula de hoje.

1.1 (Criando um projeto) Como de costume, começamos criando um projeto e dando a ele um nome de interesse, relacionado ao contexto de estudo.

1.2 (Variáveis polimórficas) O polimorfismo depende da existência de uma hierarquia de classes.

1.2.1 A Listagem 1.2.1.1 mostra a definição de uma hierarquia de classes simples. Como de costume, crie cada classe em um arquivo à parte.

Listagem 1.2.1.1

```
public class Animal {
	public void fazerBarulho () {
		System.out.println("Animal fazendo barulho");
	}
}
public class Gato extends Animal{

}
public class Cachorro extends Animal{

}
```

1.2.2 Como sabemos, Gato e Cachorro herdam o método fazerBarulho. Isso quer dizer que esse método pode ser chamado utilizando-se objetos de ambos os tipos, como mostra a Listagem 1.2.2.1.

Listagem 1.2.2.1

```
public class TesteSemVariaveisPolimorficas {
	public static void main(String[] args) {
		Gato gato = new Gato ();
		Cachorro cachorro = new Cachorro ();
		gato.fazerBarulho();
		cachorro.fazerBarulho();
	}
}
```

1.2.3 Ocorre que ambos Gato e Cachorro passam no teste É-UM Animal, ou seja, herdam de Animal. Isso quer dizer que **variáveis de referência do tipo Animal podem fazer referência a objetos do tipo Gato e do tipo Cachorro.** Por essa razão, dizemos que variáveis do tipo Animal são polimórficas. A ideia é simples. Uma variável do tipo Animal pode fazer referência a objetos de múltiplos tipos, ou seja, de um certo ponto de vista, ela se comporta de formas diferentes, de múltiplas formas. Daí o nome: variável polimórfica. Veja um exemplo na Listagem 1.2.3.1.

Listagem 1.2.3.1

```
public class TesteComPolimorfismo {
	public static void main(String[] args) {
		Animal a1 = new Gato();
		Animal a2 = new Cachorro();
		a1.fazerBarulho();
		a2.fazerBarulho();
	}
}
```

1.3 (Sobrescrita de métodos) Animais diferentes fazem sons diferentes. Apesar disso, ambos Gato e Cachorro se comportam de forma

igual quando o método fazerBarulho é colocado em execução. O que desejamos é personalizar o funcionamento desse método, de acordo com a classe em que ele aparece. Para tal, faremos uso do recurso conhecido como sobrescrita de métodos. Um método herdado pode ser sobrescrito pela classe que o herda, de modo que seu funcionamento seja personalizado.

1.3.1 Veja o exemplo da Listagem 1.3.1.1. Ambas as classes Gato e Cachorro sobrescrevem o método fazerBarulho.

Listagem 1.3.1.1

```
public class Gato extends Animal{
        public void fazerBarulho() {
                System.out.println("miau");
        }
}
public class Cachorro extends Animal{
        public void fazerBarulho() {
                System.out.println("au au");
        }
}
```

1.3.2 Execute novamente o código da Listagem 1.2.3.1. Perceba que **o método a ser chamado depende do tipo do objeto e não do tipo da variável de referência**. Ou seja, em tempo de execução a JVM decide qual método chamar, já que seu tipo (do objeto) é conhecido somente por ela.

1.4 (Um sistema de folha de pagamento sem usar polimorfismo) A fim de ilustrar a utilidade do polimorfismo, vamos implementar um sistema de folha de pagamento. Em uma determinada empresa, há diversos tipos de empregados. E cada um deles tem uma forma diferente de receber seu salário. A Tabela 1.4.1 mostra os tipos existentes de empregados.

Tabela 1.4.1

Tipo	Descrição	Cálculo
1	Empregado assalariado	salário
2	Empregado assalariado e comissionado	salário + percentual sobre o salário
3	Empregado assalariado, comissionado e bonificado	salário + percentual sobre o salário + um valor de bônus
4	Empregado horista	valor da hora * número de horas

1.4.1 A Listagem 1.4.1 mostra uma possível implementação para esse sistema. Há uma única classe Empregado que tem como responsabilidade representar todos os tipos de empregados. Ela tem um método para cálculo do salário que varia de acordo com o tipo do empregado.

Listagem 1.4.1

```
public class Empregado {
        private int tipo;
        private double salario;
        private double comissao;
        private double bonus;
        private double horasTrabalhadas;
        private double valorHora;
        //getters/setters
}
```

1.4.2 A Listagem 1.4.2.1 gera um vetor de empregados sorteados aleatoriamente.

Listagem 1.4.2.1

```
public class TesteEmpregadoSemPolimorfismo {
	public static void main(String[] args) {
		Random gerador = new Random ();
		//gera uma base de dados de empregados
		Empregado [] empregados = new Empregado[10];
		for (int i = 0; i < empregados.length; i++) {
			//construir o objeto pois cada posição vale null
			empregados[i] = new Empregado();
			int tipo = gerador.nextInt(4) + 1;
			empregados[i].setTipo(tipo);
			if (tipo == 1)
				empregados[i].setSalario(gerador.nextDouble() * 1000 + 1200);
			else if (tipo == 2) {
				empregados[i].setSalario(gerador.nextDouble() * 800 + 1000);
				empregados[i].setComissao(gerador.nextDouble());
			}
			else if (tipo == 3) {
				empregados[i].setSalario(gerador.nextDouble() * 800 + 800);
				empregados[i].setComissao(gerador.nextDouble());
				empregados[i].setBonus(gerador.nextDouble() * 500);
			}
			else if (tipo == 4) {
				empregados[i].setHorasTrabalhadas(80 + gerador.nextInt(41));
				empregados[i].setValorHora(gerador.nextDouble() * 20 + 30);
			}
		}
		//faz as contas
		for (int i = 0; i < empregados.length; i++) {
			double salario = 0;
			if (empregados[i].getTipo() == 1) {
				salario = empregados[i].getSalario();
			}
			else if (empregados[i].getTipo() == 2) {
				salario = empregados[i].getSalario() + empregados[i].getSalario()
* empregados[i].getComissao();
			}
			else if (empregados[i].getTipo() == 3) {
				salario = empregados[i].getSalario() + empregados[i].getSalario()
* empregados[i].getComissao() + empregados[i].getBonus();
			}
			else if (empregados[i].getTipo() == 4) {
				salario = empregados[i].getValorHora() *
empregados[i].getHorasTrabalhadas();
			}
			System.out.printf("Empregado %d: %.2f\n", i + 1, salario);
		}	}}
```

1.5 (Um novo tipo de empregado) O que fazer para adicionar um novo tipo de Empregado? Digamos que agora exista um novo tipo de empregado: o empregado tarefeiro. Ele recebe de acordo com o número de tarefas que realiza e de acordo com o valor da tarefa. A Listagem 1.5.1 mostra os ajustes necessários para que o programa passe a abrigar também esse novo tipo de empregado.

Listagem 1.5.1

```
//na classe Empregado
private double numeroDeTarefas;
		private double valorTarefa;
public double getNumeroDeTarefas() {
			return numeroDeTarefas;
		}


		public void setNumeroDeTarefas(double numeroDeTarefas) {
			this.numeroDeTarefas = numeroDeTarefas;
		}
		public double getValorTarefa() {
			return valorTarefa;
		}


		public void setValorTarefa(double valorTarefa) {
			this.valorTarefa = valorTarefa;
		}
//uma nova possibilidade
int tipo = gerador.nextInt(5) + 1;
//gerando dados de tarefeiro
else if (tipo == 5) {
		empregados[i].setNumeroDeTarefas(500 + gerador.nextInt(501));
		empregados[i].setValorHora(gerador.nextDouble() * 70 + 30);
}


//fazendo cálculos para tarefeiro
else if (empregados[i].getTipo() == 5) {
		salario = empregados[i].getValorTarefa() * empregados[i].getNumeroDeTarefas();
}
```

1.6 (Entendendo o princípio aberto/fechado) Qual o problema da solução proposta? Ela viola ao menos dois princípios: o princípio da alta coesão é um deles, já que a classe Empregado tem mais de uma responsabilidade. O segundo, ainda mais grave, é o **princípio aberto/fechado**. Esse princípio diz que um sistema deve estar **fechado** à alteração de código existente e **aberto** à adição de novo código. Isso é de interesse pois abrir código existente pode dar origem a erros que não existiam anteriormente. Além disso, a estrutura de seleção mantida no programa é trabalhosa e, a cada novo tipo de empregado, o programador deve ele mesmo adicionar uma nova possibilidade, como vimos. O polimorfismo tira das mãos do programador a responsabilidade por lidar

com estruturas de seleção desse tipo e ajuda, portanto, a escrever código que esteja de acordo com o princípio aberto/fechado.

1.7 (Rescrevendo a solução com polimorfismo. Crie um projeto) O uso do polimorfismo permitirá que ambos os princípios da alta coesão e aberto/fechado sejam respeitados. 1.7.1 O primeiro passo é criar uma hierarquia de classes em que cada classe representa um tipo específico de empregado. Veja a Listagem 1.7.1.1.

Nota: @Override é um recurso chamado anotação (do inglês annotation). Ela serve para solicitar ao programador que verifique se o método está de acordo com as regras de sobrescrita de métodos. Por exemplo, o nome dele e a lista de parâmetros devem ser idênticos àqueles do método sendo sobrescrito. O modificador de acesso não pode ser mais restrito do que aquele definido no método sendo sobrescrito. E ainda há outras regras para vermos adiante com as quais não é necessário se preocupar agora. Embora não seja obrigatório especificar @Override em uma sobrescrita, é uma boa prática fazê-lo.

Listagem 1.7.1.1

```
public class EmpregadoHorista extends Empregado {
        private double valorHora;
        private int numeroHoras;
        @Override
        public double calculaSalario() {
                return this.valorHora * this.numeroHoras;
        }
}
```

1.7.2 A classe de teste é exibida pela Listagem 1.7.2.1.

Listagem 1.7.2.1

```
public class TesteComPolimorfismo {
	public static void main(String[] args) {
		Random gerador = new Random ();
		Empregado [] empregados = new Empregado[10];
		for (int i = 0; i < empregados.length; i++) {
			int tipo = gerador.nextInt(4) + 1;
			switch (tipo) {
				case 1:{
					double salario = gerador.nextDouble() * 1500 + 500;
					empregados[i] = new EmpregadoAssalariado(salario);
					break;
				}
				case 2:{
					double salario = gerador.nextDouble() * 1200 + 300;
					double comissao = gerador.nextDouble();
					empregados[i] = new
EmpregadoAssalariadoComissionado(salario, comissao);
					break;
				}
				case 3:{
					double salario = gerador.nextDouble() * 1200 + 300;
					double comissao = gerador.nextDouble();
					double bonus = gerador.nextDouble() * 500;
					empregados[i] =
							new
EmpregadoAssalariadoComissionadoBonificado(salario, comissao, bonus);
					break;
				}
				case 4:{
					int numeroHoras = gerador.nextInt(101) + 60;
					double valorHora = gerador.nextDouble() * 20 + 30;
					empregados[i] = new EmpregadoHorista(valorHora,
numeroHoras);
				}
			}
		}
		//faz as contas, está de acordo com o princípio aberto fechado
		for (int i = 0; i < empregados.length; i++) {
			System.out.printf("Empregado %d: %.2f\n", i + 1,
//polimorfismo dinâmico acontece nessa linha de código
 empregados[i].calculaSalario());
		}
	}
}
```

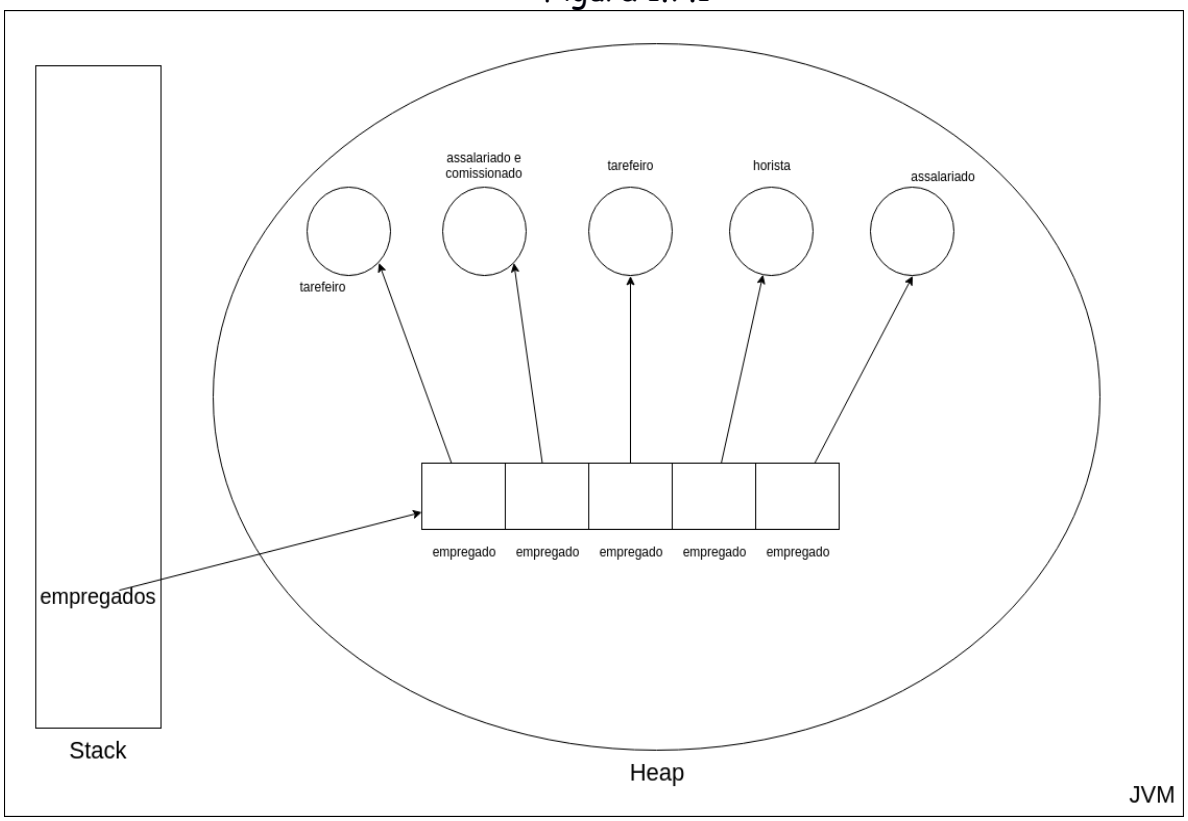
1.7.3 Perceba que adicionar o Empregado Tarefeiro implica em criar uma nova classe para representá-lo. A partir daí, uma vez que ele faça parte do vetor de empregados, a estrutura de repetição responsável por fazer os cálculos de folha de pagamento o processará sem ter de ser alterada. Veja a Listagem 1.7.3.1.

Listagem 1.7.3.1

```
//nova classe
public class EmpregadoTarefeiro extends Empregado {

	private double valorTarefa;
	private int numeroTarefas;


	public EmpregadoTarefeiro(double valorTarefa, int numeroTarefas) {
		this.valorTarefa = valorTarefa;
		this.numeroTarefas = numeroTarefas;
	}
		@Override
	public double calculaSalario() {
		return this.valorTarefa * this.numeroTarefas;
	}
}


//ajustes na geração de empregados
int tipo = gerador.nextInt(5) + 1;
case 5: {
	int numeroTarefas = 1000 + gerador.nextInt(501);
	double valorTarefa = gerador.nextDouble() * 60 + 20;
	empregados[i] = new EmpregadoTarefeiro (valorTarefa, numeroTarefas);
}
```

A Figura 1.7.1 mostra um possível cenário para a memória da JVM quando esse programa é executado.

Figura 1.7.1



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.