PROGRAMAÇÃO ORIENTADA A OBJETOS

AULA 5

Prof. Leonardo Gomes

CONVERSA INICIAL

Nesta aula, o principal conceito que iremos abordar é o polimorfismo, que em linhas gerais é a capacidade de entendermos uma variável como pertencendo a diferentes tipos, baseando-se na forma com que é instanciada. Essa capacidade traz diversas facilidades e aumenta muito a flexibilidade dos códigos orientados a objeto. O polimorfismo também é considerado um dos pilares da programação orientada a objetos.

Ao final desta aula, esperamos atingir os seguintes objetivos, que serão avaliados ao longo da disciplina da forma indicada.

Quadro 1 – Objetivos

Objetivos	**Avaliação**
Aplicar os conceitos de polimorfismo dentro da orientação a objetos.	Questionário e questões dissertativas.
Desenvolver algoritmos que fazem uso de uma lógica que envolva polimorfismo.	Questionário e questões dissertativas.
Conhecer e diferenciar o conceito de classe abstrata e interface.	Questionário e questões dissertativas.

TEMA 1 – POLIMORFISMO

Neste tema, vamos debater o conceito de polimorfismo dentro da orientação a objetos e seus benefícios para a representação de códigos. O nome *polimorfismo* vem do grego (*polýs* = muitas; *morphé* = formas) e geralmente descreve a capacidade de objetos de uma superclasse assumirem a forma (métodos e atributos internos) de diferentes subclasses, mas existem outras situações que também são denominadas de *polimorfismo* que veremos na sequência.

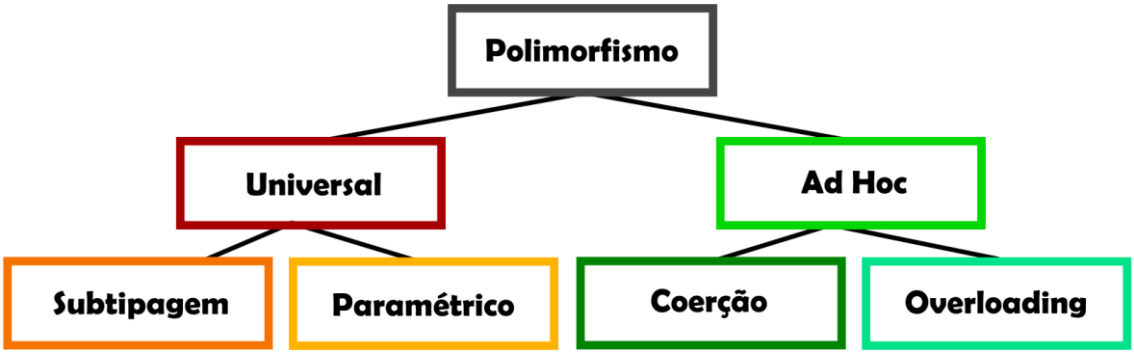
Segundo a classificação mais amplamente adotada na literatura sobre polimorfismo, podemos agrupá-lo em duas categorias, **universal** e **ad hoc**:

- **Universal:** é o tipo de polimorfismo em que temos um mesmo algoritmo, um mesmo comportamento, que pode ser executado para diversas classes ou dados primitivos.

- **Ad Hoc:** é outra forma de polimorfismo, que atua em um conjunto específico de classes para as quais uma mesma chamada de método permite comportamentos diferentes para cada tipo.

Dentro de cada uma das duas categorias, existem outros dois tipos de polimorfismos, totalizando quatro tipos diferentes: **subtipagem**, **paramétrico**, **coerção** e ***overloading***. Veja, a seguir, a organização das diferentes categorias de polimorfismo e uma explicação mais detalhada de cada uma.

Figura 1 – Representação das diferentes categorias de polimorfismo



1.1 Subtipagem

É a forma mais usualmente associada ao nome polimorfismo. Ela ocorre quando temos uma superclasse que possui determinado método, e suas subclasses reimplementam esse método com outro comportamento. Lembramos que uma subclasse, além de herdar métodos e atributos, herda ainda a tipagem, portanto, é possível instanciar uma subclasse e referenciá-la como membro da sua superclasse. No próximo tema, veremos exemplos práticos em código Java.

1.2 Paramétrico

É o tipo de polimorfismo em que uma função e os dados dentro dela podem ser escritos de forma genérica para diferentes tipos de dado. Uma função matemática que você deseja que funcione da mesma forma para valores de entrada do tipo *int*, *float* ou *double* seria um exemplo de uso. Nas linguagens Java e C#, o conceito é chamado de *Generics*, enquanto no C++ damos o nome de *Template*. Veremos detalhes de implementação ainda nesta aula.

1.3 Coerção

Esse tipo de polimorfismo mais frequentemente visto em códigos com tipos primitivos, embora seja possível também com objetos, ocorre quando fazemos conversão seja ela implícita, feitas de forma automática pelo compilador, ou explícita, com código descrevendo a transformação entre tipos diferentes de dados.

As coerções facilitam a escrita dos códigos, e em certas situações podem inclusive aumentar a legibilidade. No entanto, pode ser também fonte de erros de difícil detecção. A seguir, um exemplo de alguns códigos em que a coerção existe.

```
float altura = 2;
int peso = 77.5;
int idade = (int) 30.5;
```

No primeiro caso, **altura** está recebendo um valor inteiro. Considerando que a variável é *float*, existe uma coerção implícita, o dado poderia ser descrito como 2.0f caso o programador prefira deixar explícita que se trata de um *float*. Um erro comum entre iniciantes na programação seria fazer uma divisão como **(float resultado = 3/2;)**. Neste caso, como o valor 3 e 2 são inteiros, a divisão será uma divisão inteira e o resultado será 1, e não 1.5 como se espera. Nesse tipo de situação, indicar que os valores da divisão são *float* evitaria o erro.

No caso das variáveis **peso** e **idade,** a conversão é de *float* para *int*, a primeira de forma implícita e a segunda de forma explícita utilizando o conceito chamado de ***type cast***.

1.4 *Overloading*

Neste tipo de polimorfismo, temos funções com o mesmo nome, mas com parâmetros de entrada diferentes, o que permite que executem códigos distintos. Em português, podemos chamar esse conceito de *sobrecarga*.

Por exemplo, suponha as funções:

```
int maior(int,int);
int maior(int,int,int);
```

A primeira função recebe dois inteiros e retorna o maior, a segunda recebe três inteiros e da mesma forma retorna o maior também. Quase que a totalidade das linguagens modernas de programação permite este tipo de polimorfismo.

TEMA 2 – POLIMORFISMO NA LINGUAGEM JAVA

Neste tema, vamos discutir detalhes da implementação de polimorfismo na linguagem Java e os códigos propriamente ditos. Mas, primeiramente, é importante reforçar o conceito de referência em Java.

2.1 Referência e instanciação

Quando criamos uma variável de uma classe na linguagem Java, esta se comporta como uma referência, semelhante ao conceito de ponteiro da C/C++. Em outras palavras, a variável indica (aponta) uma posição de memória. Para indicar que desejamos criar um novo objeto na memória reservando espaço e efetivamente instanciar o mesmo, utilizamos o comando **new**. No código abaixo, observe que criamos três variáveis da classe *Aluno*, porém, todas fazem referência ao mesmo objeto, o mesmo espaço de memória; dessa forma, se modificarmos uma, a mudança vale para todas as variáveis. Se desejarmos que cada uma seja uma variável independente, deveríamos criar três instâncias (comando *new*).

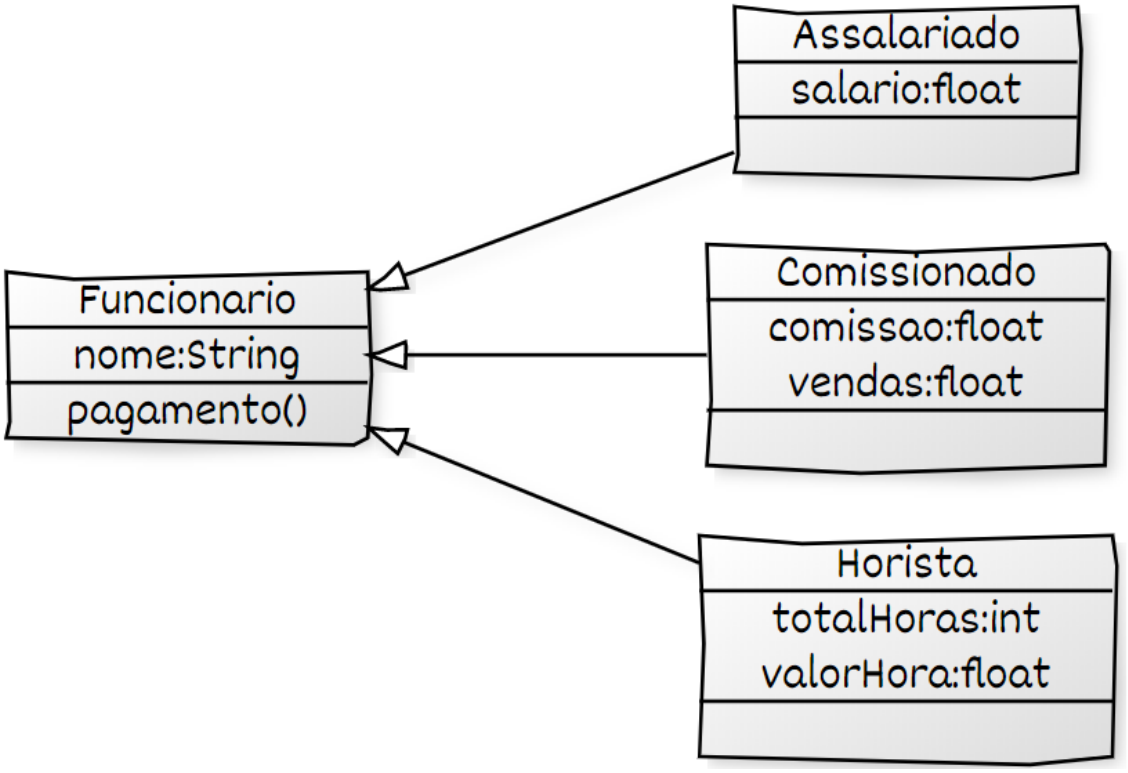
```
Aluno mario = new Aluno();
Aluno luigi = mario;
Aluno yoshi = mario;

mario.nome = "super mario";
System.out.println(mario.nome); //imprimirá  "Super Mario"
System.out.println(luigi.nome);//imprimirá  "Super Mario"
System.out.println(yoshi.nome);//imprimirá  "Super Mario"
luigi.nome = "Super luigi";
System.out.println(mario.nome); //imprimirá  "Super Luigi"
System.out.println(luigi.nome); //imprimirá  "Super Luigi"
System.out.println(yoshi.nome); //imprimirá  "Super Luigi"
```

2.2 Exemplo prático de polimorfismo

Vamos partir para um exemplo prático para analisar o polimorfismo da categoria subtipagem, a forma mais comumente associada ao polimorfismo. Supomos a seguinte situação: temos uma classe para representar os **funcionários** de uma empresa, no entanto, os funcionários dessa empresa são contratados seguindo regimes diferentes. Existem **assalariados**, que recebem um valor fixo mensal, os **comissionados**, que recebem um percentual das vendas que realizam, e os **horistas**, que recebem baseado no número de horas trabalhadas. Uma forma elegante de representar essa situação utilizando orientação a objetos seria criar uma superclasse para funcionário e subclasses diferentes para as diferentes categorias de funcionário. A seguir, o diagrama UML e código das classes.

Figura 2 – Diagrama



```
public class Funcionario {

	String nome;

	public Funcionario(String nome) {
		this.nome = nome;
	}

	public float pagamento() {
		System.out.println("Calculo de pagamento");
		return 0;
	}

}

public class Assalariado extends Funcionario{
	float salario;

	public Assalariado(String nome,float salario) {
		super(nome);
		this.salario=salario;
	}

	public float pagamento(){
		return salario;
	}
}

public class Comissionado extends Funcionario{
	float vendas;
	float comissao;
	public Comissionado(String nome,float vendas, float
comissao) {
		super(nome);
```

```
            this.vendas= vendas;
            this.comissao= comissao;
      }

      public float pagamento(){
            return comissao*vendas;
      }
}

public class Horista extends Funcionario{
      int horasTrabalhadas;
      float precoHora;
      public Horista(String nome,int horasTrabalhadas, float
precoHora) {
            super(nome);
            this.horasTrabalhadas= horasTrabalhadas;
            this.precoHora= precoHora;
      }

      public float pagamento(){
            return precoHora*horasTrabalhadas;
      }
}
```

Até aqui, nenhuma novidade no código, as classes específicas foram criadas cada uma com um método pagamento diferente e atributos próprios para representar a forma que esse pagamento é calculado. No entanto, observe que é possível declarar uma variável da superclasse **Funcionario** e fazer a mesma apontar para objetos instanciados como subclasses **Assalariado, Comissionado** e **Horista**. Nesse caso, a chamada do método *pagamento* tem duas opções, executar a implementação da superclasse ou a da subclasse, nessa situação, será sempre executado o método da subclasse, conforme a variável foi instanciada. Confira essa situação no código a seguir.

```
Funcionario f;

f = new Horista("Mario",100,40.5f);
System.out.println("Horista: " +f.pagamento());
f = new Comissionado("Luigi",50000,0.05f);
System.out.println("Comissionado: " +f.pagamento());
f = new Assalariado("Yoshi",3500);
System.out.println("Assalariado: " +f.pagamento());
```

Talvez você se pergunte: “qual é a grande vantagem do polimorfismo?”. Imagine que desejamos armazenar todos os nossos funcionários em uma estrutura de ArrayList. Sem polimorfismo, a única forma seria a criação de três ArrayLists, pois temos três subclasses de funcionários. Desse modo, seria necessário um *array* para cada, dando muito mais trabalho na hora de realizar uma busca ou qualquer outro código. No entanto, com o uso de polimorfismo, podemos criar um único *array* da superclasse funcionário e adicionar objetos das diferentes subclasses que executarão sempre os métodos conforme foram instanciadas. No exemplo a seguir, vemos um código que adiciona alguns funcionários de subclasses distintas em um mesmo *array* e, na sequência, calcula o total do pagamento de todos os funcionários deste *array*.

```
ArrayList<Funcionario> listaFuncionarios = new
ArrayList<Funcionario>();

listaFuncionarios.add( new Horista("Mario",100,40.5f) );
listaFuncionarios.add( new
Comissionado("Luigi",50000,0.05f) );
listaFuncionarios.add( new Assalariado("Yoshi",3500) );

float totalPagamento=0;
for (Funcionario f : listaFuncionarios) {
      totalPagamento+= f.pagamento();
}
System.out.println("Total pagamento: " + totalPagamento);
```

TEMA 3 – CLASSE ABSTRATA

No contexto de orientação a objetos, eventualmente teremos classes que não desejamos instanciar. Se formos analisar, na vida real existem conceitos abstratos que não possuem um objeto físico ligado a eles, por exemplo, uma forma geométrica. Sem especificarmos que forma geométrica estamos nos referindo, fica impossível analisar qualquer uma de suas propriedades físicas. Desse modo, podemos afirmar que uma forma geométrica ocupa uma certa área, mas sem especificar que forma, é impossível dizer como calcular essa medida.

A ideia de classe abstrata propõe um conceito análogo que, na prática, implica que uma classe que você especifique como abstrata não poderá ser instanciada. Mas de que serve uma classe que não pode ser instanciada?

Propriedades das classes abstratas:

1. **Pode referenciar objetos de subclasses** graças ao polimorfismo.
2. Permite que criemos **métodos desprovidos de implementação**, mas que obrigatoriamente devem ser implementados por suas subclasses.
3. **Não podem ser instanciadas**, ocorre um erro ao tentarmos instanciar um objeto dessa classe. Erros de compilação, ao contrário do senso comum, são muito positivos pois são fáceis de se identificar e corrigir, diferentemente de um *bug* que só será identificado durante a execução do programa e não possui qualquer indicação de onde possa estar no código. Os erros de compilação caem em duas categorias: erros léxicos, quando um comando é escrito errado, e semânticos, quando o comando é escrito corretamente, mas em local que não faça sentido; o compilador detecta esses erros, que são fáceis de localizar. Erros semânticos são aqueles em que o comando está escrito certo, no local certo, mas executa instruções diferentes das desejadas, em outras palavras, o significado do que está escrito não é o que se imaginava.

Poderíamos nunca adotar classes abstratas a princípio sem grandes prejuízos, mas ao pensarmos em projetos grandes com muitos programadores participando, fica muito fácil descrever uma classe abstrata para que sirva de modelo para subclasses feitas por outros programadores.

Vamos analisar um exemplo prático: suponha uma classe FormaGeometrica que possua um método chamado *calculaArea*. Esse método

não será implementado pela classe FormaGeometrica, mas será obrigatória para suas classes filhas. Veja o código a seguir.

```
01.     public abstract class FormaGeometrica {
02.           public double area;
03.           public abstract void calculaArea();
04.     }

05.     class Quadrado extends FormaGeometrica{
06.           public void calculaArea(){
07.                 Scanner teclado = new Scanner(System.in);
08.                 System.out.println("Digite a medida do
   lado");
09.                 double lado = teclado.nextDouble();
10.                 area = lado*lado;
11.           }
12.     }

13.     class Circulo extends FormaGeometrica{
14.           public void calculaArea(){
15.                 Scanner teclado = new Scanner(System.in);
16.                 System.out.println("Digite o raio");
17.                 double raio = teclado.nextDouble();
18.                 area = raio*raio*Math.PI;
19.           }
20.     }

21.     public static void main(String[] args) {
22.                 FormaGeometrica forma;
23.
24.                 //Forma é um circulo
25.                 forma = new Circulo();
26.                 forma.calculaArea();
27.                 System.out.println(forma.area);
```

```
28.
29.                 //Forma agora é um quadrado
30.                 forma = new Quadrado();
31.                 forma.calculaArea();
32.                 System.out.println(forma.area);
33.           }
```

No código anterior, observe que na linha 01 utilizamos a palavra reservada *abstract* para indicar que a classe é abstração e pode ser instanciada com o comando **new**. Na linha 03, indicamos que o método calculaArea é abstrato, o que implica que os filhos não abstratos de FormaGeometrica são obrigados a implementá-los; caso contrário, a IDE apontará errado, o que facilita que programadores que decidam criar classes filhas de FormaGeometrica sigam adequadamente os escopos das funções.

TEMA 4 – INTERFACE

Neste tema, vamos debater um conceito bastante próximo ao de classe abstrata, as interfaces.

Na linguagem Java, não é possível termos múltiplas superclasses para uma mesma subclasse, certas linguagens permitem isso, como o C++, porém, não é o caso do Java. No entanto, temos o que chamamos de *Interface*.

A palavra *interface* diz respeito ao meio de ligação/comunicação entre dois sistemas. Por exemplo, é possível ligar uma televisão em um computador por meio de uma interface HDMI, o protocolo de comunicação envolvido define características elétricas, como tensão e corrente, bem como o padrão de transmissão, taxa de símbolos e velocidade de transmissão. Ela permite ligar um mesmo computador com diferentes televisores criados por fabricantes distintas, e tudo funciona de forma transparente, desde que o protocolo de Interface HDMI seja respeitado por ambos. A analogia aqui é a mesma para a programação. Suponha que temos uma classe BancoDeDados, que tem os seus métodos acessados por uma classe Principal. Se desejarmos substituir a classe BancoDeDados por uma outra classe BancoDeDadosAlternativo, que internamente se comunique com um banco diferente, teremos que adaptar a chamada dos métodos pela classe Principal, gerando um retrabalho muito grande que é desnecessário, visto que, em essência, as classes BancoDeDados

e BancoDeDadosAlternativo fazem as mesmas coisas, porém utilizando estratégias internas diferentes. Esse é o cenário perfeito para o uso de interface, pois ela força que classes diferentes implementem métodos com os mesmos nomes, entradas e saídas, garantindo que a classe Principal possa acessar as diferentes classes de banco de dados da mesma forma.

Em resumo, uma interface pode ser entendida como um protocolo que explica como deve ser a assinatura dos métodos de uma classe.

As interfaces funcionam de forma semelhante a classes que não podem ser instanciadas, mas podem servir de referência para cenários de polimorfismo. Vejamos o código abaixo:

```
01.     interface Animal {
02.       public void emitirSom(); //método sem corpo
03.       public void dormir(); // método sem corpo
04.     }
05.     class Gato implements Animal {
06.       public void emitirSom() {
07.         System.out.println("O gato fala: miau miau");
08.       }
09.       public void dormir() {
10.         System.out.println("Zzz");
11.       }
12.     }
13.
14.     class Principal {
15.       public static void main(String[] args) {
16.         Gato tom = new Gato();
17.         tom.emitirSom();
18.         tom.dormir();
19.         Animal obj = tom;
20.         obj.emitirSom();
21.         obj.dormir();
22.       }
23.     }
```

No Java, declaramos interface dentro de seu próprio arquivo semelhante a uma classe e, no lugar da palavra ***class***, escrevemos ***interface*** (linha 01). Dentro do bloco de código associado à interface, declaramos a assinatura dos métodos que desejamos que sejam implementados pelas classes.

No exemplo anterior, a classe Gato implementa a interface Animal (linha 05), obrigando a classe Gato a implementar os métodos da interface. Observe que a classe Gato poderia implementar mais de uma interface apenas separando elas por vírgula (por exemplo: `class Gato implements Animal,Mamifero`). Interface funciona de forma muito semelhante à herança, no entanto, em vez da palavra ***extends,*** utilizamos a palavra ***implements***. Atente para a diferença na nomenclatura, entendemos que as classes implementam interfaces e herdam superclasses. Quais são as diferenças entre interface e classe abstrata?

Quadro 2 – Interface e classe abstrata

Propriedade	**Interface**	**Classe abstrata**
Herança	Classes podem implementar diversas interfaces.	Uma classe só pode herdar uma única superclasse.
Métodos	Interface só possui a assinatura dos métodos.	Uma classe abstrata pode implementar códigos dentro de seus métodos, que serão ou não sobrescritos.
Atributos	Só pode possuir atributos estáticos.	Pode ter tanto atributos estáticos quanto não estáticos.
Adaptação	Fácil adaptar uma classe existente para implementar uma interface, bastando implementar os métodos conforme a assinatura.	Adaptar uma classe existente para herdar uma classe abstrata pode ser trabalhoso por ser necessário modificar a hierarquia já existente de heranças.
Quando usar?	Classes que compartilham mesmo comportamento, assinatura.	Classes que compartilham os mesmos atributos e precisam ter seu estado avaliado de forma compatível.

Modificações adicionais	Ao adicionar um novo método a uma interface, todas as classes devem trazer suas implementações.	Ao adicionar um novo método, é possível trazer uma implementação padrão que servirá para todas as classes filhas.

TEMA 5 – ENUM

Em diversas linguagens de programação, incluindo a Java, existe o conceito de enum, que consiste de uma classe especial de rápida e simples implementação que é capaz de representar um grupo de constantes enumerando-as geralmente por debaixo dos panos, com inteiros em sequência. Geralmente é adotado para representar estados.

Para criar um enum, utilizamos a palavra reservada **enum** em vez de **class** ou **interface**. E dentro do corpo do enum, deve vir o nome das constantes separadas por vírgula, pela convenção estabelecida oficialmente pela Oracle. Ademais, as constantes devem sempre descritas em letras maiúsculas na linguagem Java.

Suponha que desejamos implementar um sistema para um site que vende roupas, em que cada roupa é destinada a uma estação do ano. Como representarmos isso? Temos diversas estratégias:

- ***String***: poderíamos utilizar uma *string* com o nome da estação, o que ocuparia um espaço de memória muito além do necessário para uma tarefa tão simples, tornando a comparação entre estações lenta. *String estacaoRoupa = "inverno".*
- **Int**: podemos criar uma estratégia na qual associamos um inteiro diferente para cada estação, (inverno=1, primavera=2,...). Rápido para comparar, ocupa pouca memória, mas gera códigos pouco legíveis. *int estacaoRoupa = 1.*
- **Constantes**: semelhante à solução com uso de inteiros, porém, criamos uma coleção de constantes associando cada estação com um número, solução rápida e que deixa o código mais claro. *int estacaoRoupa = INVERNO;*
- **Enum**: semelhante ao uso de constantes, no entanto, facilmente podemos agrupar as constantes em um conjunto comum denominado

Estacao e declararmos a variável do próprio tipo Estacao, deixando ainda mais claro o código. *Estacao estacaoRoupa = Estacao.INVERNO.*

Veja a seguir o exemplo de como ficaria o enum *Estacao*.

```
enum Estacao {
  VERAO,
  OUTONO,
  INVERNO,
 PRIMAVERA
}


public class Principal{
  public static void main(String[] args) {
    Estacao estacaoRoupa = Estacao.INVERNO;

    switch(estacaoRoupa) {
      case VERAO:
        System.out.println("Arrase na praia");
        break;
      case OUTONO:
         System.out.println("Passe o outono com elegância");
        break;
      case INVERNO:
        System.out.println("Se agasalhe bem e com estilo");
        break;
      case PRIMAVERA:
        System.out.println("Se vista bem na estação das
flores");
        break;
    }
  }
}
```

É possível ainda fazer um *loop* por todas as estações utilizando o método *values()* que retorna um *array* contendo todas as constantes de um enum.

```
for (Estacao est : Estacao.values()) {
  System.out.println(est);
}
```

Enum pode ser declarado no próprio arquivo como uma classe, e geralmente é, ou pode ser declarado internamente dentro de uma classe existente para ser utilizado apenas localmente.

Enum, na linguagem Java, é representado como uma classe, que inclusive pode possuir métodos e atributos, no entanto, suas constantes são sempre entendidas como **public**, **static**, **final.**

- *public*: as constantes são acessíveis por qualquer classe que possua visibilidade do enum.
- *static*: existe apenas uma constante para a classe inteira, e não uma por instância.
- *final*: não pode ter o valor alterado em tempo de execução.

Quando utilizar enum? Ele é recomendado especialmente em situações em que você tem que descrever uma coleção de valores fixos que não variam, como meses, cores, baralho de cartas etc.

FINALIZANDO

Nesta aula, debatemos sobre o conceito de polimorfismo, primeiramente de forma geral, dentro do contexto de orientação a objetos como um todo, e posteriormente suas especificidades na linguagem Java.

Na sequência, vimos a questão da criação de superclasses abstratas e o conceito bastante semelhante da interface, discutimos as suas diferenças e aplicações.

Por fim, discutimos o conceito de enum, que facilita o trabalho de agruparmos e classificarmos conjunto de constantes, geralmente adotado para representar estados de alguma situação.

Nas próximas aulas, vamos finalizar nosso conteúdo teórico debatendo as exceções, tratamento de erros e alguns conceitos adicionais dentro da linguagem Java.

REFERÊNCIAS

BARNES, D. J.; KÖLLING, M. **Programação orientada a objetos com Java**. 4. ed. São Paulo: Pearson Prentice Hall, 2009.

DEITEL, P.; DEITEL, H. **Java Como programar**. 10. ed. São Paulo: Pearson, 2017.

LARMAN, C. **Utilizando UML e Padrões**: Uma Introdução à Análise e ao Projeto Orientados a Objetos e ao Desenvolvimento Iterativo. 3. ed. Porto Alegre: Bookman, 2007.

MEDEIROS, E. S. de. **Desenvolvendo software com UML 2.0**: definitivo. São Paulo: Pearson Makron Books, 2004.

PAGE-JONES, M. **Fundamentos do desenho orientado a objetos com UML**. São Paulo: Makron Book, 2001.

PFLEEGER, S. L. **Engenharia de software**: teoria e prática. 2. ed. São Paulo: Prentice Hall, 2004.

SINTES, T. **Aprenda programação orientada a objetos em 21 dias**. 5ª reimpressão. São Paulo: Pearson Education do Brasil, 2014.

SOMMERVILLE, I. **Engenharia de software**. 9. ed. São Paulo: Pearson, 2011.