Análise e Projeto Orientados por Objetos

Aula 01 – Orientação a Objetos

Edirlei Soares de Lima
<edirlei@iprj.uerj.br>

Paradigmas de Programação

- Um **paradigma de programação** consiste na filosofia adotada na construção de softwares.

- É um **modelo**, **padrão** ou **estilo de programação** suportado por linguagens que agrupam certas características comuns.

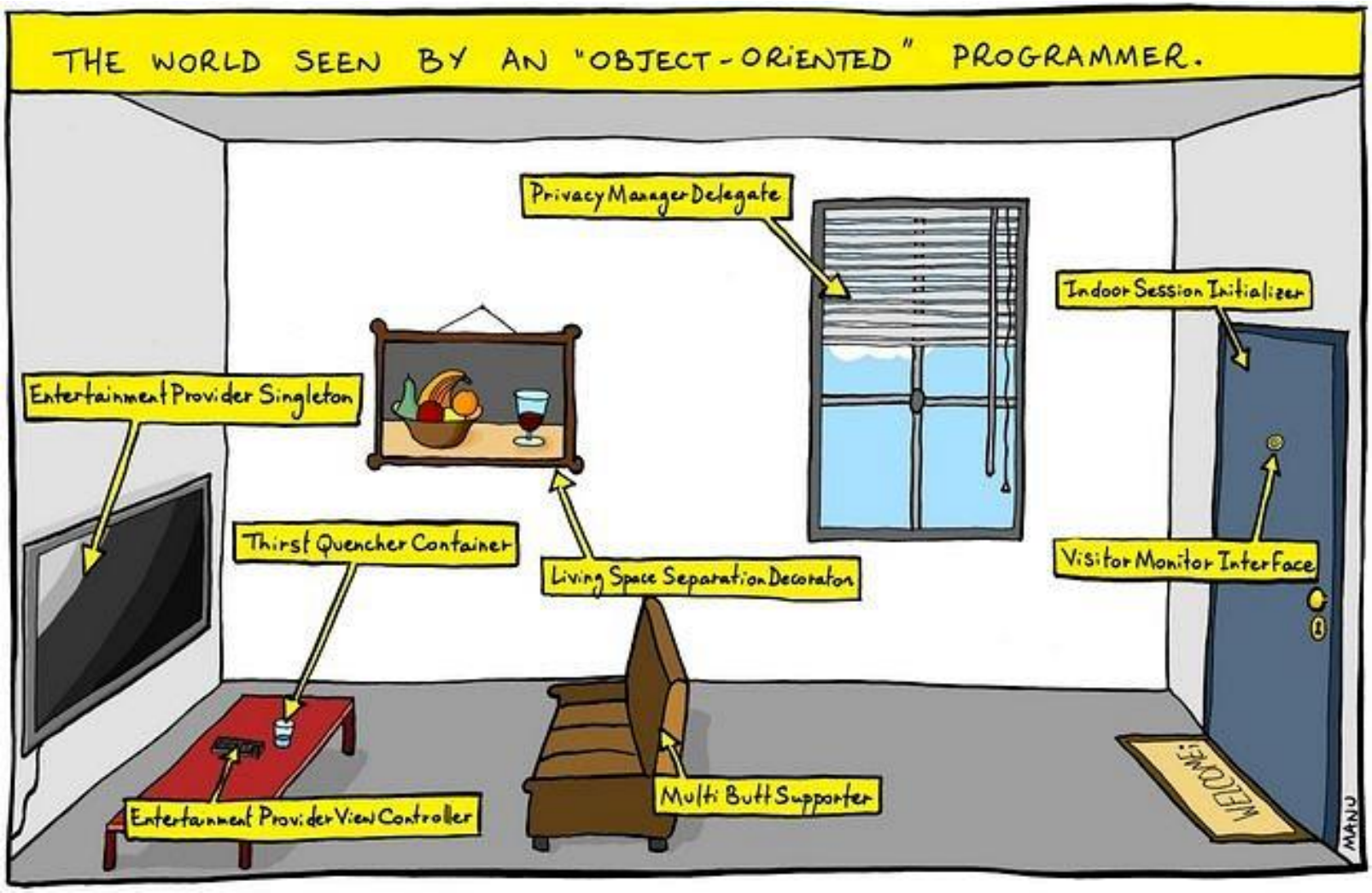
- **Exemplos:**
  - Imperativo e Estruturado (C, Pascal, Basic, ...);
  - Lógico (Prolog, ...);
  - Funcional (Lisp, Haskell, ...);
  - **Orientado a Objetos** (Java, C++, C#, ...);

Orientação a Objetos

- Consiste em um paradigma de **análise**, **projeto** e **programação** de sistemas baseado na composição e interação entre diversas unidades de software chamadas de **objetos**.

- Sugere a diminuição da distância entre a **modelagem computacional** e o **mundo real**:
  - O ser humano se relaciona com o mundo através de conceitos de objetos;
  - Estamos sempre identificando qualquer objeto ao nosso redor;
  - Para isso lhe damos nomes, e de acordo com suas características lhes classificamos em grupos;

Orientação a Objetos



Fonte: http://www.bonkersworld.net/object-world/

Orientação a Objetos

- Surgiu na tentativa de solucionar **problemas complexos** existentes através do desenvolvimento de sistema **menos complexos**, **confiáveis** e com **baixo custo** de desenvolvimento e manutenção.

- **Por que usar a orientação a objetos?**
  - Organização do código;
  - Aumenta a reutilização de código;
  - Reduz tempo de manutenção de código;
  - Reduz complexidade através da melhoria do grau de abstração;
  - Ampla utilização comercial;
    - 4 das 5 linguagens mais populares hoje são totalmente baseadas em orientação a objeto (http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html).

Diagrama de Classes - UML

- **UML (Unified Modeling Language):** linguagem de modelagem que permite representar um sistema de forma padronizada.

- **Diagrama de classes:**
  - Mostra um **conjunto de classes** e seus **relacionamentos**;
  - É o diagrama central da modelagem orientada a objetos;

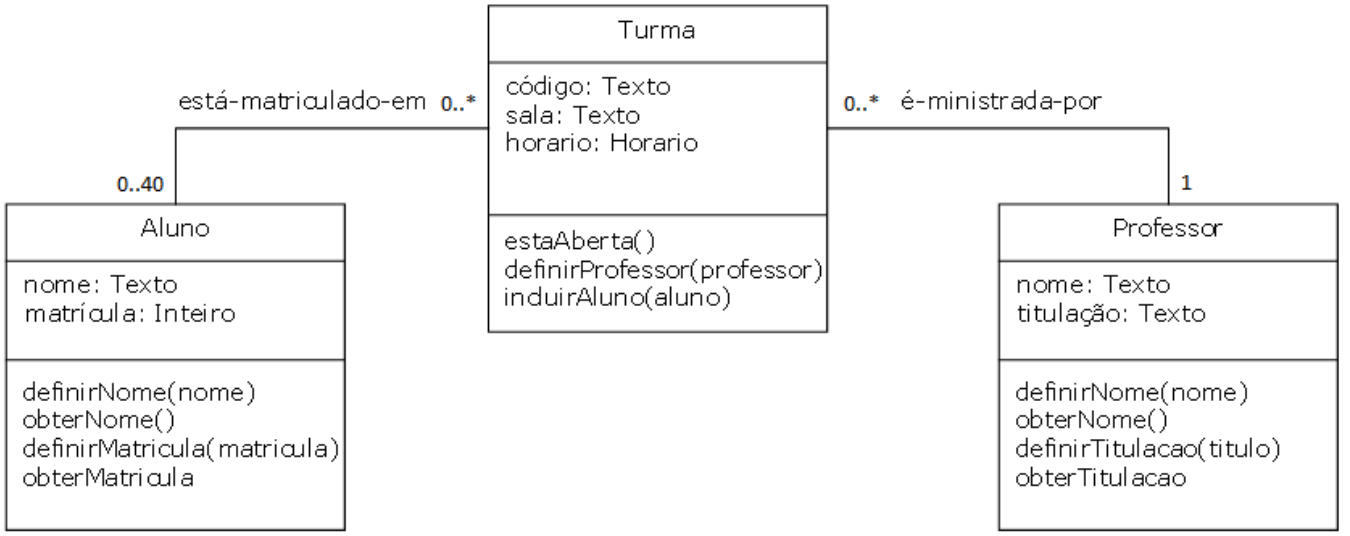
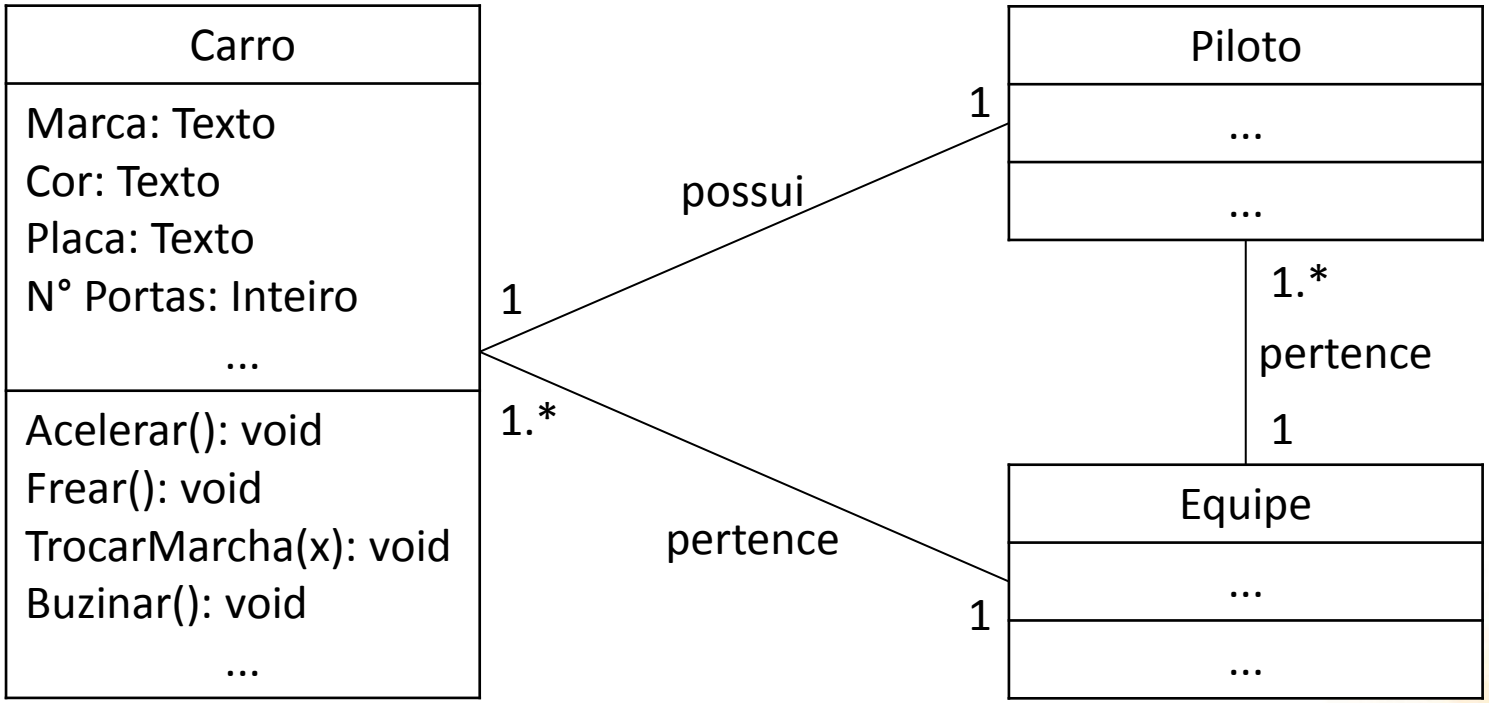


Diagrama de Classes - UML

- Graficamente, as classes são representadas por retângulos incluindo nome, atributos e métodos.



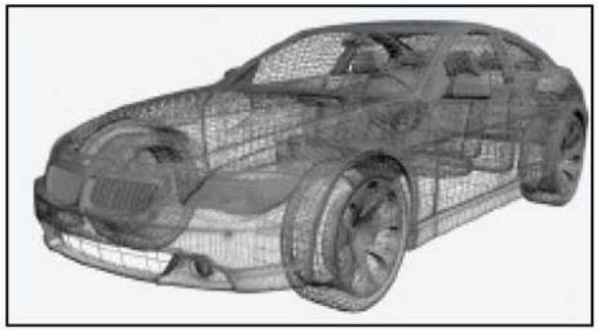
Relacionamento: Associação

Classes e Objetos

- A classe é o **modelo** ou **molde** de construção de objetos. Ela define as características e comportamentos que os objetos irão possuir.

- E sob o ponto de vista da programação:
  - O que é uma classe?
  - O que é um atributo?
  - O que é um método?
  - O que é um objeto?
  - Como o objeto é usado?
  - Como tudo isso é codificado???

Classe Carro



Objeto Carro2

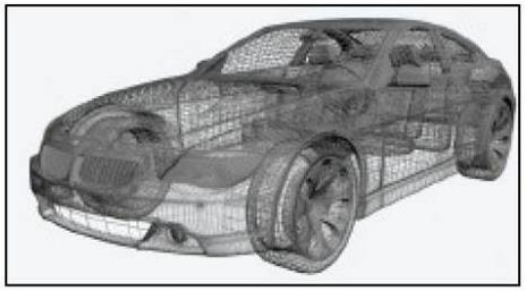
Criação de Classes

```
public class Carro {

    private String marca;
    private String cor;
    private String placa;
    private int portas;
    private int marcha;
    private double velocidade;

    public void Acelerar()
    {
        velocidade += marcha * 10;
    }

    public void Frear()
    {
        velocidade -= marcha * 10;
    }

    ...
}
```

Declaração

Atributos

Métodos



Carro
- Marca: Texto - Cor: Texto - Placa: Texto - N° Portas: Inteiro ...
+ Acelerar(): void + Frear(): void + TrocarMarcha(x): void + Buzinar(): void ...

Utilizando Objetos

- Para manipularmos objetos precisamos declarar **variáveis de objetos.**
- Uma variável de objeto é uma **referência** para um objeto.
- A **declaração de uma variável de objeto** é semelhante a declaração de uma variável normal:

```
Carro carro1;     /* carro1 não referencia nenhum objeto,
                     o seu valor inicial é null */
```

- A simples declaração de uma variável de objeto não é suficiente para a **criação de um objeto**.

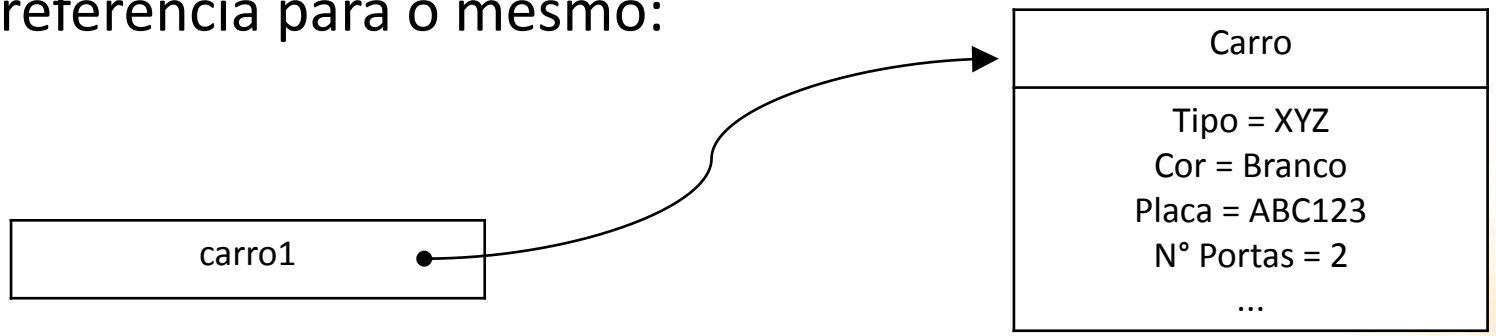
Utilizando Objetos

- A criação de um objeto deve ser explicitamente feita através do operador **new**:

```
Carro carro1;

carro1 = new Carro(); /* instancia o objeto carro1 */
```

Método construtor da classe Carro.

- O operador new **aloca o objeto em memória** e retorna uma referência para o mesmo:



Utilizando Objetos

- Um objeto expõe o seu **comportamento** através dos métodos.

- É possivel **enviar mensagens** para os objetos objetos instanciados:

```
Carro carro1 = new Carro();
Carro carro2 = new Carro();

carro1.mudarMarcha(2);
carro1.aumentaVelocidade(5);

carro2.mudarMarcha(4);
carro2.aumentaVelocidade(10);
```

Envia a mensagem mudarMarcha para o objeto carro1.

Envia a mensagem aumentaVelocidade para o objeto carro2.

Pilares da Orientação a Objetos

- A orientação a objetos está sedimentada sobre quatro pilares derivados do princípio da abstração:
  - Encapsulamento
  - Herança
  - Composição
  - Polimorfismo

Pilares da Orientação a Objetos

- A orientação a objetos está sedimentada sobre quatro pilares derivados do princípio da abstração:
  - **Encapsulamento**
  - Herança
  - Composição
  - Polimorfismo

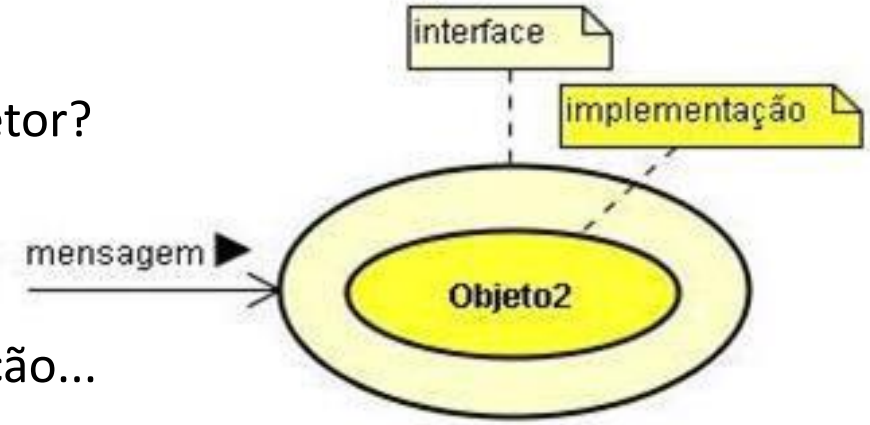
Encapsulamento

- Encapsulamento é a característica da orientação a objetos capaz de **ocultar** partes (dados e detalhes) de implementação interna de classes do mundo exterior.

- Os objetos ficam **protegidos** em uma capsula (interface).

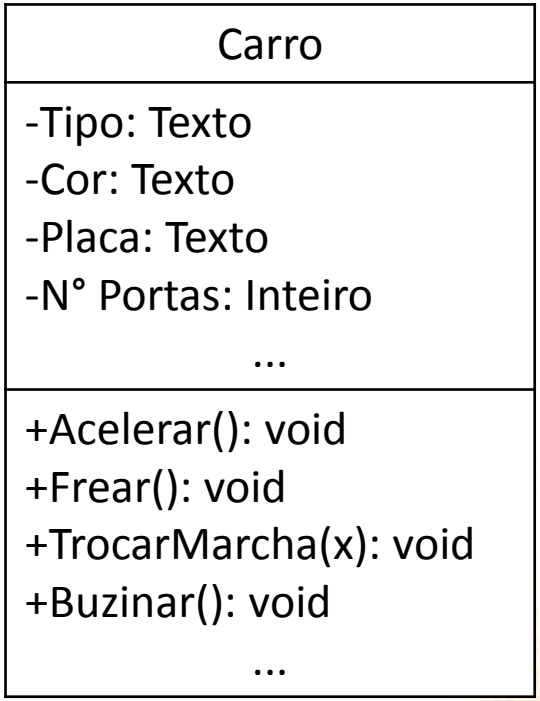
- Evita o acesso indevido e a corrupção de dados.
  - Exemplo: como desligar o projetor?

- Vantagens:
  - Abstração, manutenção, proteção...



Encapsulamento - UML

- **Visibilidade**:
  - **Público (+)**: o atributo ou método pode ser acessado por qualquer outra classe.
  - **Privado (-)**: o atributo ou método pode ser acessado apenas por métodos da mesma classe.
  - **Protegido (#)**: funciona como o private, exceto que as classes filhas ou derivadas também terão acesso ao atributo ou método.



Modificadores de Acesso

```
public class Carro {
    public double velocidade;

    private String cor;


    public void Acelerar()
    {
        velocidade += marcha * 10;
    }

    public void Frear()
    {
        velocidade -= marcha * 10;
    }
}
```

O modificador **public** declara que a classe pode ser usada por qualquer outra classe. Sem public, uma classe pode ser usada somente por classes do mesmo pacote

O modificador **public** declara que o atributo pode ser acessado por métodos externos à classe na qual ele foi definido.

O modificador **private** declara que o atributo não pode ser acessado por métodos externos à classe na qual ele foi definido.

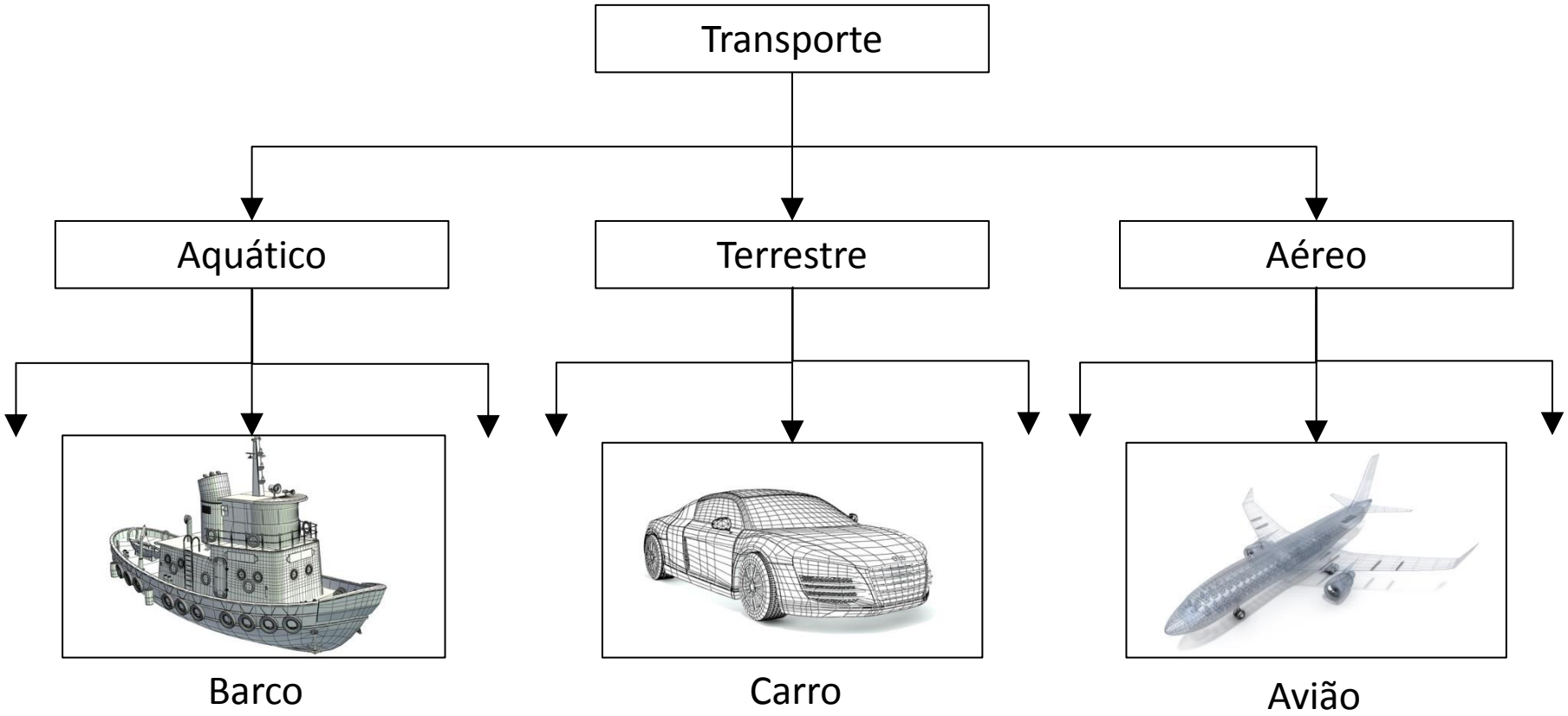
O modificador **public** declara que o método pode ser executado por métodos externos à classe na qual ele foi definido.

Pilares da Orientação a Objetos

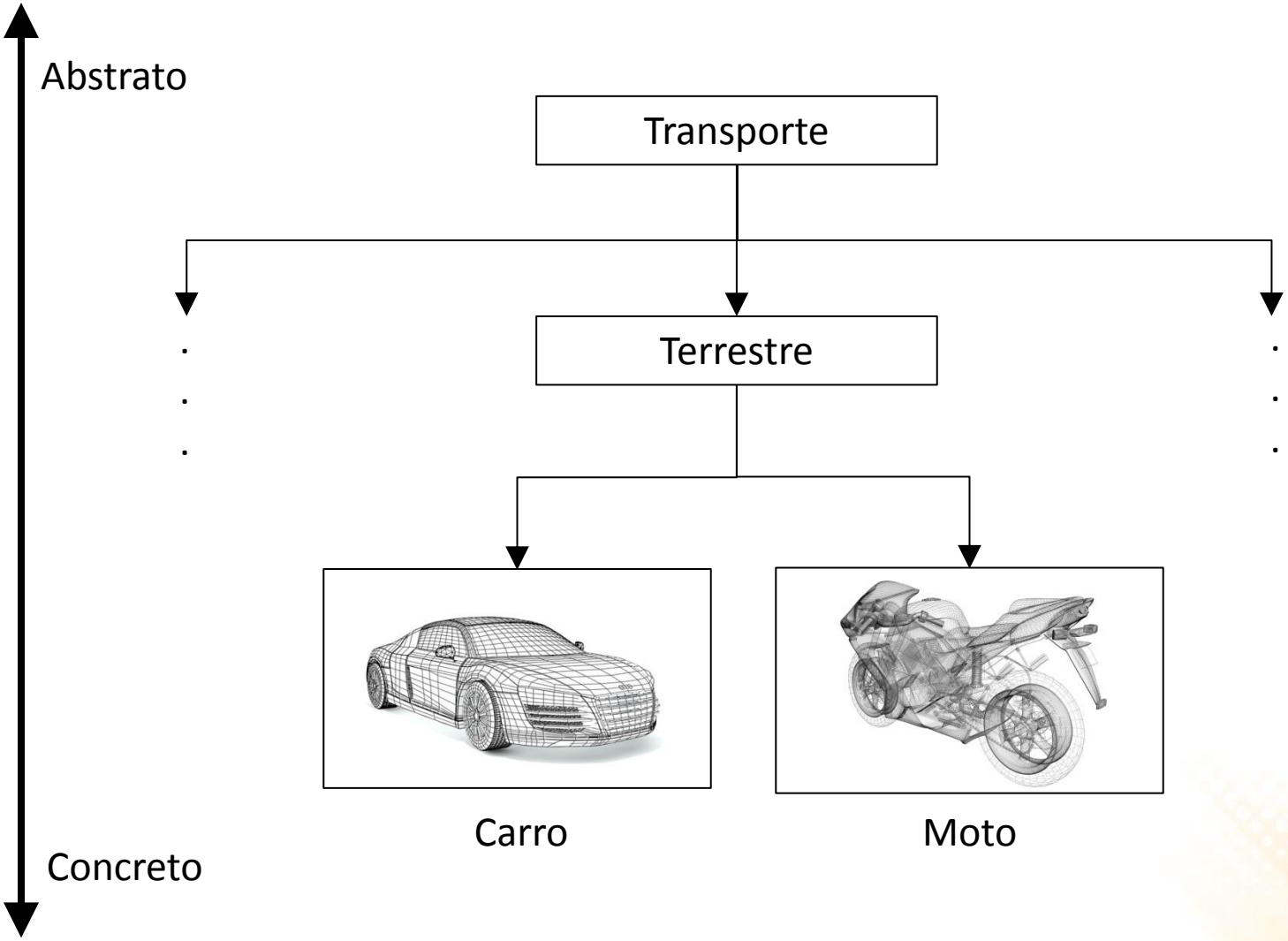
- A orientação a objetos está sedimentada sobre quatro pilares derivados do princípio da abstração:
  - Encapsulamento
  - **Herança**
  - Composição
  - Polimorfismo

Herança

- Herança é o mecanismo pelo qual uma classe pode "**herdar**" as características e métodos de outra classe para expandi-la ou especializá-la de alguma forma.

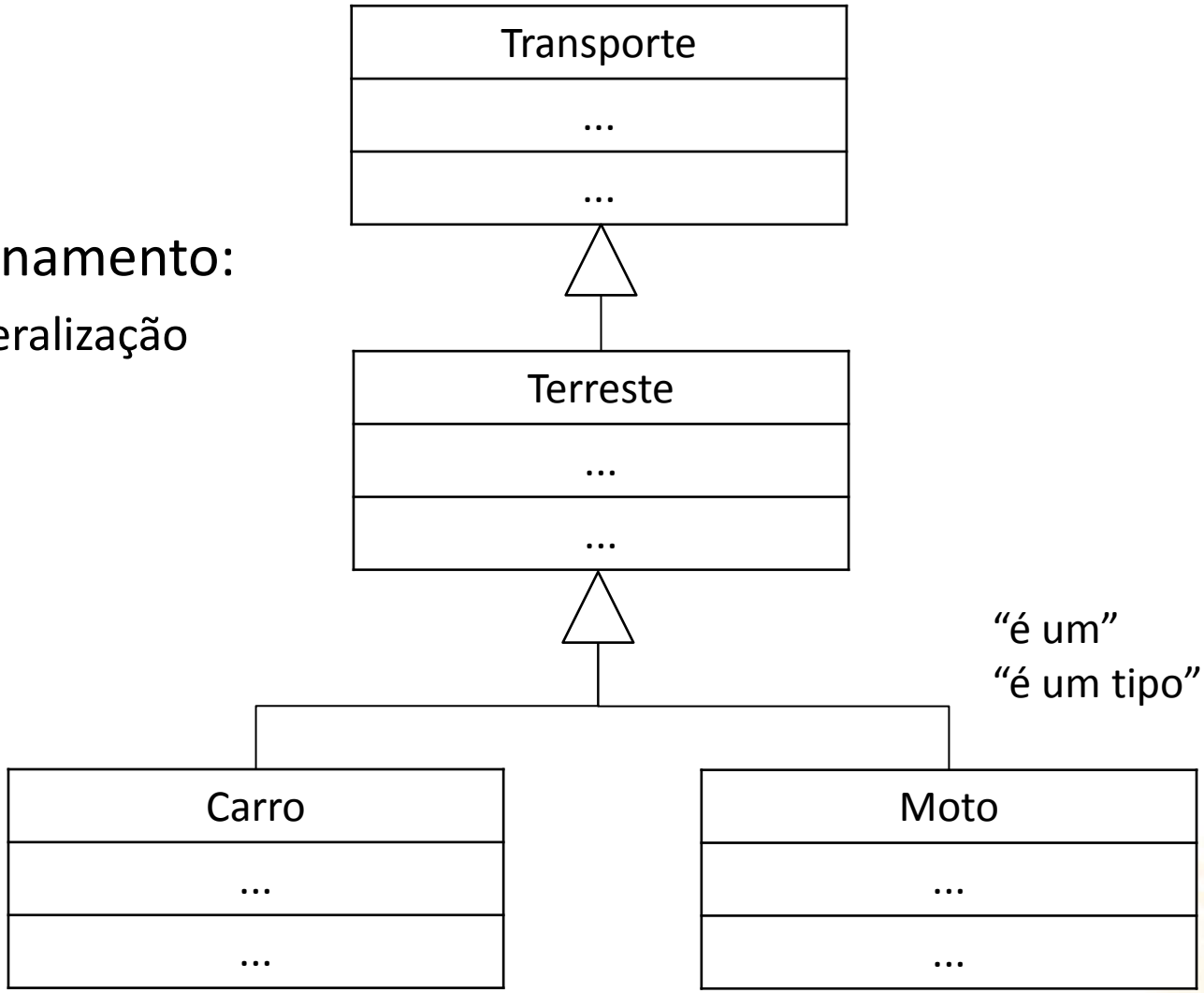


Generalização e Especialização



Herança - UML

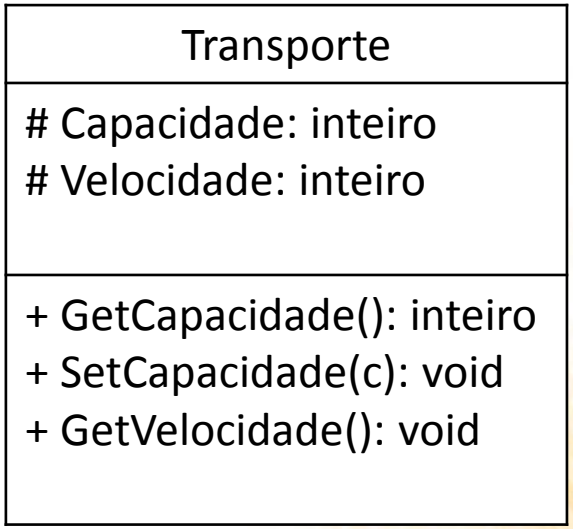
- Relacionamento:
  - Generalização



Herança

```
public class Transporte {
    protected int capacidade;
    protected int velocidade;

    public Transporte()
    {
        velocidade = 0;
    }
    public int GetCapacidade()
    {
        return capacidade;
    }
    public void SetCapacidade(int cap)
    {
        capacidade = cap;
    }
    public int GetVelocidade()
    {
        return velocidade;
    }
}
```

O modificador **protected** determina que apenas a própria classe e as classes filhas poderão ter acesso ao atributo.



Herança

```
public class Terrestre extends Transporte{

    protected int numRodas;

    public Terrestre(int rodas)
    {
        super();
        numRodas = rodas;
    }

    public int GetNumRodas()
    {
        return numRodas;
    }

    public void SetNumRodas(int num)
    {
        numRodas = num;
    }
}
```

A classe Terrestre **herda** as características da classe Transporte.

Chamada ao **método construtor** da classe pai (Transporte).

Terrestre
N° Rodas: inteiro
+ GetNumRodas() : inteiro + SetNumRodas(n): void

Herança

```
public class Carro extends Terrestre{
    private String cor;
    private String placa;
    private int marcha;

    public Carro(String ncor, String nplaca)
    {
        super(4);
        cor = ncor;
        placa = nplaca;
        marcha = 0;
    }
    public int GetMarcha()
    {
        return marcha;
    }
    public void TrocarMarcha(int novaMarcha)
    {
        marcha = novaMarcha;
    }
    ...
```

A classe Carro **herda** as características da classe Terrestre.

Chamada ao **método construtor** da classe pai (Terrestre).

Carro
- Cor: Texto - Placa: Texto - Marcha: Inteiro
+ GetMarcha(): Inteiro + TrocaMarcha(n): void + Acelerar(): void + Frear(): void

Herança

```
    ...


    public void Acelerar()
    {
        velocidade += marcha * 10;
    }

    public void Frear()
    {
        velocidade -= marcha * 10;
    }

}
```

Carro
- Cor: Texto - Placa: Texto - Marcha: Inteiro
+ GetMarcha(): Inteiro + TrocaMarcha(n): void + Acelerar(): void + Frear(): void

Herança

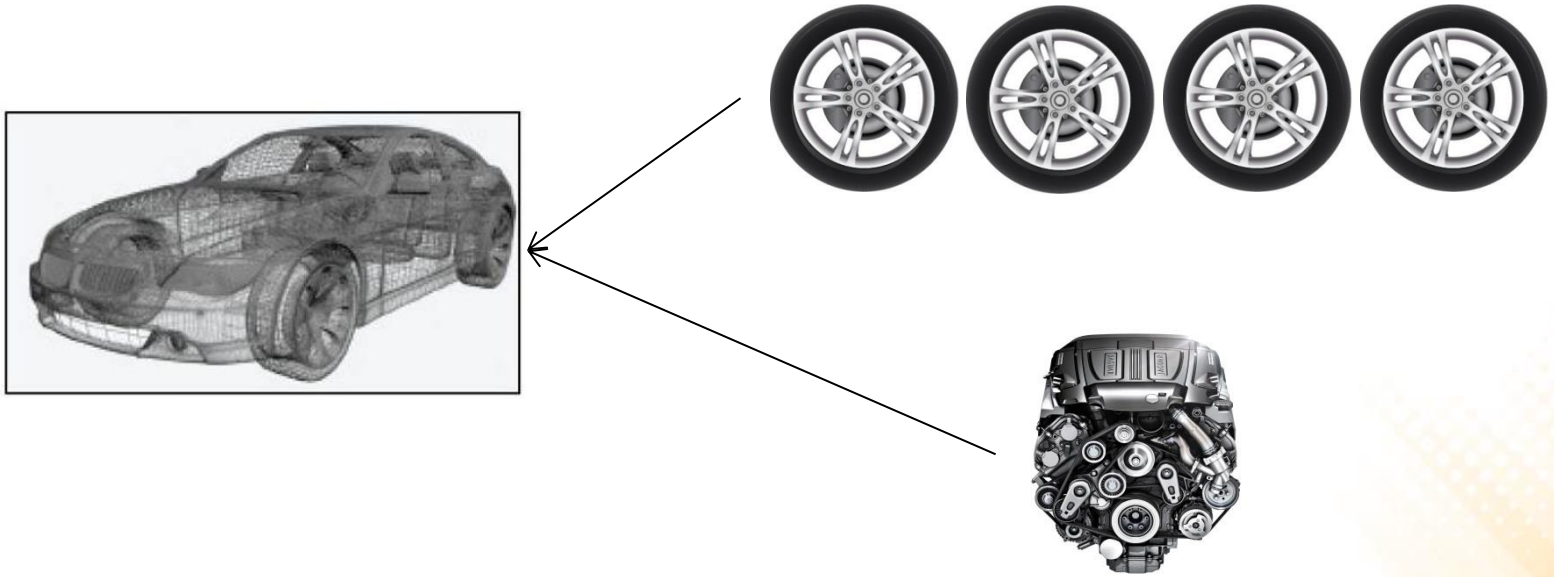
```
public static void main(String[] args) {

  Carro meuCarro = new Carro("Vermelho", "ABC-1234");

  meuCarro.TrocarMarcha(1);
  meuCarro.Acelerar();
  meuCarro.Acelerar();

  System.out.println("Velocidade do Carro: "+meuCarro.GetVelocidade());

}
```

Pilares da Orientação a Objetos

- A orientação a objetos está sedimentada sobre quatro pilares derivados do princípio da abstração:
  - Encapsulamento
  - Herança
  - **Composição**
  - Polimorfismo

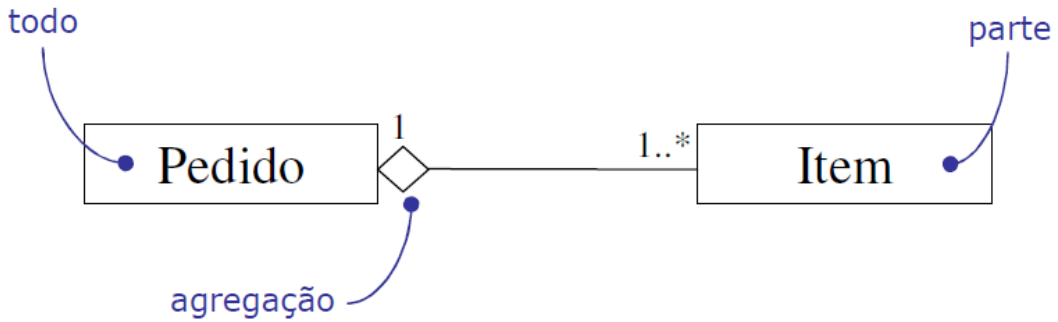
Composição

- A composição (ou agregação) é a característica da orientação a objetos que permite combinar objetos simples em objetos mais complexos.
  - Possibilita a reutilização de código;
  - Um objeto mais complexo pode ser composto de partes mais simples;



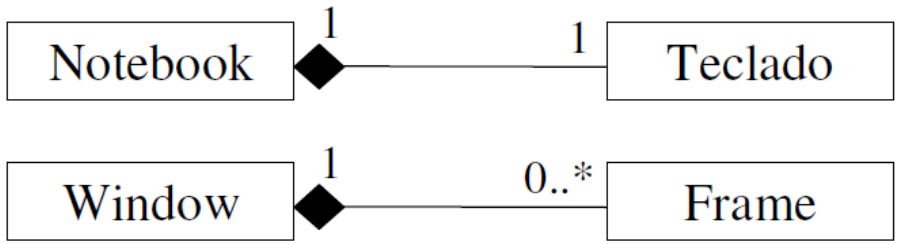
Composição – UML

- Relacionamento: Agregação
  - É um tipo especial de associação;
  - Utilizada para indicar “todo-parte”;
  - Um objeto “parte” pode fazer parte de vários objetos “todo”.



Composição – UML

- Relacionamento: Composição
  - É uma variante semanticamente mais “forte” da agregação;
  - Os objetos “parte” só podem pertencer a um único objeto “todo” e têm o seu tempo de vida coincidente com o dele;
  - Quando o “todo” morre todas as suas “partes” também morrem;

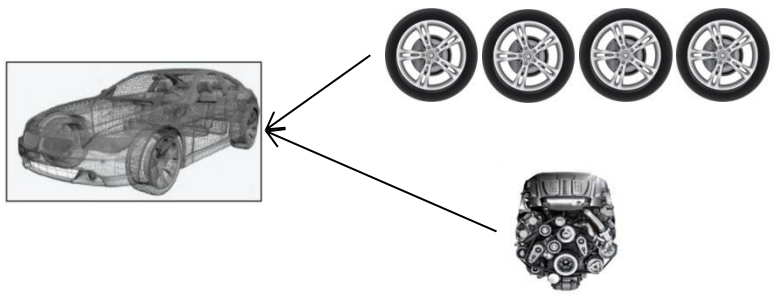


Composição

```
public class Carro{
    private Roda[] rodas;
    private Motor motor;

    public Carro()
    {
        rodas = new Roda[4];
        for (int x = 0; x < 4; x++)
           rodas[x] = new Roda();

        motor = new Motor();
    }

    ...
}
```

```
public class Roda{
    ...
    public Roda()
    {
        ...
    }
    ...
}
```

```
public class Motor{
    ...
    public Roda()
    {
        ...
    }
    ...
}
```



Pilares da Orientação a Objetos

- A orientação a objetos está sedimentada sobre quatro pilares derivados do princípio da abstração:
  - Encapsulamento
  - Herança
  - Composição
  - **Polimorfismo**

Polimorfismo

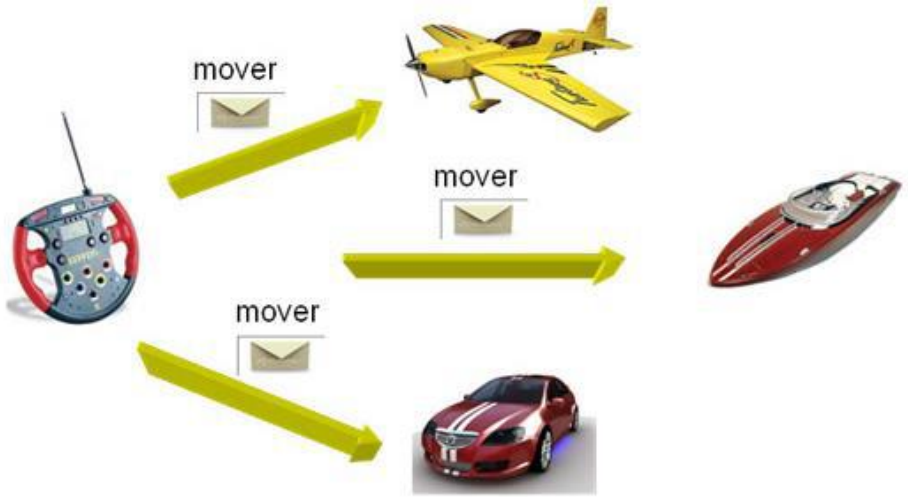
- O polimorfismo deriva da palavra polimorfo, que significa multiforme, ou que pode variar a forma.

- Na orientação a objetos, 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.

Polimorfismo

- **Exemplo:** Um dono de uma fábrica de brinquedos solicitou que seus engenheiros criassem um mesmo controle remoto para todos os brinquedos de sua fábrica.

- Assim quando o brinquedo recebe o sinal MOVER, ele se move de acordo com a sua função:
  - Para o avião, mover significa VOAR;
  - Para o barco significa NAVEGAR;
  - Para o automóvel CORRER;



Polimorfismo

- O Polimorfismo pode ser classificado de três maneiras:
  - **Polimorfismo de sobrecarga:** permite que uma classe possa ter vários métodos com o mesmo nome, mas com assinaturas distintas;
  - **Polimorfismo de Sobreposição:** permite que uma classe possua um método com a mesma assinatura (nome, tipo e ordem dos parâmetros) que um método da sua superclasse (o método da classe derivada sobrescreve o método da superclasse);
  - **Polimorfismo de Inclusão:** permite que um objeto de uma classe superior assuma a forma de qualquer um dos seus descendentes;

Polimorfismo de Sobrecarga

- Polimorfismo de sobrecarga:

```
public class Carro extends Terrestre{

    ...

    public void Acelerar()
    {
        velocidade += marcha * 10;
    }

    public void Acelerar(double forca)
    {
        velocidade += marcha * 10 * forca;
    }

    ...
}
```

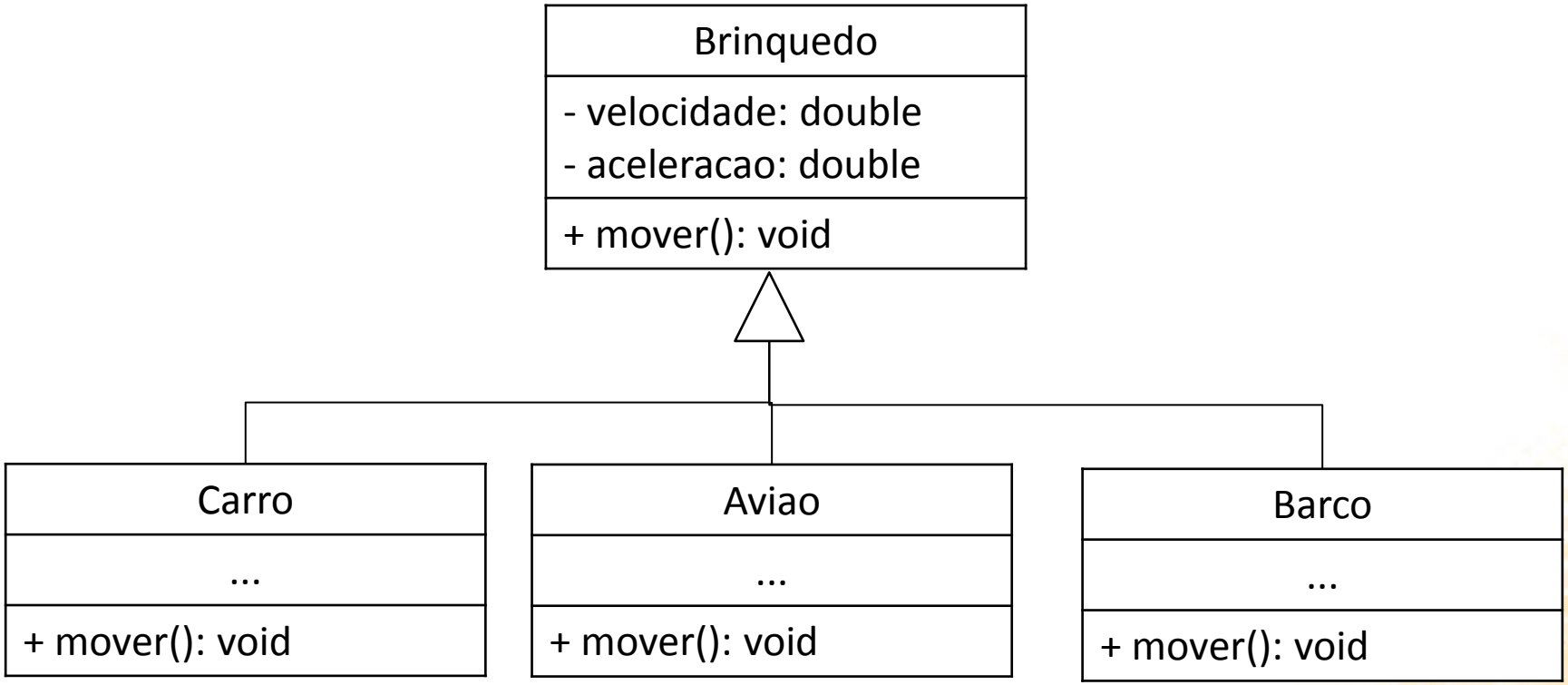
Polimorfismo de Sobrecarga

- Polimorfismo de sobrecarga de métodos construtores:

```
public class Carro extends Terrestre{

    ...

    public Carro()
    {
        super(4);
    }

    public Carro(String ncor, String nplaca)
    {
        super(4);
        cor = ncor;
        placa = nplaca;
        marcha = 0;
    }
    ...
}
```

Polimorfismo de Sobreposição

- Considere que a classe Brinquedo possui como descendentes as classes Carro, Avião e Barco:



Polimorfismo de Sobreposição

```
public class Brinquedo{
    ...
    public Brinquedo()
    {
    }
    public void mover()
    {
        System.out.print("Mover brinquedo!");
    }
}
```

```
public class Carro extends Brinquedo{
    public Carro()
    {
    }
    @Override
    public void mover()
    {
        System.out.print("CORRER!");
    }
}
```

Polimorfismo de Sobreposição

```
public class Aviao extends Brinquedo{
    public Aviao()
    {
    }
    @Override
    public void mover()
    {
        System.out.print("VOAR!");
    }
}
```

```
public class Barco extends Brinquedo{
    public Barco()
    {
    }
    @Override
    public void mover()
    {
        System.out.print("NAVEGAR!");
    }
}
```

Polimorfismo de Sobreposição

```
public class ControleRemoto{
    private Brinquedo brinquedo;

    public ControleRemoto(Brinquedo b)
    {
        brinquedo = b;
    }

    public void mover()
    {
        brinquedo.mover();
    }
}
```

Polimorfismo de inclusão

```
public static void main(String[] args) {
    Carro carro = new Carro();
    ControleRemoto = new ControleRemoto(carro);
    ControleRemoto.mover();
}
```

Classes Abstratas e Operações Abstratas

- O modificador `abstract` pode ser utilizado na declaração de uma classe para definir que ela é abstrata.
  - Uma classe abstrata normalmente possui um ou mais métodos abstratos.

- Um método abstrato não possui implementação, seu propósito é obrigar as classes descendentes a fornecerem implementações concretas das operações.

- Uma classe abstrata não pode ser instanciada diretamente.

Classes Abstratas e Operações Abstratas

```
public abstract class Terrestre extends Transporte{

    ...

    public abstract void Acelerar();
}
```

```
public class Carro extends Terrestre{

    ...

    @Override
    public void Acelerar()
    {
        velocidade += marcha * 10;
    }
}
```

Ocorrerá um erro de compilação se a classe Carro não implementar o método Acelerar.

Exercícios

Lista de Exercícios 01

http://www.inf.puc-rio.br/~elima/poo/