INTERFACES

SENAI <LAB365>

SUMÁRIO

APRESENTAÇÃO

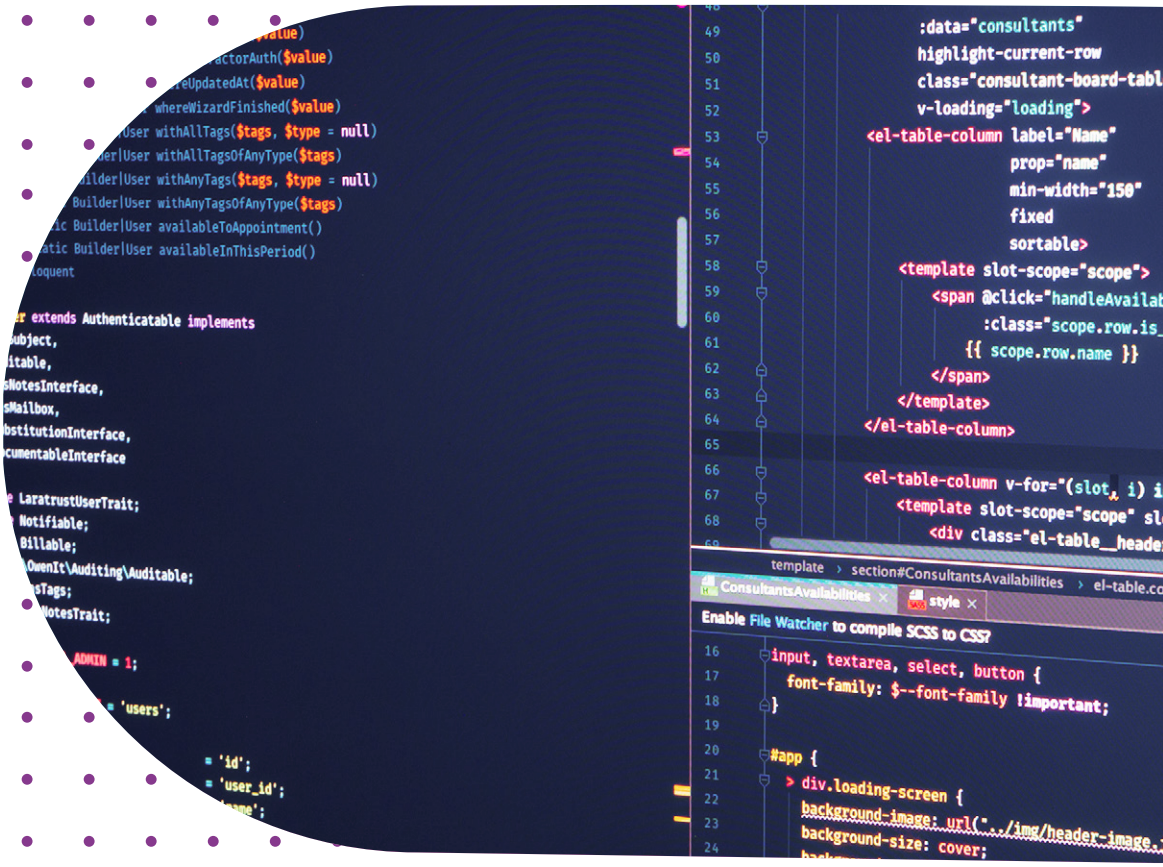
Em capítulos anteriores, aprendemos sobre a herança, que é um dos principais conceitos da orientação a objetos, o qual você utilizará muito em seu dia a dia como desenvolvedor. Aprendemos que uma classe pode herdar atributos e métodos de uma outra classe através da palavra reservada *extends*. Abaixo, segue um exemplo para revisarmos o conhecimento:

```
public class Animal {
	private String cor;
	private String nomeCientifico;
	private String raca;
	private int idade;

	public Animal() {

	}

	public Animal(String cor, String nomeCientifico, String raca,
	int idade) {
		this.cor = cor;
		this.nomeCientifico = nomeCientifico;
		this.raca = raca;
		this.idade = idade;
	}

	public String getCor() {
		return cor;
	}

	public String getNomeCientifico() {
		return nomeCientifico;
	}

	public String getRaca() {
		return raca;
	}

	public int getIdade() {
		return idade;
	}
	//setters foram omitidos
```

```
        public void come() {
                System.out.println("Estou comendo minha comida");
        }

        public void comunica() {
                System.out.println("Estou me comunicando");
        }

}

public class Cachorro extends Animal {

        public Cachorro(String cor, String nomeCientifico, String raca,
        int idade) {
        super(cor, nomeCientifico, raca, idade);
        }

        @Override
        public void comunica() {
                System.out.println("Au au");
        }

}
```

Com este código, podemos observar que a classe “Cachorro” herda da classe “Animal”, portanto, pode-se dizer que um cachorro é um animal.

Pode-se observar que “Cachorro” sobrescreve o método “comunica”, indicando que o cachorro possui uma forma peculiar de se comunicar através de seu latido.



Também, vimos o conceito de classe abstrata. Uma classe abstrata é aquela que não podemos instanciar através do operador new. Dessa maneira, não se pode criar objetos de uma classe abstrata. Observe, abaixo, a classe animal sendo reescrita como uma classe abstrata.

```
public abstract class Animal {
	private String cor;
	private String nomeCientifico;
	private String raca;
	private int idade;

	public Animal() {

	}
```

```
public Animal(String cor, String nomeCientifico, String raca,
int idade) {
      this.cor = cor;
      this.nomeCientifico = nomeCientifico;
      this.raca = raca;
      this.idade = idade;
}

public String getCor() {
      return cor;
}

public String getNomeCientifico() {
      return nomeCientifico;
}

public String getRaca() {
      return raca;
}

public int getIdade() {
      return idade;
}

//setters foram omitidos

public void come() {
      System.out.println("Estou comendo minha comida");
}

public abstract void comunica();

}
```

Neste momento, não podemos criar objetos do tipo “Animal”, mas podemos criar objetos filhos de “Animal”, tais como “Cachorro”, “Gato” etc. Além disso, a classe “Animal” tem um método abstrato que deve ser implementado em cada classe filha. Trata-se do método “comunica”. Se esse método não for sobrescrito e implementado, ocorrerá um erro de compilação e o programa não funcionará.

Vamos imaginar, agora, que queremos separar as categorias de animais em terrestres, aquáticos e voadores. Se utilizássemos herança, precisaríamos criar classes intermediárias, tais como “AnimalTerrestre”, “AnimalAquatico” e “AnimalVoador”. Cada uma dessas classes herdaria de “Animal” e teria seus próprios atributos e métodos específicos também. Assim sendo, a classe “Cachorro” herdaria de “AnimalTerrestre”, a classe “Peixe” de “AnimalAquatico” e a classe “Abelha” de “AnimalVoador”. Provavelmente, as classes AnimalTerrestre, “AnimalAquatico” e “AnimalVoador” seriam abstratas. Criaríamos, então, instâncias de seus filhos, tais como “new Peixe()”, “new Abelha()” ou “new Tigre()”.

Agora, observe a classe “Sapo”: O sapo pode viver tanto na terra quanto na água. A classe “Sapo” deveria herdar de “AnimalTerrestre” ou “AnimalAquatico”?

Se a classe “Sapo” herdasse de “AnimalTerrestre”, descartaríamos suas características de animal aquático. O inverso também é verdadeiro, ou seja, se a classe “Sapo” herdasse de AnimalAquatico, descartaríamos suas características terrestres.

A herança não é capaz de resolver esse impasse, uma vez que uma classe só pode herdar de uma única classe mãe.

Outras linguagens de programação, tais como Python e C++, suportam o conceito de herança múltipla. Dessa forma, um sapo poderia ser um animal terrestre e aquático ao mesmo tempo, pois o sapo poderia herdar de “AnimalTerrestre” e de “AnimalAquatico” simultaneamente. No Java, isso não é possível. Então, precisamos lançar mão de um outro recurso muito poderoso da orientação a objetos, que é o escopo deste capítulo. A esse recurso, damos o nome de interfaces.

O que são interfaces?

Interfaces são especificações de contratos que nossas classes devem seguir, ou seja, quando uma classe implementa uma interface, a classe deve implementar os métodos especificados na interface. Uma classe pode implementar quantas interfaces forem necessárias.

Com isso em mente, não precisamos criar as classes abstratas “AnimalTerrestre”, “AnimalAquatico” e “AnimalVoador”. Podemos, em vez disso, criar interfaces com seus respectivos métodos a serem implementados. Abaixo está a sintaxe de criação da interface “Aquatico”:

```
public interface Aquatico {
  void nada();
}
```

Com esta interface, podemos manter a classe “Sapo” e a classe “Peixe” herdando da classe “Animal”. Tanto “Peixe” quanto “Sapo” implementariam a interface “Aquatico”, como mostra o código a seguir:

```
public class Sapo extends Animal implements Aquatico {

	public Sapo(String cor, String nomeCientifico, String raca,
	int idade) {
		super(cor, nomeCientifico, raca, idade);
	}

	@Override
	public void comunica() {
		System.out.println(“croac croac croac”);
	}

	@Override
	public void nada() {
	//código que representa o nadar do sapo
	}

}
```

Ao implementar a interface “Aquatico”, a classe “Sapo” teve que implementar o método “anda”, que, neste material, foi omitida a sua implementação.

Agora, na sequência, segue o código da interface “Terrestre”:

```
public interface Terrestre {

    void anda();
    void corre();

}
```

Agora, vamos fazer a classe “Sapo” implementar a interface “Terrestre”, além da interface “Aquatico”. Segue abaixo o código da classe reescrita:

```
public class Sapo extends Animal implements Aquatico, Terrestre {

      public Sapo(String cor, String nomeCientifico, String raca,
      int idade) {
         super(cor, nomeCientifico, raca, idade);
      }

      @Override
      public void comunica() {
         System.out.println(“croac croac croac”);
      }

      @Override
      public void nada() {
      //código que representa o nadar do sapo
      }

      @Override
      public void anda() {
      //código que representa o comportamento do andar do sapo
      }

      @Override
      public void corre() {
      //Código que representa o comportamento de correr do sapo
      }
}
```

Como se pode observar, a classe “Sapo” pôde implementar tanto a interface “Aquatico” quanto a interface “Terrestre”, sem que isso cause erros de compilação em nosso programa. Se tivéssemos outros objetos representando animais com comportamentos aquáticos e terrestres, ambos implementariam as duas interfaces separadas por vírgula, como observamos no código da classe “Sapo” acima. Para exemplificar essa possibilidade, vamos observar o código da classe “Pato”, conforme descrito abaixo:

```
public class Pato extends Animal implements Aquatico, Terrestre {

      public Pato(String cor, String nomeCientifico, String raca,
      int idade) {
         super(cor, nomeCientifico, raca, idade);
      }

      @Override
      public void comunica() {
         System.out.println(“quá quá quá”);
      }

      @Override
      public void nada() {
      //código que representa o nadar do pato
      }

      @Override
      public void anda() {
      //código que representa o comportamento do andar do pato
      }

      @Override
      public void corre() {
      //Código que representa o comportamento de correr do pato
      }

}
```

Agora, vamos implementar a interface “Voador”, conforme o código abaixo:

```
public interface Voador {

    void voa();

}
```

Agora, podemos ter animais que, ao implementarem essa interface, saberão voar através do método “voa”. Dessa maneira, podemos ter borboletas, águias, abelhas... que podem voar através da implementação da interface citada acima. Vejamos alguns exemplos abaixo:

```
public class Aguia extends Animal implements Voador, Terrestre {

      public Aguia(String cor, String nomeCientifico, String raca,
      int idade) {
         super(cor, nomeCientifico, raca, idade);
      }

      @Override
      public void comunica() {
         System.out.println(“Grito da águia”);
      }

      @Override
      public void voa() {
      //código que representa o comportamento de voar da águia
      }

      @Override
      public void anda() {
      //código que representa o comportamento do andar da águia
      }

      @Override
      public void corre() {
      //Código que representa o comportamento de correr da águia
      }
}
```

```java
public class Borboleta extends Animal implements Voador {

      public Borboleta(String cor, String nomeCientifico, String raca,
      int idade) {
         super(cor, nomeCientifico, raca, idade);
      }

      @Override
      public void comunica() {
         System.out.println("Comunicação da borboleta");
      }

      @Override
      public void voa() {
      //código que representa o comportamento de voar da águia
      }

}
```

Vantagens do uso de interfaces

Vimos, anteriormente, que as interfaces foram úteis para separar os animais em categorias, tais como animais terrestres, voadores e aquáticos. Observamos que a herança não poderia ser utilizada, pois alguns animais poderiam pertencer a mais de uma categoria ao mesmo tempo, o que não seria suportado com a herança. Além disso, interfaces também podem ser utilizadas para se adquirir polimorfismo, tal qual acontece com a herança. Veja a problemática do zoológico abaixo:

Um zoológico possui diferentes animais. O estabelecimento nos contratou para fazer um programa em Java que, dada uma lista de animais, deveria contar quantos animais existem nessa lista. A lista de animais representa os animais que vivem no zoológico. Esse programa terá duas classes, sendo que uma contará os animais e a outra representará o nosso programa principal com o método "main". Segue abaixo o código da classe "ContadorDeAnimais":

```java
public class ContadorDeAnimais {

    private Animal[] animais;

    public ContadorDeAnimais(Animal[] animais) {
        this.animais = animais;
    }

    public int contaOsAnimais() {
        return this.animais.length;
    }

}
```

Como pudemos verificar, a classe “ContadorDeAnimais” possui o atributo “animais”, que é privado, ou seja, só pode ser acessado de dentro da classe. Esse atributo guarda uma referência para um array de animais. Esse array deve ser passado, obrigatoriamente, no construtor da classe durante a criação de um objeto dessa classe através do operador new.

O método “contaOsAnimais” não recebe parâmetros, pois retorna o tamanho do array de animais referenciado pelo atributo privado “animais”.

Agora, vejamos como fica a classe do programa principal, que terá o método “main”:

```
public class Programa {

      public static void main(String [] args) {

         Animal a1 = new Tigre(“vermelho”, “Panthera Tigris”,
         “Sem raça definida”, 5);

         Animal a2 = new Girafa(“azul”, “Girafa”, “Sem raça
         definida”, 5);

         Animal a3 = new Cachorro(“branco”, “Canis Familiaris”,
         “Poodle”, 5);

         Animal[] animais = {a1, a2, a3};

         ContadorDeAnimais contador = new ContadorDeAnimais
         (animais);

         int totalDeAnimais = contador.contaOsAnimais();

         System.out.println(“O total de animais no zoológico é: ” +
         totalDeAnimais);

       }

}
```

Se executarmos o programa do jeito que está, o resultado que veremos no console é: “O total de animais no zoológico é 3”.

Com o passar do tempo, a lista de animais do zoológico foi crescendo, e agora o estabelecimento quer contar os animais separadamente por categoria. Para isso, vamos ter que adicionar três métodos na classe “ContadorDeAnimais”: um método para contar os animais terrestres, outro para contar os animais aquáticos e o último para contar os animais voadores. Devemos fazer dessa forma, pois o método “contaOsAnimais” retorna o tamanho do array de animais genéricos, sem considerar se são terrestres ou aquáticos ou voadores. Além disso, vamos transformar o atributo “animais” em três atributos privados. Cada atributo vai referenciar um array de categorias de animais. Observe a classe “ContadorDeAnimais” reescrita para atender o novo requisito no código abaixo:

```
public class ContadorDeAnimais {

      private Terrestre[] animaisTerrestres;
      private Aquatico[] animaisAquaticos;
      private Voador[] animaisVoadores;

      public ContadorDeAnimais(Terrestre[] animaisTerrestres,
      Aquatico[] animaisAquaticos, Voador[] animaisVoadores) {
         this.animaisTerrestres = animaisTerrestres;
         this.animaisAquaticos = animaisAquaticos;
         this.animaisVoadores = animaisVoadores;
      }

      public int contaOsTerrestres() {
         return this.animaisTerrestres.length;
      }

      public int contaOsAquaticos() {
         return this.animaisAquaticos.length;
      }

      public int contaOsVoadores() {
         return this.animaisVoadores.length;
      }

}
```

Da forma como ficou agora, temos três atributos privados que referenciam três arrays cada. O array de animais terrestres aceita animais que implementam a interface “Terrestre”, tais como tigres, gatos, cachorros, leões, entre outros.

O array de animais aquáticos aceita animais que implementam a interface “Aquatico”, tais como peixe, sapo, baleia, pato, entre outros.

Se as classes “Sapo” e “Pato” implementarem tanto a interface “Terrestre” quanto a interface “Aquatico”, então o sapo e o pato poderão compor a lista de animais terrestres e de animais aquáticos.

Este foi um exemplo claro de que, com interface, podemos usufruir do polimorfismo, como fazíamos com a herança. A diferença básica é que, com a herança, tínhamos o reuso de código, baseado na herança de atributos e métodos de uma classe pai para uma classe filha. Com interfaces, não temos reuso de código. No entanto, cada interface implementada obriga a classe que a implementa a ter uma implementação para os métodos daquela interface. Dessa forma, um objeto criado por uma classe que implementa uma interface pode ser referenciado por uma variável de referência do tipo da interface. Veja o exemplo abaixo:

```
Voador animalVoador = new Borboleta(“amarela”, “Borboleta”, “sem raça definida”, 1);
```

Observe que criamos um objeto do tipo Borboleta, mas esse objeto é referenciado pela variável de referência chamada animalVoador. Essa variável é do tipo Voador, ou seja, o tipo da referência é diferente do tipo do objeto referenciado, o que caracteriza polimorfismo.

Como a classe “Borboleta” implementa a interface “Voador”, uma variável do tipo Voador pode perfeitamente referenciar um objeto do tipo Borboleta. No entanto, não podemos utilizar uma variável de referência do tipo Voador para apontar para um objeto do tipo Tigre, pois este se trata de um animal terrestre, por implementar a interface Terrestre.

Guido José Warken Filho

Experiência com desenvolvimento de aplicações Web com JavaScript (React e Angular) e PHP. Entusiasta e palestrante sobre os conceitos e aplicações de acessibilidade Web para desenvolvedores com limitação visual. Atualmente é Software Developer no Grupo Boticário e Mentor Educacional no LAB365 do SENAI/SC.”

SENAI <LAB365>