Roteiro 8

Neste roteiro trabalharemos com os conceitos de **Herança e Polimorfismo**.

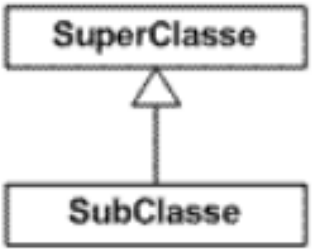
Já aplicamos o Polimorfismo da forma mais simples possível que foi utilizando a Sobrecarga do método construtor, mas o conceito é mais amplo e podemos utilizar de outra forma.

Então vamos ao conceito de **Polimorfismo** - deriva da palavra polimorfo, que significa multiforme, ou que pode variar a forma. Para a POO, polimorfismo é a habilidade de objetos de classes diferentes responderem a mesma mensagem de diferentes maneiras. Ou seja, várias formas de responder à mesma mensagem.

Podemos trabalhar com Polimorfismo de duas maneiras:

- **Sobrecarga (*overload*)** - Lembrando que sobrecarga consiste em permitir que dentro da mesma classe tenhamos métodos com mesmo nome, mas com diferentes parâmetros (Roteiro 5).
- **Sobreposição (*override*)** – Para o uso desta técnica precisamos do conceito de Herança em OO. Esta técnica permite reescrever um método em uma subclasse que possua um comportamento diferente do método de mesmo nome na superclasse.

Herança – é um princípio de OO, que permite que as classes compartilhem atributos e métodos, através de “heranças”. Ela é usada na intenção de reaproveitar código ou comportamento, seja generalizando ou especializando operações e atributos.



Agora iremos aplicar estes conceitos durante o roteiro.

Parte 1 (roteiro8.parte1) – Herança

Cenário inicial :

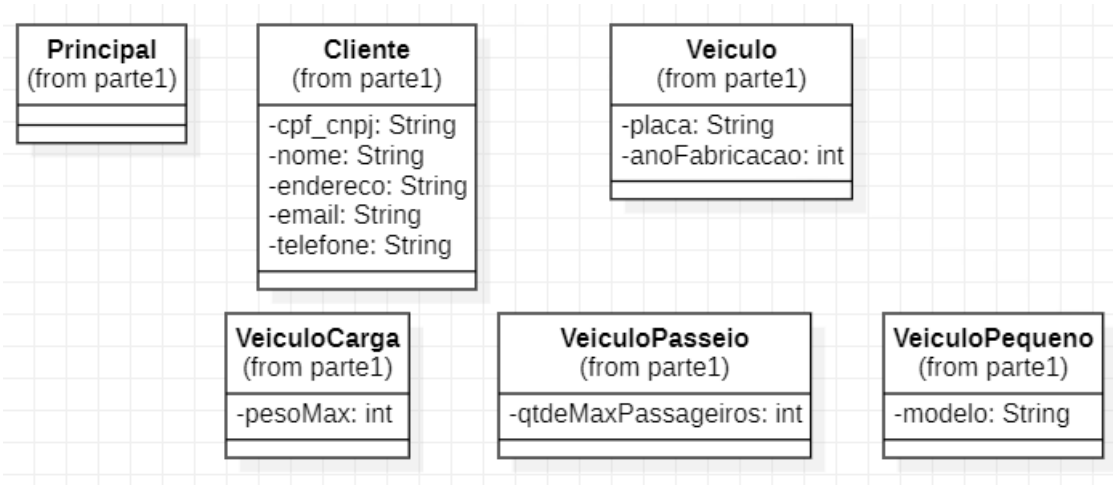
Deseja-se fazer um controle para cobranças de pedágio em uma rodovia. Para isso, neste cenário inicial teremos apenas o cliente com seus atributos (cpf_cnpj, nome, endereço, email, telefone), e o veículo que este cliente possui. O veículo por sua vez, tem como atributo : placa e ano de fabricação. Acontece que existem 3 tipos de veículos e o valor cobrado no pedágio depende do tipo de veículo que o cliente possui :

- Veículos de carga que devem registrar também o peso máximo que podem carregar. O pedágio cobrado é de R$ 2,00 por quilo que pode carregar.
- Veículos de passeio que devem registrar a quantidade máxima de passageiros possíveis. O pedágio cobrado é de R$ 5,00 por pessoa que o veículo comporta.
- Veículos pequenos que devem registrar o modelo. O pedágio cobrado tem taxa única de R$ 6,00.

1 – Crie o pacote **roteiro8.parte1** com a classes **Principal**, **Cliente, Veiculo, VeiculoCarga, VeiculoPasseio, e VeiculoPequeno.**

Observe que estamos criando uma classe para cada tipo de veículo justamente para aplicar o conceito de herança.

Crie as classes com seus atributos de acordo com o que foi indicado no cenário inicial e no diagrama abaixo.



2 – Na classe **Cliente**, crie o construtor com todos os atributos, os seus Gets e Sets.

3 – Na classe **Veiculo**, crie o construtor com todos os atributos, os seus Gets e Sets.

4 – Para as classes **VeiculoCarga, VeiculoPasseio, e VeiculoPequeno** iremos aplicar a **Herança**.

O uso do comando **extends** conforme o código abaixo indica que que estamos aplicando a herança. Ou seja, estas classes irão herdar os atributos e métodos da classe Veiculo.

OBS.: Ao aplicar a herança nestas classes, possivelmente será indicado algum erro, pois a SuperClasse ou classe Pai tem um construtor, e as SubClasses ou classe Filha não tem construtor. O erro indica que obrigatoriamente teremos que criar um método construtor nas classes filha, já eu a classe pai tem um construtor.

Continuação no Item 5

```
public class VeiculoCarga extends Veiculo {

   private int pesoMax;

   {Gets e Sets}

}
```

```
public class VeiculoPasseio extends Veiculo {

   private int qtdeMaxPassageiros;

   {Gets e Sets}

}
```

```
public class VeiculoPequeno extends Veiculo {

   private String modelo;

   {Gets e Sets}

}
```

5 – Crie agora o método construtor para cada uma das classes filha **VeiculoCarga, VeiculoPasseio, e VeiculoPequeno**

Veja que dentro do construtor de cada uma das classes filha existe a chamada do construtor Pai através do comando **super()**

A herança neste caso também é uma forma de garantir que todas as classes filha irão ter obrigatoriamente o construtor da classe Pai. Afinal, não faz sentido criar um Veículo sem as informações de placa e ano de fabricação.

ATENÇÃO : Com estes construtores implementados, estamos garantindo apenas o preenchimento dos atributos da classe Pai (placa e ano de fabricação). Mas existem atributos específicos de cada classe, que não estão sendo preenchidos neste momento.

Continuação no Item 6

```
public class VeiculoCarga extends Veiculo {

   private int pesoMax;

   public VeiculoCarga(String placa, int anoFabricacao) {
         super(placa, anoFabricacao);
   }

   {Gets e Sets}

}
```

```
public class VeiculoPasseio extends Veiculo {

   private int qtdeMaxPassageiros;

   public VeiculoPasseio(String placa, int anoFabricacao) {
         super(placa, anoFabricacao);
   }

   {Gets e Sets}

}
```

```
public class VeiculoPequeno extends Veiculo {

   private String modelo;

   public VeiculoPequeno(String placa, int anoFabricacao) {
         super(placa, anoFabricacao);
   }


   {Gets e Sets}

}
```

6 – Modifique os construtores das classes filha **VeiculoCarga, VeiculoPasseio, e VeiculoPequeno** para que os atributos específicos de cada classe sejam contemplados.

```
public class VeiculoCarga extends Veiculo {

   private int pesoMax;

   public VeiculoCarga(String placa, int anoFabricacao, int pesoMax) {
        super(placa, anoFabricacao);
        this.pesoMax = pesoMax;
   }

   {Gets e Sets}

}
```

```
public class VeiculoPasseio extends Veiculo {

   private int qtdeMaxPassageiros;

   public VeiculoPasseio(String placa, int anoFabricacao, int qtdeMaxPassageiros) {
        super(placa, anoFabricacao);
        this. qtdeMaxPassageiros = qtdeMaxPassageiros;
   }

   {Gets e Sets}

}
```

```
public class VeiculoPequeno extends Veiculo {

   private String modelo;

   public VeiculoPequeno(String placa, int anoFabricacao, String modelo) {
        super(placa, anoFabricacao);
        this.modelo = modelo;
   }


   {Gets e Sets}

}
```

7 – Teste a criação dos objetos dos diferentes tipos de Veículos na classe Principal.

Faça os devidos testes, modificando e exibindo os dados dos veículos criados.

Consegue modificar e exibir tanto dados da classe Pai quanto da classe Filha ?

```
public class Principal {

    public static void main(String[] args) {

      Veiculo v01 = new Veiculo("11111-BA", 2010);
      VeiculoCarga v02 = new VeiculoCarga("22222-BA", 2011, 5000);
      VeiculoPasseio v03 = new VeiculoPasseio("33333-BA", 2012, 5);
      VeiculoPequeno v04 = new VeiculoPequeno("33333-BA", 2012, "moto");

    }
}
```

8 - Para o cenário descrito no roteiro, faz sentido a criação do veículo v01, já que ele não é nenhum dos 3 tipos descrito ? Considera isso uma falha de projeto ?

Se para o projeto esta questão sobre o veículo v01 for crítica, podemos utilizar o conceito de Classe Abstrata.

Classe Abstrata - É um tipo de classe especial que não pode ser instanciada, apenas herdada. Sendo assim, uma classe abstrata não pode ter um objeto criado a partir de sua instanciação. Essas classes são muito importantes quando não queremos criar um objeto a partir de uma classe “geral”, apenas de suas “subclasses”.

Se fizer a mudança abaixo não será mais possível criar o veículo v01,

```
public abstract class Veiculo{

   private String placa;
   private int anoFabricacao;

   {Construtor}


   {Gets e Sets}

}
```

Parte 2 (roteiro8.parte2) – Polimorfismo

1 – Crie o pacote **roteiro8.parte2** com a cópia das classes implementados na parte1.

2 – Vamos agora resolver a questão do cálculo do pedágio descrito no cenário. Considerando que existe uma taxa única no valor de R$ 6.00, mas que é cobrado este valor apenas no caso de veículo pequeno, vamos implementar o polimorfismo de **sobreposição (*override*)** apenas nas classes VeiculoCarga e VeiculoPasseio. O cálculo do pedágio em si ficará na classe Pai (Veículo).

Para isso, crie o atributo taxaPedagio do tipo **protected**. O acesso protected garante que apenas as classes filha podem acessar este atributo. Crie também o método para o cálculo do pedágio conforme o código abaixo.

```
public abstract class Veiculo{

   private String placa;
   private int anoFabricacao;
   protected double taxaPedagio = 6.0;

   {Construtor}

   {Gets e Sets}

   Public double calcPedagio(){
       return this.taxaPedagio;
   }
}
```

3 – Implemente a sobreposição (override) do método calcPedagio nas classes **VeiculoCarga** e **VeiculoPasseio**.

```
public class VeiculoCarga extends Veiculo {

   private int pesoMax;

   {Construtor}

   {Gets e Sets}

    @Override
    public double calcPedagio() {
        super.taxaPedagio = 2.0;
        return super.taxaPedagio * this.pesoMax;
    }
}
```

```
public class VeiculoPasseio extends Veiculo {

   private int qtdeMaxPassageiros;

   {Construtor}

   {Gets e Sets}

    @Override
    public double calcPedagio() {
        super.taxaPedagio = 5.0;
        return super.taxaPedagio * this.qtdeMaxPassageiros;
    }
}
```

4 – Teste a criação dos objetos dos diferentes tipos de Veículos na classe Principal.

Faça os devidos testes, modificando e exibindo os dados dos veículos criados.

O cálculo do pedágio teve o resultado esperado ?

```
public class Principal {

    public static void main(String[] args) {

      VeiculoCarga v02 = new VeiculoCarga("22222-BA", 2011, 5000);
      VeiculoPasseio v03 = new VeiculoPasseio("33333-BA", 2012, 5);
      VeiculoPequeno v04 = new VeiculoPequeno("33333-BA", 2012, "moto");

      System.out.println("Pedágio v02 : " + v02.calcPedagio());
      System.out.println("Pedágio v03 : " + v03.calcPedagio());
      System.out.println("Pedágio v04 : " + v04.calcPedagio());

    }
}
```

Parte 3 (roteiro8.parte3) – Exercício

1 – Crie o pacote **roteiro8.parte3** com a cópia das classes implementados na parte2.

2 – Faça as adaptações necessárias na classe Cliente para guardar a informação de que um cliente possui um veículo.

3 – Na classe Principal crie um cliente qualquer para teste. Set um veículo para este cliente e apresente os dados do cliente, indicando quanto este cliente deve pagar pelo pedágio.
Faça testes “setando” diferentes tipos de veículos para ver o resultado.

4 – Utilize o software StarUML para construir o diagrama de classes fazendo engenharia reversa do código.