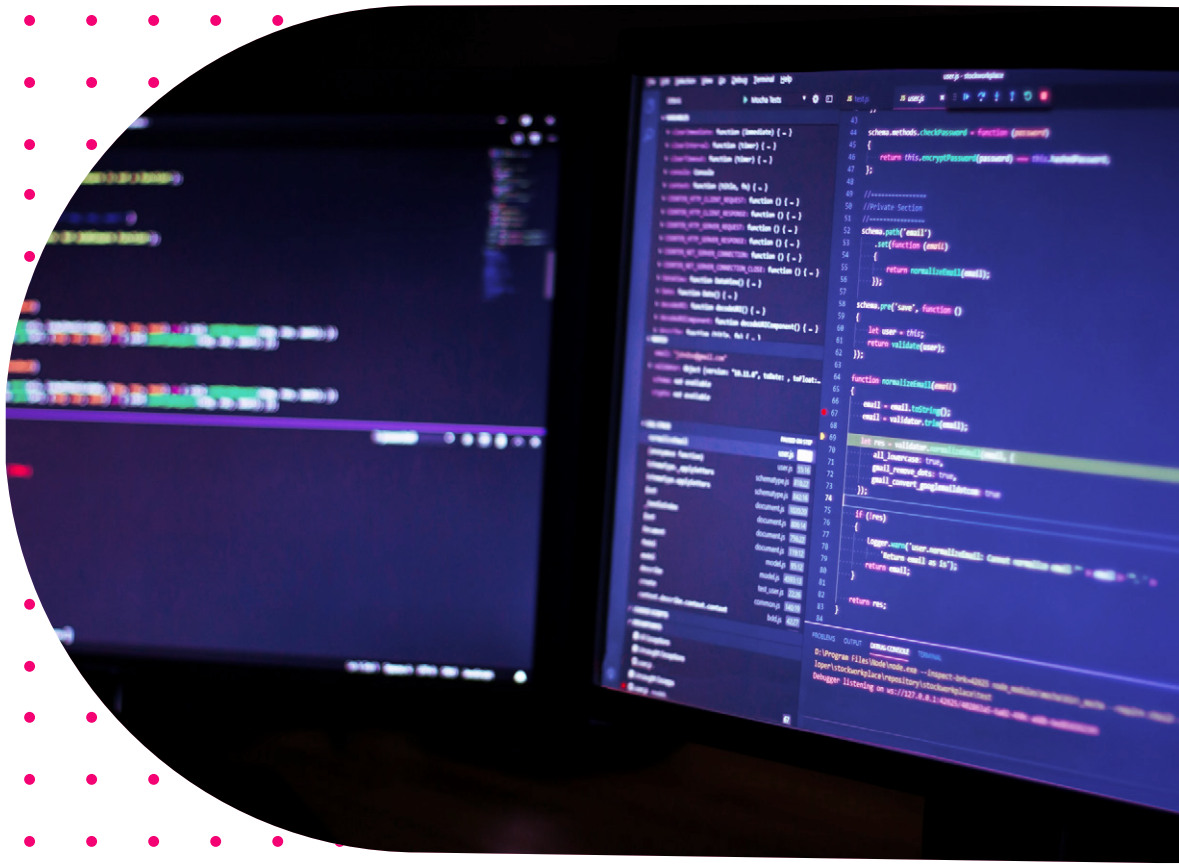
HERANÇA E POLIMORFISMO

SENAI <LAB365>

SUMÁRIO

ORIENTAÇÃO A OBJETOS – RESUMO

O paradigma da Orientação a Objetos (OO) torna mais fácil o reuso de códigos, acoplamento, coesão, entre outras técnicas, para deixar o código mais profissional e de fácil manutenção. A Orientação a Objetos preocupa-se com a modelagem dos processos (análise e projeto).



Antes de chegar até aqui, provavelmente você já tenha passado por alguns conceitos, como o de classes e objetos (a classe, com seus atributos e métodos, ao ser instanciada, cria um objeto – objetos só existem em tempo de execução) e o de encapsulamentos (que evitam resultados inesperados ou acessos indevidos).

Com isso fresco na memória, enfim, chegou a hora de conhecermos Herança e Polimorfismo.

ENTENDENDO HERANÇA

A herança permite a criação de novas classes a partir de outras previamente criadas, é um relacionamento entre classes. Quando uma classe é criada com herança, ela herda os atributos e métodos da outra. Essa classe (chamada de subclasse, classe derivada ou classe filha) passa a ser uma extensão da outra classe (chamada de superclasse, classe base, classe pai ou classe mãe). Além do acesso aos atributos e métodos, também é possível criar novos métodos ou sobrepor os métodos da classe pai.

É possível haver uma hierarquia de classes em vários níveis. Porém, é importante salientar que, quanto mais níveis, mais complexa fica a implementação ao se herdar essas classes.

A herança deve ser aplicada ao desenvolvermos uma classe da forma mais abstrata possível, podendo-se reutilizar seu conceito e seus membros em conceitos similares. Dessa forma, evitamos a repetição de códigos pelo sistema.

Para compreendermos melhor a herança, um exemplo básico é a relação entre o ser humano e um cão. Eles são muito diferentes, mas possuem características em comum: são mamíferos. Também, um beija-flor e um avestruz são bem diferentes, mas ambos são aves. Por sua vez, podemos notar uma relação entre os mamíferos e as aves, ambos são animais. Com isso, concluímos que animal seria a classe mais abstrata até o momento.

Para ficar mais claro, repare na imagem abaixo:

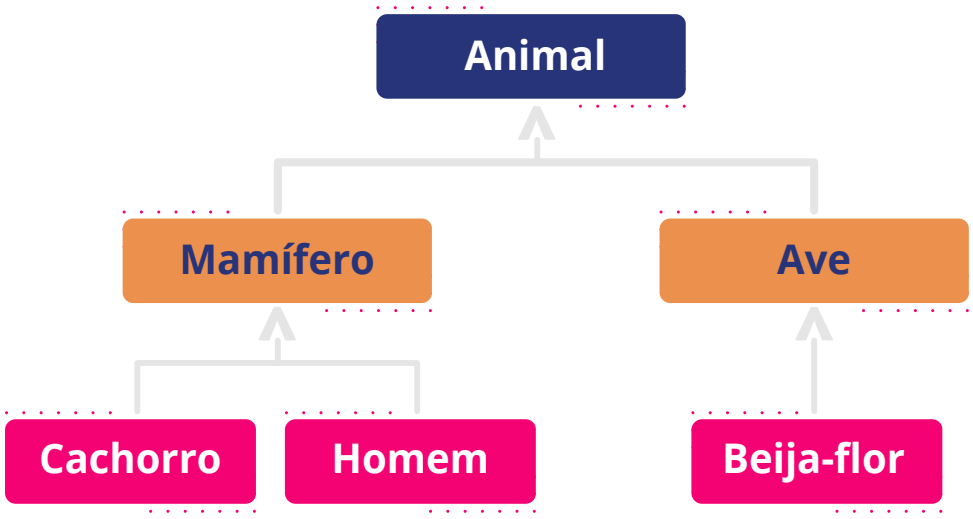


Figura 1 - Exemplo de herança

Adaptado de Melo (2011)

Praticando

Chegou o grande momento de praticarmos. Vamos criar, então, uma classe abstrata para ser o topo da hierarquia. Ao trabalharmos com uma classe abstrata, indica-se que ela não pode ser instanciada diretamente, ou seja, serve somente como modelo para que outras classes a estendam.

Desse modo, vamos criar um pacote de nome “modelos” e, dentro dele, criar a classe abstrata “Planta”, com três atributos informativos e um atributo para uso interno, sendo eles: nome, tipo, porte e plantado (um booleano já recebendo o valor falso). Em seu construtor, constarão os atributos nome, tipo e porte.

Quanto aos métodos, vamos criar algumas ações de acesso restrito às classes filhas, tais como regar, podar, adubar e tomar sol. Contaremos, também, com um método privado para verificar se foi realizado o plantio e, claro, incluiremos algumas ações públicas, sendo elas: plantar e cultivar.

Não esqueça de manter o encapsulamento, criando os métodos *getters* e *setters.* Como o nosso atributo “plantado” é para uso interno, vamos deixar o *setter* dele com modificador *protected*.

A classe ficará assim:

```
public abstract class Planta {

	private String nome;
	private String tipo;
	private String porte;
	private boolean plantado = false;

	/* Construtor */
	public Planta(String nome, String tipo, String porte) {
		this.nome = nome;
		this.tipo = tipo;
		this.porte = porte;
	}

	/* Métodos comuns */
	public void plantar() {
		if (plantado) {
			System.out.println(nome + ": Plantio já realizado
			anteriormente!\n");
			return;
		}
		System.out.print(nome + ": Plantando...");
		plantado = true;
		System.out.println(".....");
		System.out.println(nome + ": Plantio realizado!\n");
	}
	public void cultivar() {
		if (!verificaPlantio())
			return;
		regar();
		podar();
		adubar();
		tomarSol();
	}
	protected void regar() {
		if (!verificaPlantio())
			return;
		System.out.print(nome + ": Regando");
		System.out.println(".....");
		System.out.println(nome + ": Regada\n");
	}
```

```
protected void podar() {
      if (!verificaPlantio())
            return;
      System.out.print(nome + ": Realizando a poda");
      System.out.println(".....");
      System.out.println(nome + ": Poda realizada\n");
}
protected void adubar() {
      if (!verificaPlantio())
            return;
      System.out.print(nome + ": Adubando " + nome);
      System.out.println("......");
      System.out.println(nome + ": Adubada\n");
}
protected void tomarSol() {
      if (!verificaPlantio())
            return;
      System.out.print(nome + ": Tomando sol");
      System.out.println("................");
      System.out.println(nome + ": Retirada do sol\n");
}
private boolean verificaPlantio() {
      boolean plantado = isPlantado();
      if (!plantado) {
            System.out.println(nome + ": Plantio não realizado!
            Plante antes de cultivar!\n");
      }
      return plantado;
}

/* Getters e Setters */
public String getNome() {
      return nome;
}
public void setNome(String nome) {
      this.nome = nome;
}
public String getTipo() {
      return tipo;
}
public void setTipo(String tipo) {
      this.tipo = tipo;
}
```

```
	public String getPorte() {
		return porte;
	}
	public void setPorte(String porte) {
		this.porte = porte;
	}
	public boolean isPlantado() {
		return plantado;
	}
	protected void setPlantado(boolean plantado) {
		this.plantado = plantado;
	}

	@Override
	public String toString() {
		return "\n# " + nome + "\n# Planta do tipo " + tipo +
"\n# Porte " + porte;
	}
}
```

Para estender a classe “Planta”, vamos criar, no mesmo pacote, uma classe “Arbusto”. Nela, criaremos um construtor somente com o parâmetro “nome”, pois, no lugar dos atributos “tipo” e “porte”, vamos enviar respectivamente os textos fixos “Arbusto” e “Médio” e repassaremos tudo isso ao construtor da classe pai.

Nessa classe, vamos criar um método “modelar” para adicionar essa ação ao cultivo. Também, vamos sobrepor o método “cultivar”, utilizar nele o método da classe pai, para então incrementarmos com nosso novo método “modelar”. Dessa forma, o método “cultivar” fará tudo o que faz na classe pai e, em seguida, continuará com a nossa nova implementação.

A classe ficará assim:

```
public class Arbusto extends Planta {
	public Arbusto(String nome) {
		super(nome, “Arbusto”, “Médio”);
	}

	@Override
	public void cultivar() {
		super.cultivar();
		modelar();
	}

	public void modelar() {
		System.out.print(getNome() + “: Modelando...”);
		System.out.println(“...............”);
		System.out.println(getNome() + “: Modelado!\n”);
	}
}
```

Estamos prontos para utilizar esta classe em nossa classe com método “*main*”. Então, vamos criar, ao lado do pacote “modelos”, uma classe “HerancaPolimorfismoMain”. Dentro do seu método “*main*”, vamos criar uma instância de “Arbusto” com nome “Pingo de ouro” e chamar os métodos “plantar” e “cultivar”.

A classe ficará assim:

```
public class HerancaPolimorfismoMain {

	public static void main(String[] args) {
		Arbusto arbusto = new Arbusto(“Pingo de ouro”);
		System.out.println(arbusto);
		arbusto.plantar();
		arbusto.cultivar();
	}
}
```

Ao executarmos o método, o retorno no console será o seguinte:

```
# Pingo de ouro
# Planta do tipo Arbusto
# Porte Médio
Pingo de ouro: Plantando...................
Pingo de ouro: Plantio realizado!

Pingo de ouro: Regando.....................
Pingo de ouro: Regada

Pingo de ouro: Realizando a poda...........
Pingo de ouro: Poda realizada

Pingo de ouro: Adubando Pingo de ouro......
Pingo de ouro: Adubada

Pingo de ouro: Tomando sol.................
Pingo de ouro: Retirada do sol

Pingo de ouro: Modelando...................
Pingo de ouro: Modelado!
```

Como pudemos observar, mesmo sem a implementação do método “plantar”, ele pôde ser executado. Isso ocorreu justamente devido à herança, percebendo-se a necessidade de alterar o método somente quando o comportamento foge do padrão. Na sequência, vamos simular a sobreposição total de um método.

Agora, no mesmo pacote “modelos”, vamos criar a classe “Aquatica"”, que também vai estender a classe “Planta”. Nela, criaremos um construtor sem o atributo “tipo”, pois vamos utilizar o texto fixo “Aquatica” em seu lugar e repassaremos todo o resto ao construtor da classe pai (semelhante à forma que implementamos na classe “Arbusto”). Nessa classe, vamos sobrepor os métodos “plantar” e “cultivar”, pois essas ações são diferentes com plantas aquáticas.

A classe ficará assim:

```
public class Aquatica extends Planta {
	public Aquatica(String nome, String porte) {
		super(nome, “Aquática”, porte);
	}
	@Override
	public void plantar() {
		System.out.println(getNome() + “ solta sobre a
		água!”);
		setPlantado(true);
	}
	@Override
	public void cultivar() {
		tomarSol();
	}
}
```

Note que não criamos novos métodos, somente utilizamos alguns outros próprios da classe pai para a realização das ações necessárias. Com isso, em nosso método “*main*”, vamos comentar os códigos referentes ao arbusto e criar uma instância da classe “Aquatica”, com nome “Vitória-régia” e de porte “Grande”, e, da mesma forma que fizemos com a classe “Arbusto”, chamaremos os métodos “plantar” e “cultivar”.

O código ficará desta forma:

```
public class HerancaPolimorfismoMain {

	public static void main(String[] args) {
//		Arbusto arbusto = new Arbusto(“Pingo de ouro”);
//		System.out.println(arbusto);
//		arbusto.plantar();
//		arbusto.cultivar();
		Aquatica aquatica = new Aquatica(“Vitória-régia”,
	“Grande”);
		System.out.println(aquatica);
		aquatica.plantar();
		aquatica.cultivar();
	}

}
```

Ao executarmos o método, teremos um retorno mais breve no console:

```
# Vitória-régia

# Planta do tipo Aquática

# Porte Grande

Vitória-régia solta sobre a água!

Vitória-régia: Tomando sol.................

Vitória-régia: Retirada do sol
```

Assim, concluímos que tudo o que é comum às classes pertencentes à hierarquia fica implementado na classe pai e, assim, vai se estendendo às demais classes, as quais, por sua vez, podem também ser estendidas por outras, criando, assim, classes com comportamentos cada vez mais específicos. Até o momento, estendemos apenas um nível, porém a hierarquia pode seguir, por exemplo, fazendo com que a classe “Aquatica” seja estendida por outra.

ENTENDENDO POLIMORFISMO

Nos exemplos acima, criamos métodos na classe “Planta” que foram parciais ou completamente modificados em suas classes filhas para que tivessem comportamentos distintos.

Usamos polimorfismo, apresentando um exemplo de nosso projeto atual, quando precisamos utilizar um método da classe “Planta” (classe pai) em uma instância de sua classe filha. Na classe filha, esse método é marcado pela notação “*@Override*” e sua assinatura (nome e lista de parâmetros) é mantida como a de sua classe pai.

Até então, sempre criamos instâncias do mesmo tipo da declaração e as utilizamos como se fossem a classe filha. Por exemplo, a variável “arbusto” foi declarada do tipo “Arbusto” (desta forma: “Arbusto arbusto = new Arbusto(“Pingo de ouro”)”). Para exemplificar o polimorfismo, vamos criar um método que receba como parâmetro uma instância de “Planta” e que utilize os métodos existentes em “Planta”.

Para um melhor detalhamento sobre o que faremos em nossa classe “HerancaPolimorfismoMain”, criaremos um método “plantarCultivar” que receberá como parâmetro uma instância de “Pessoa” e que fará uso das ações “plantar” e “cultivar”. Vamos também substituir os métodos “plantar” e “cultivar” usados com as instâncias de “arbusto” e “Aquatica” para usar nosso novo método, passando suas respectivas instâncias como parâmetro.

A classe ficará assim:

```
public class HerancaPolimorfismoMain {

	public static void main(String[] args) {

		Arbusto arbusto = new Arbusto(“Pingo de ouro”);

		System.out.println(arbusto);

		plantarCultivar(arbusto);


		Aquatica aquatica = new Aquatica(“Vitória-régia”,
		“Grande”);

		System.out.println(aquatica);

		plantarCultivar(aquatica);

	}

	private static void plantarCultivar(Planta planta) {

		planta.plantar();

		planta.cultivar();

	}

}
```

Antes de executar o método “*main*”, vamos relembrar que as classes “Arbusto” e “Aquatica” estendem a classe “Planta”. Em “Arbusto”, utilizamos a sobreposição do método “cultivar” e o incrementamos com o método “modelar”, e, em “Aquatica”, sobrepomos os métodos “plantar” e “cultivar”, reescrevendo ambos e utilizando alguns outros métodos da classe pai.

Enfim, executaremos o método, e o resultado no console será:

```
# Pingo de ouro
# Planta do tipo Arbusto
# Porte Médio
Pingo de ouro: Plantando...................
Pingo de ouro: Plantio realizado!

Pingo de ouro: Regando.....................
Pingo de ouro: Regada

Pingo de ouro: Realizando a poda...........
Pingo de ouro: Poda realizada

Pingo de ouro: Adubando Pingo de ouro......
Pingo de ouro: Adubada

Pingo de ouro: Tomando sol.................
Pingo de ouro: Retirada do sol

Pingo de ouro: Modelando...................
Pingo de ouro: Modelado!

# Vitória-régia
# Planta do tipo Aquática
# Porte Grande
Vitória-régia solta sobre a água!
Vitória-régia: Tomando sol.................
Vitória-régia: Retirada do sol
```

Ao analisarmos o console, notamos que o comportamento é idêntico ao anterior. Isso porque o polimorfismo nos possibilita total flexibilidade em implementar ou sobrepor métodos das classes estendidas.

O comportamento que será adotado por um método só será definido durante a execução, e o método a ser executado sempre respeitará a hierarquia. Ao indicar que o parâmetro é do tipo “Planta”, fica implícito que a instância recebida precisa ser do tipo “Planta” ou fazer parte de sua hierarquia, tornando-se, assim, uma boa prática para a reutilização de código.

Se alterarmos o tipo do parâmetro de “Planta” para “Arbusto”, não poderemos utilizar esse método, passando-se uma instância de “Aquatica” (assim como se alterarmos para “Aquatica” e passarmos uma instância de “Arbusto”). Isso se deve ao fato de que, ao analisarmos a classe “Arbusto”, veremos que ela possui o método “modelar”, que não faz parte da estrutura de “Aquatica”. Ao realizarmos a troca de tipo “Planta” para “Arbusto”, a IDE não saberá resolver a situação ao utilizar o método com “Aquatica”, mesmo que em ambas as classes exista a implementação dos métodos a serem utilizados, como se observa na imagem abaixo:

Figura 2 - Alteração de parâmetro

do Autor (2022)

REFERÊNCIAS

CARVALHO, T. L. **Orientação a Objetos:** Aprenda seus conceitos e suas aplicabilidades de forma efetiva. São Paulo (SP): Editora Casa do Código, 2016.

FORTUNATO, C. **Herança em JAVA:** Teoria e prática. Medium, 5 mar./2020. Disponível em: https://medium.com/caiquefortunato/heran%C3%A7a-em-java-teoria-e-pr%C3%A1tica-2ca7d9b0f3de. Acesso em: 29 maio 2022.

MELO, I. V. A. **Herança x Composição.** Recapitulando - PET News, Computação – Universidade Federal de Campina Grande. Campina Grande (PB), jun./2011. Disponível em: http://www.dsc.ufcg.edu.br/~pet/jornal/junho2011/materias/recapitulando.html. Acesso em: 29 maio 2022.

RICARTE, I. L. **Polimorfismo.** DCA/FEEC/UNICAMP, Campinas (SP), 28 jun./2000. Disponível em: https://www.dca.fee.unicamp.br/cursos/PooJava/polimorf/index.html. Acesso em: 29 maio 2022.

TURINI, R. **Desbravando Java e Orientação a Objetos:** Um guia para o iniciante da linguagem Front Cover. São Paulo (SP): Editora Casa do Código, 2014.

Gabriel Augustin

Experiência em desenvolvimento de aplicações Web com Java e Spring. Atualmente lidera equipes de desenvolvimento *back-end* e *front-end*, em projetos com foco em *gateways* de pagamento e criptomoedas. Atualmente é Tech Leader na KahshPay e Mentor Educacional no LAB365 do SENAI/SC.

SENAI <LAB365>