



Herança

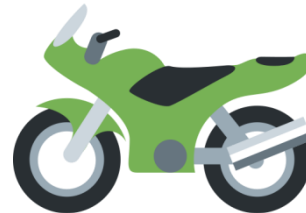
Prof^a. Rachel Reis

rachel@inf.ufpr.br



Problema

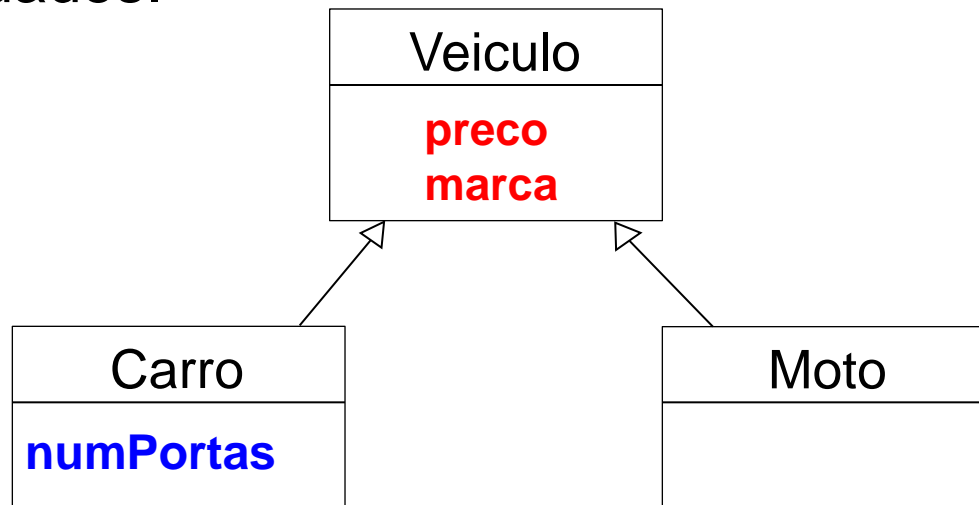
- Considere que uma concessionária tenha diferentes tipos de veículos para venda. Todos os veículos possuem preço e marca.
- Os **carros** possuem também o atributo número de portas.
- As **motos** possuem apenas os atributos preço e marca.





Herança

- É uma forma de reutilização de software em que novas classes são criadas, absorvendo membros de uma classe existente e aprimorada com novas características e funcionalidades.





Representação sem Herança

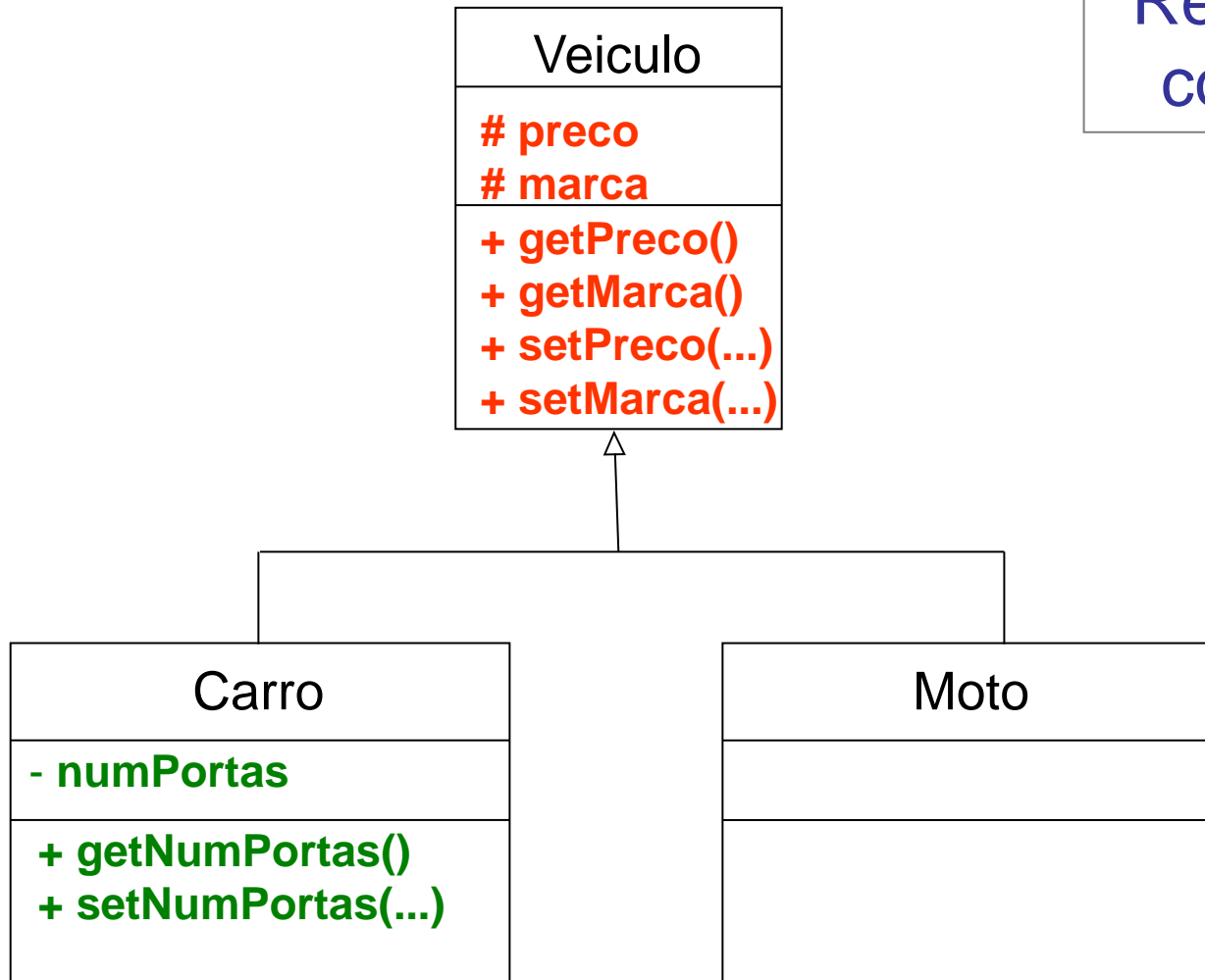
Carro
- preco - marca - numPortas
+ getPreco() + getMarca() + setPreco(...) + setMarca(...) + getNumPortas() + setNumPortas(...)

Veiculo
- preco - marca
+ getPreco() + getMarca() + setPreco(...) + setMarca(...)

Código repetido!

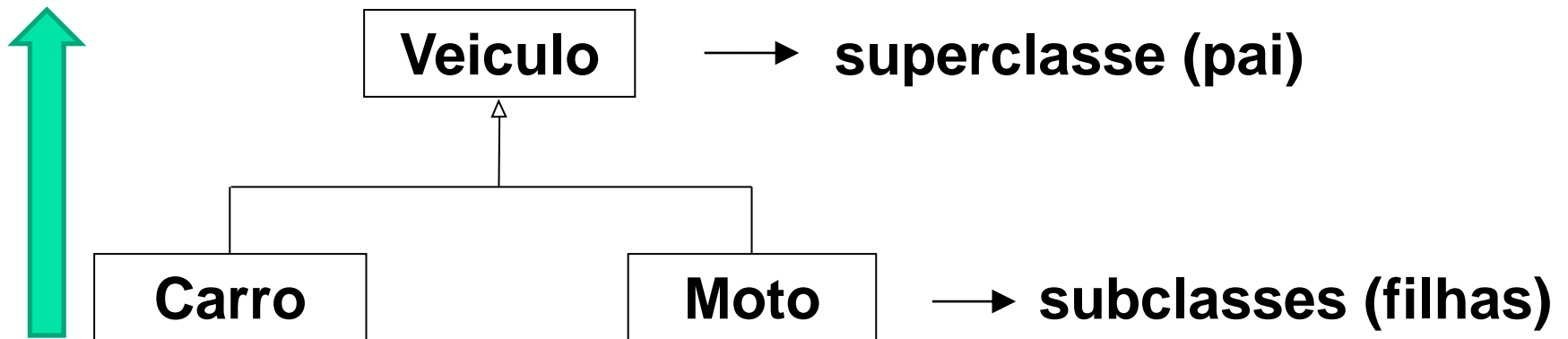
Moto
- preco - marca
+ getPreco() + getMarca() + setPreco(...) + setMarca(...)

Representação com Herança



- Na UML conhecido como Generalização

Exemplo de Herança



- Todos os carros e motos são veículos, mas nem todos os veículos são carros.



Herança

Um subclasse “herda” atributos e métodos de sua superclasse e os utiliza como se fossem declarados dentro da própria classe.





Como implementar Herança?

- Requisitos:
 - 1) Usar a palavra **extends** nas subclasses.


```
public class Veiculo  
{  
    ...  
}
```

```
public class Carro extends Veiculo  
{  
    ...  
}
```

```
public class Moto extends Veiculo  
{  
    ...  
}
```



Como implementar Herança?

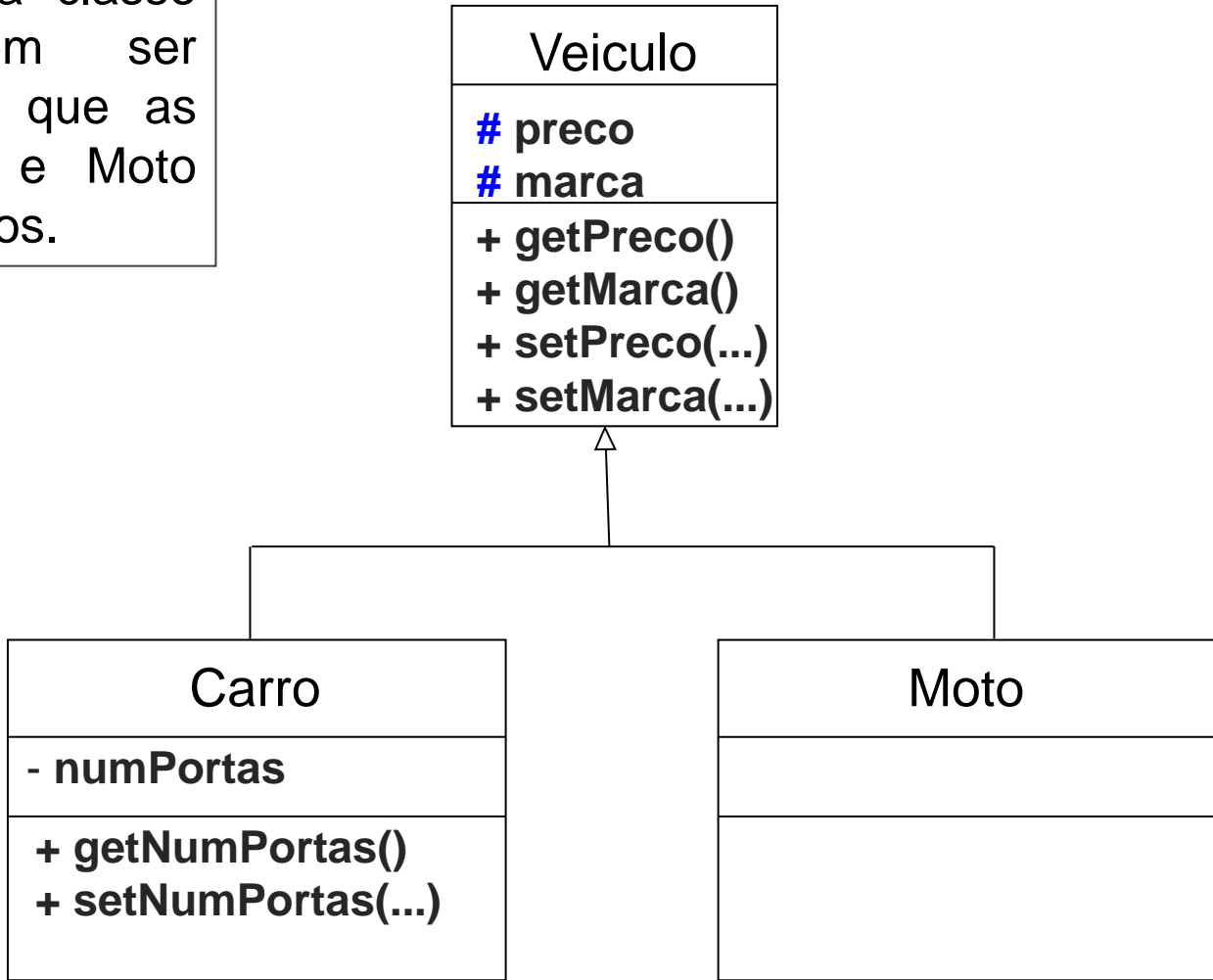
- Requisitos:
 - 1) Usar a palavra `extends` nas subclasses.
 - 2) Declarar os atributos da superclasse como **protected**.



Membros protected

- Os membros *protected* de uma superclasse podem ser acessados:
 - 1) por membros da própria superclasse
 - 2) por membros das subclasses
 - 3) por membros da classe no mesmo pacote

Os atributos da classe Veiculo devem ser *protected* para que as classes Carro e Moto possam herdá-los.



```
public class Veiculo{  
    protected float preco;  
    protected String marca;  
}
```

```
public class Carro extends Veiculo{  
    ...  
}
```

```
public class Moto extends Veiculo{  
    ...  
}
```

```
public class Principal{  
    public static void main(...){  
        ...  
    }  
}
```

```

public class Veiculo{
    protected float preco;
    protected String marca;
    // Métodos get
    public float getPreco() {
        return this.preco;
    }
    public String getMarca() {
        return this.marca;
    }
    // Métodos set
    public void setPreco(float preco) {
        if(preco > 0.0)
            this.preco = preco;
    }
    public void setMarca(String marca) {
        this.marca = marca;
    }
}

```

Veiculo
preco
marca
+ getPreco()
+ getMarca()
+ setPreco(...)
+ setMarca(...)

```
public class Carro extends Veiculo
{
    private int numPortas;

    // Método get
    public int getNumPortas()
    {
        return this.numPortas;
    }

    // Método set
    public void setNumPortas(int numPortas)
    {
        if(numPortas > 0)
            this.numPortas = numPortas;
    }
}
```

Carro
- numPortas
+ getNumPortas() + setNumPortas(...)

```
public class Moto extends Veiculo  
{  
  
}
```

Moto


```
public class Principal
{
    public static void main(String args[]) {

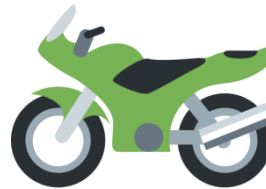
        Carro objeto1 = new Carro();
        Moto objeto2 = new Moto();

    }
}
```



- 0.0
- null
- 0

→ **objeto1**



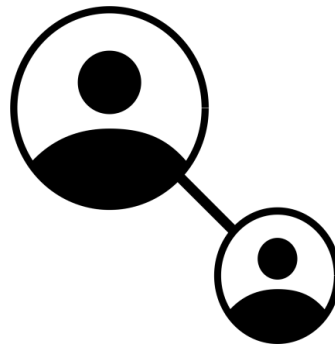
- 0.0
- null

→ **objeto2**



Herança - Resumo

- Herança é o compartilhamento de **atributos** e **métodos** entre classes baseadas num relacionamento hierárquico do tipo “**Pai-Filho**”, ou seja, a classe pai contém definições que podem ser utilizadas nas classes definidas como filho.





Tipos de Herança

- Herança Simples: uma classe é derivada de uma única superclasse
- Herança múltipla: uma classe é derivada de mais de uma superclasse

O Java **NÃO** permite herança múltipla



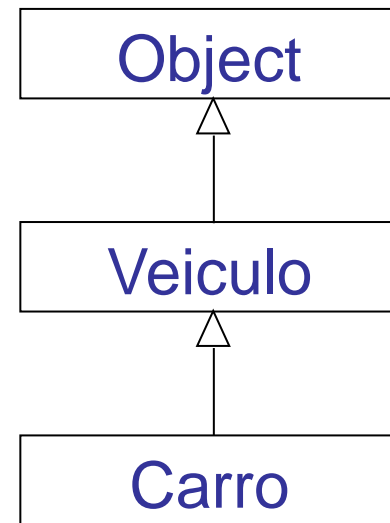
Quando utilizar Herança?

- Regra: realizar a pergunta “É um?”
- Exemplo:
 - Cavalo é um Animal ?
 - Alface é um Vegetal?
 - Gorgonzola é um Queijo?
 - Carro é um Veículo?



Herança em Java

- Em Java todas as classes herdam da classe **Object**.
- Logo,
 - Cavalo é um Object
 - Alface é um Object
 - Gorgonzola é um Object
 - Carro é um Object



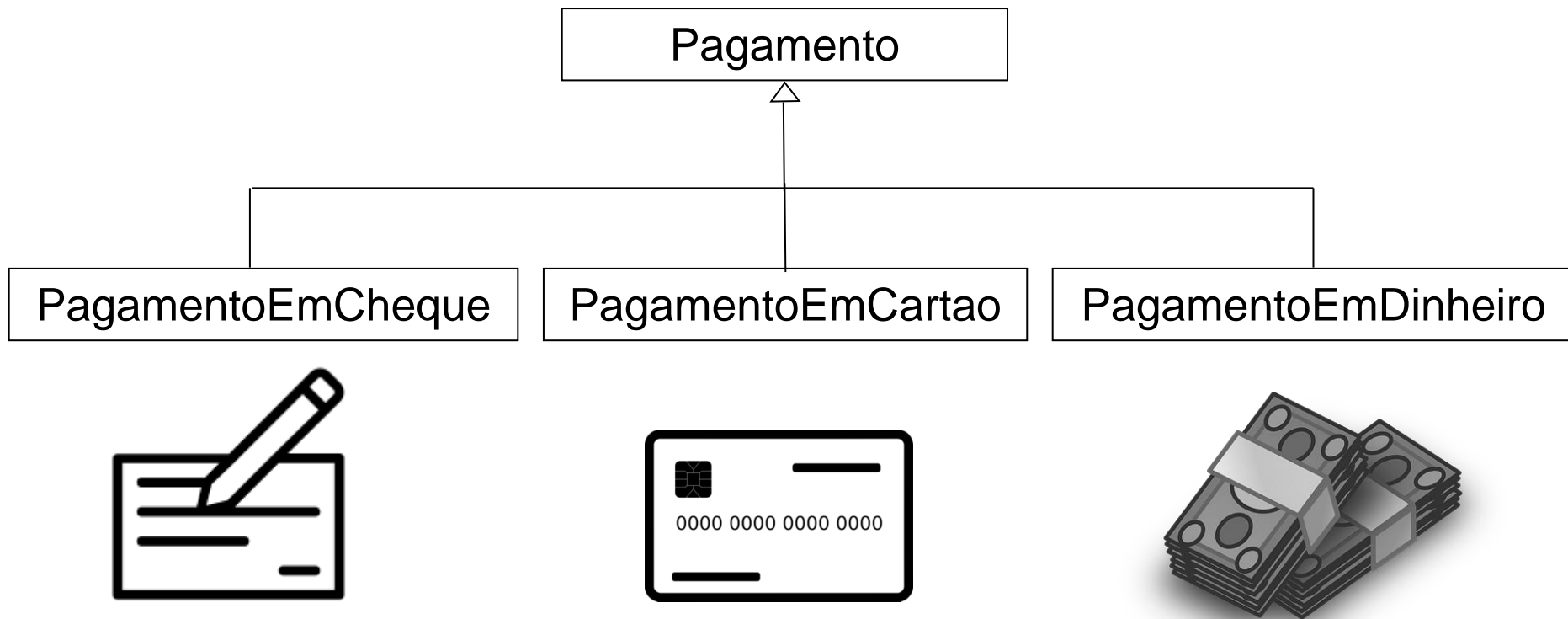


Métodos da classe Object

- Exemplos:
 - `toString()`: retorna a representação do objeto em formato de string.
 - `equals(Object obj)`: compara dois Objects retornando `true` se forem iguais ou `false` se forem diferentes
 - `hashCode()`: retorna um código hash (`int`) para o objeto.

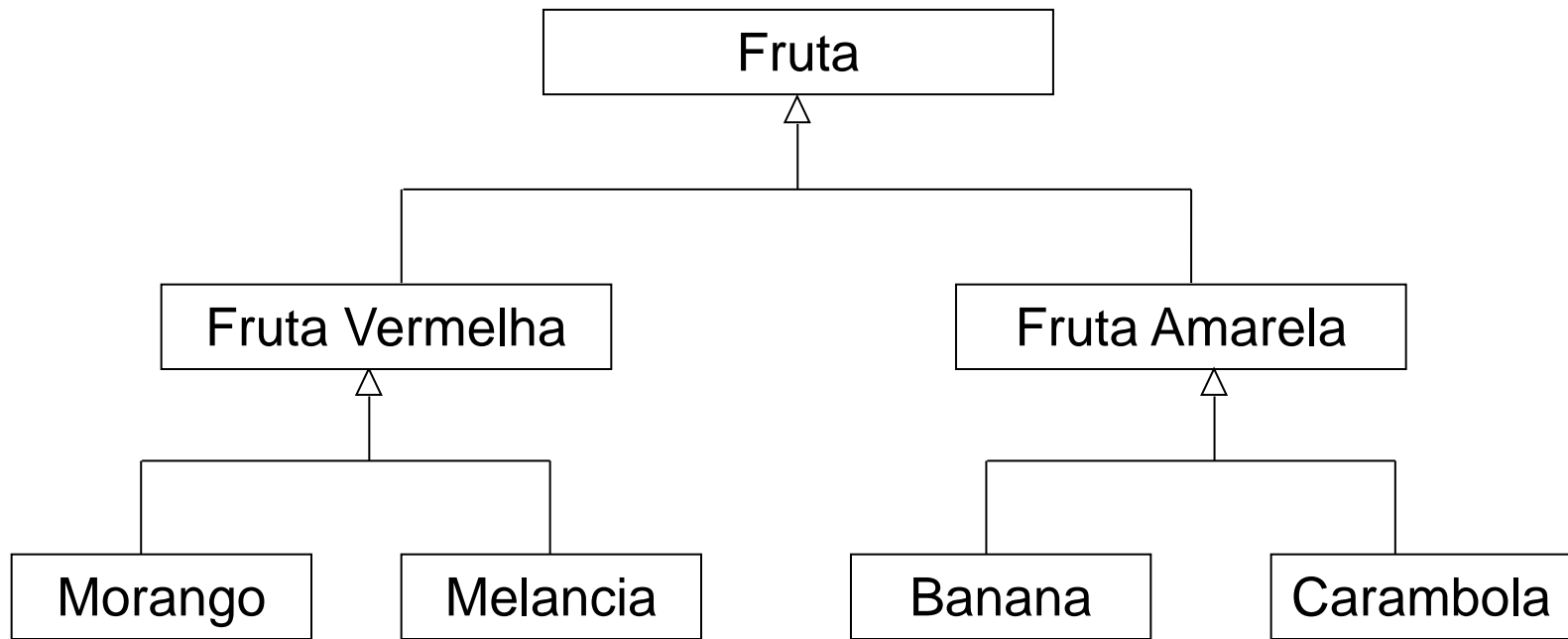


Exemplo de Herança





Exemplo de Herança





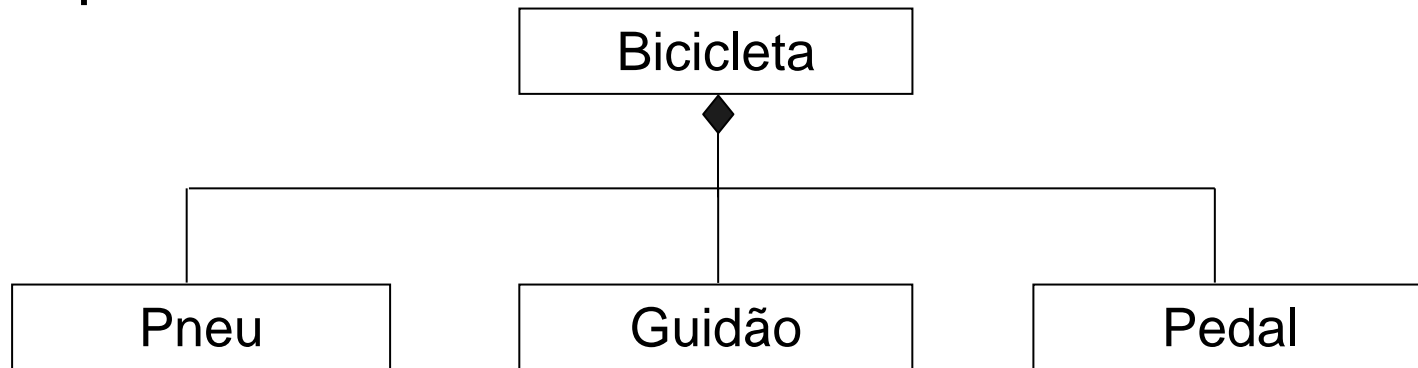
Não é Herança

- Nem todo relacionamento entre classes é de herança.
- Relacionamento “tem um”
 - Na UML conhecido como **composição** (o todo não existe ou não faz sentido sem as partes).



Não é Herança

- Exemplo:



✓ Pneu é uma Bicicleta?

Não

✓ Guidão é uma Bicicleta?

Não

✓ Pedal é uma Bicicleta?

Não



Construtores - Herança

- Construtores são herdados pelas subclasses?



No!



Construtores - Herança

- Instanciar um objeto da subclasse inicia uma cadeia de chamadas de construtores.
- O construtor da subclasse, antes de realizar suas tarefas, invoca o construtor de sua superclasse.
 - Explicitamente (via super)
 - Implicitamente (chamando o construtor-padrão)

superclasse

```
public class Veiculo{  
    protected float preco;  
    protected String marca;
```

```
    public Veiculo(float preco, String marca){  
        ...  
    }
```

```
}
```

Qual o problema?

subclasse

```
public class Carro extends Veiculo{  
    private int numPortas;
```

```
    public Carro(int numPortas, float preco, String marca){  
        ...
```

```
    }
```

```
}
```

```
public class Veiculo{
    protected float preco;
    protected String marca;

    public Veiculo() {}
    public Veiculo(float preco, String marca){
        ...
    }
}
```

```
public class Carro extends Veiculo{
    private int numPortas;

    public Carro(int numPortas, float preco, String marca){
        super();
        ...
    }
}
```

```
public class Veiculo{
    protected float preco;
    protected String marca;

    public Veiculo() {}
    public Veiculo(float preco, String marca){
        ...
    }
}
```

```
public class Carro extends Veiculo{
    private int numPortas;

    public Carro(int numPortas, float preco, String marca){
        // super() está implícito neste construtor
        ...
    }
}
```

```
public class Veiculo{  
    protected float preco;  
    protected String marca;  
  
    public Veiculo(float preco, String marca){  
        ...  
    }  
}
```

```
public class Carro extends Veiculo{  
    private int numPortas;  
  
    public Carro(int numPortas, float preco, String marca){  
        super(preco, marca);  
        ...  
    }  
}
```




Uso do "super" - Construtor

- A palavra chave super é sempre usada pelo construtor das subclasses para chamar o construtor da superclasse.
- O acesso ao construtor da superclasse é útil para aumentar a **reutilização de código**.

```
public class Veiculo{
    protected float preco;
    protected String marca;

    public Veiculo(float preco, String marca){
        this.setPreco(preco);
        this.setMarca(marca);
    }
}
```

```
public class Carro extends Veiculo{
    private int numPortas;

    public Carro(int numPortas, float preco, String marca){
        super(preco, marca);
        this.setNumPortas(numPortas);
    }
}
```



Uso do "super" - Construtor

- A palavra chave super é sempre usada pelo construtor das subclasses para chamar o construtor da superclasse.
- O acesso ao construtor da superclasse é útil para aumentar a reutilização de código.
- Construtores de superclasses só podem ser chamados de dentro de construtores de subclasses (primeira linha de código).

```
public class Veiculo{
    protected float preco;
    protected String marca;

    public Veiculo() {}
    public Veiculo(float preco, String marca) {
        ...
    }
}
```

```
public class Carro extends Veiculo{
    private int numPortas;

    public Carro(int numPortas, float preco, String marca) {
        super();
        ...
    }
}
```

```
public class Veiculo{  
    protected float preco;  
    protected String marca;  
  
    public Veiculo(float preco, String marca){  
  
        ...  
    }  
}
```

```
public class Carro extends Veiculo{  
    private int numPortas;  
  
    public Carro(int numPortas, float preco, String marca){  
        super(preco, marca);  
        ...  
    }  
}
```



Métodos - Herança

- Opções:
 - 1) O método da subclasse pode **sobrepôr** o método da superclasse.



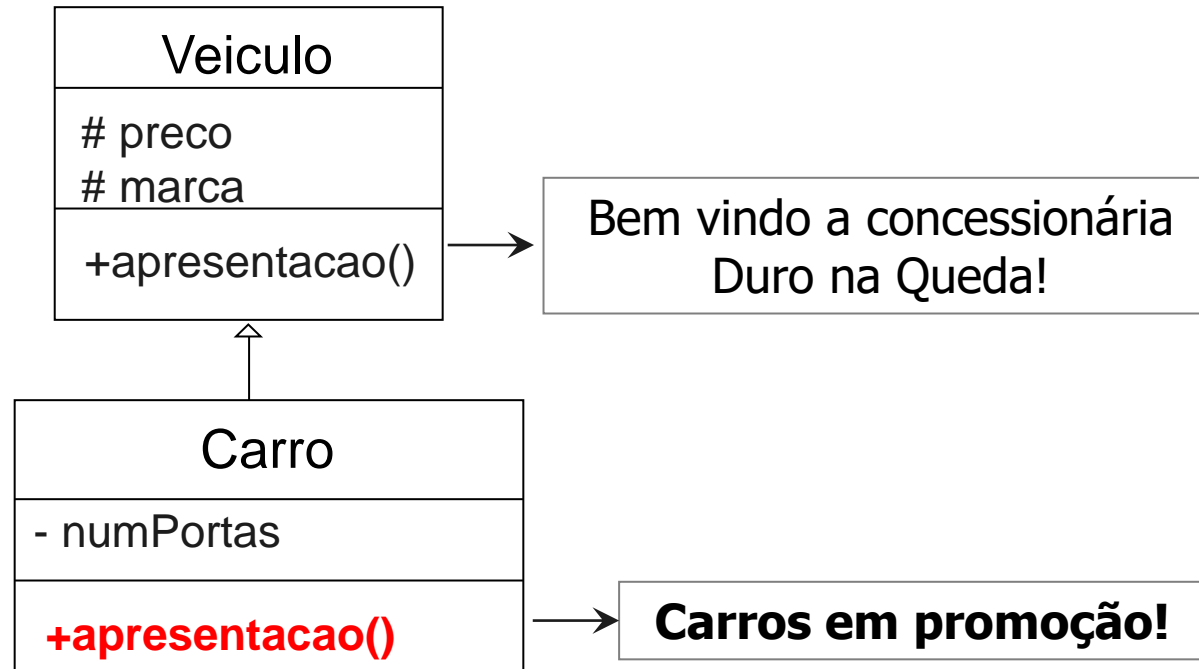
Sobreposição de Métodos

- Um classe filha (subclasse) pode fornecer outra implementação para um método herdado, caracterizando a sobreposição (*overriding*) do método da classe pai.

```
public class Pai
{
    public void metodoX(int p)
    {
        //Código do metodoX da classe Pai
    }
}
```

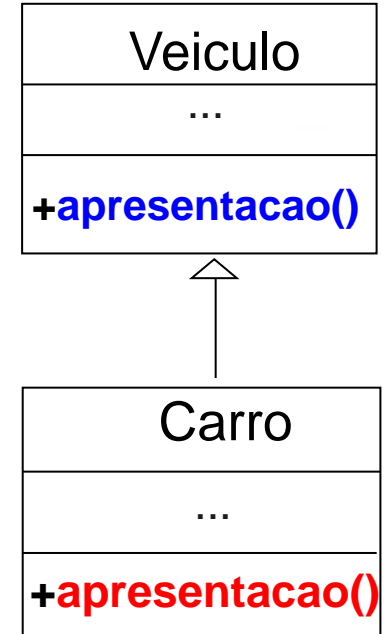
```
public class Filho extends Pai
{
    public void metodoX(int p)
    {
        /*O código do metodoX da classe filho
        sobrepõe o metodoX da classe pai */
    }
}
```


- Exemplo: o método **apresentacao()** da subclasse sobrepõe o método da superclasse.



```
public class Veiculo
{
    public void apresentacao()
    {
        System.out.println("Bem vindo...");
    }
}
```

```
public class Carro extends Veiculo
{
    public void apresentacao()
    {
        System.out.println("Carros em ...");
    }
}
```





Regras - sobreposição de métodos

1. O nome do método **tem** que ser o mesmo.
2. A lista de parâmetros **tem** que ser exatamente a mesma.
3. O tipo de retorno **tem** que ser o mesmo.
4. O nível de acesso (**protected**, **public**, etc) **não pode** ser mais restrito que o do método redefinido.

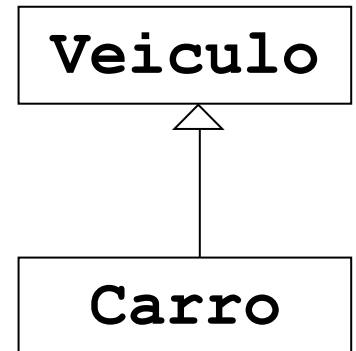


Regras - sobreposição de métodos

1. O nome do método **tem** que ser o mesmo.

```
public void apresentacao()  
{  
    System.out.println("Bem vindo ...");  
}
```

```
public void apresentacao()  
{  
    System.out.println("Carros em ...");  
}
```



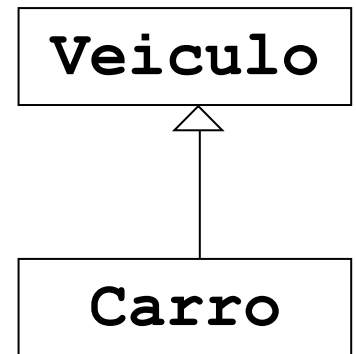


Regras - sobreposição de métodos

2. A lista de parâmetros **tem** que ser exatamente a mesma.

```
public void apresentacao()  
{  
    System.out.println("Bem vindo ...");  
}
```

```
public void apresentacao()  
{  
    System.out.println("Carros em ...");  
}
```



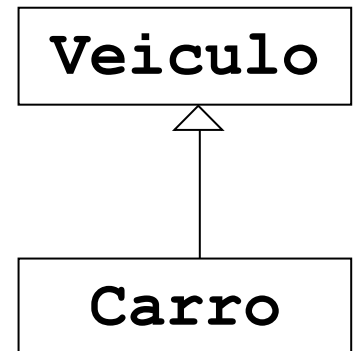


Regras - sobreposição de métodos

3. O tipo de retorno **tem** que ser o mesmo.

```
public void apresentacao()  
{  
    System.out.println("Bem vindo ...");  
}
```

```
public void apresentacao()  
{  
    System.out.println("Carros da ...");  
}
```



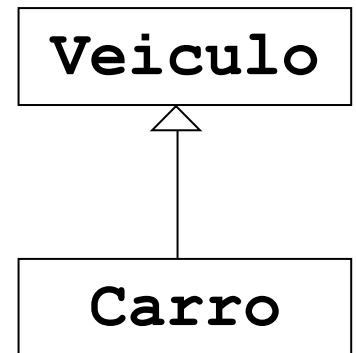


Regras - sobreposição de métodos

4. O modificador de acesso **não pode** ser mais restrito que o do método redefinido.

```
public void apresentacao()  
{  
    System.out.println("Bem vindo ...");  
}
```

```
public void apresentacao()  
{  
    System.out.println("Carros da ...");  
}
```





Métodos - Herança

- Opções:
 - 1) O método da subclasse pode sobrepor o método da superclasse.
 - 2) O método da subclasse pode acessar o método da superclasse usando **super()**



Uso do "super" - Métodos

- Para acessar um método da superclasse que também foi definido na subclasse:

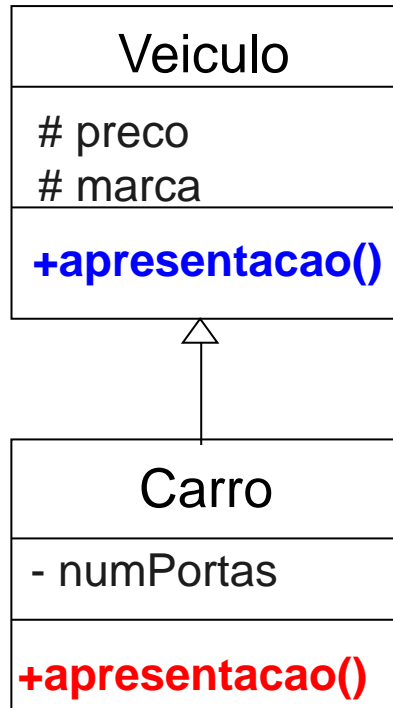
```
super.nomeDoMetodo(lista de parâmetros);
```

```
public class Pai
{
    public void metodoX(int p)
    {
        //Aqui tem um código que queremos manter
    }
}
```

```
public class Filho extends Pai
{
    public void metodoX(int p)
    {
        super.metodoX(p) ;

        //Aqui adiciono mais funcionalidades
    }
}
```

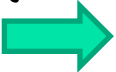
- Exemplo: usar a palavra chave “super” para imprimir as duas mensagens.



**Bem vindo a concessionária
Duro na Queda!!!**

Carros em promoção!

```
public class Veiculo
{
    public void apresentacao()
    {
        System.out.println("Bem vindo ...");
    }
}
```

```
public class Carro extends Veiculo
{
    public void apresentacao()
    {
         super.apresentacao();
        System.out.println("Carros em ...");
    }
}
```



Classe Final

- Uma classe deve ser declarada como **final** quando ela não puder ser herdada por nenhuma outra classe.

```
public final class <nome_da_classe>
{
    ...
}
```

- Qualquer tentativa de criar subclasses resultará em **erro de compilação**.



Métodos Final

- Um método deve ser declarada como **final** quando ela não puder ser sobrescrito.

```
<modificador> <tipo> final <nome>()  
{  
    ...  
}
```

```
public void final mover(int x, int y)  
{  
    ...  
}
```



Referências

- Deitel, P. J.; Deitel, H. M. (2017). Java como programar. 10a edição. São Paulo: Pearson Prentice Hall.
- Barnes, D. J. (2009). Programação orientada a objetos com Java: uma introdução prática usando o BlueJ (4. ed.). São Paulo, SP: Prentice Hall.
- Boratti, I. C. (2007). Programação orientada a objetos em Java. Florianópolis, SC: Visual Books.