

Interfaces e Polimorfismo

Polimorfismo (poli=muitos, morfo=forma) é uma característica essencial de linguagens orientadas a objeto

Como funciona?

- Um objeto que faz papel de interface serve de **intermediário** fixo entre o programa-cliente e os objetos que irão executar as mensagens recebidas
- O programa-cliente não precisa saber da existência dos outros objetos
- Objetos podem ser substituídos sem que os programas que usam a interface sejam afetados

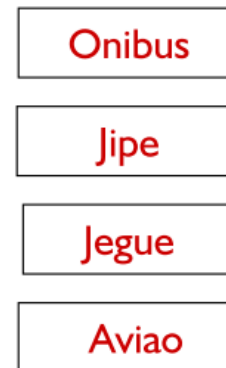
- *Polimorfismo significa que um objeto pode ser usado no lugar de outro objeto*

*Usuário do objeto
enxerga somente esta interface*

*freia()
acelera()
vira(l)*



- Uma interface
- Múltiplas implementações

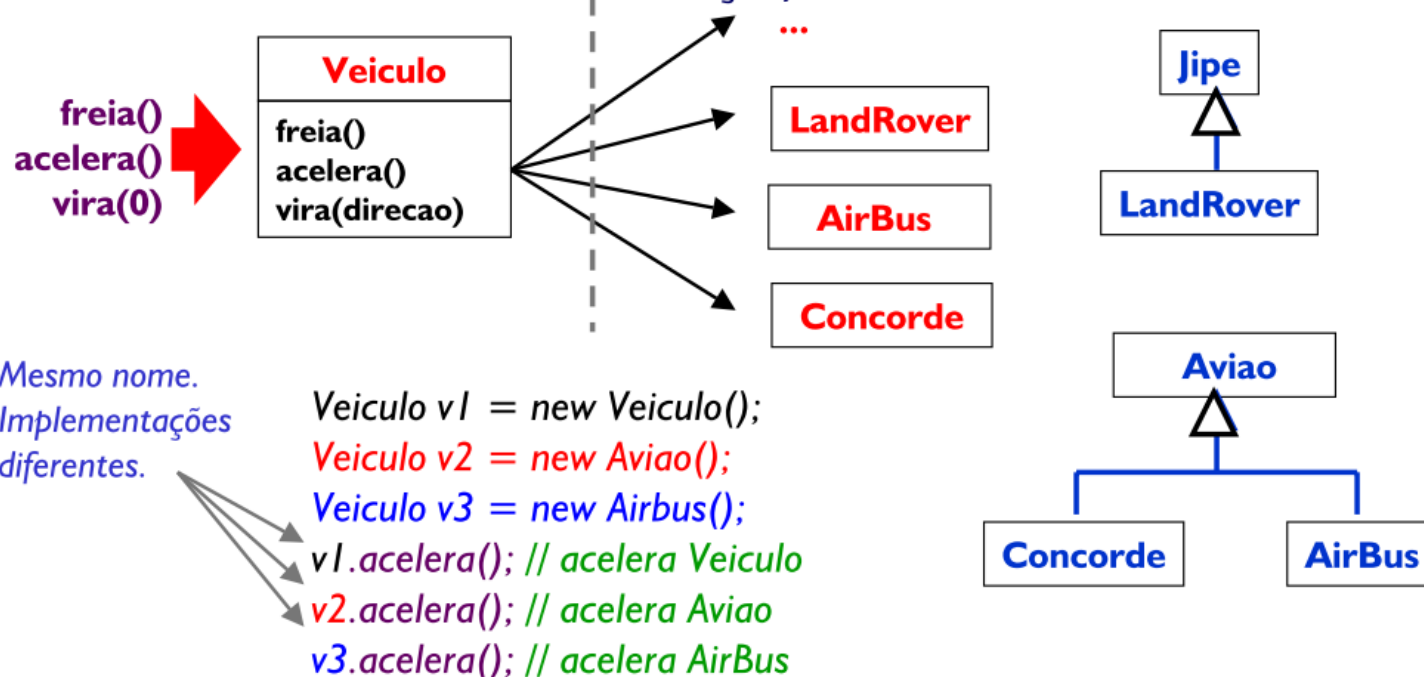


*Subclasses
de Veiculo!
(herdam
todos os
métodos)*

*Por exemplo: objeto do tipo **Manobrista** sabe usar comandos básicos para controlar **Veiculo** (não interessa a ele saber como cada **Veiculo** diferente vai acelerar, frear ou mudar de direção). Se outro objeto tiver a mesma interface, **Manobrista** saberá usá-lo*

*Usuário de Veiculo
ignora existência desses
objetos substituíveis*

- *Novos objetos podem ser usados em programas que não previam a sua existência*
 - *Garantia que métodos da interface existem nas classes novas*
 - *Objetos de novas classes podem ser criados e usados (programa pode ser estendido durante a execução)*



Polimorfismo permite **separar a interface da implementação**

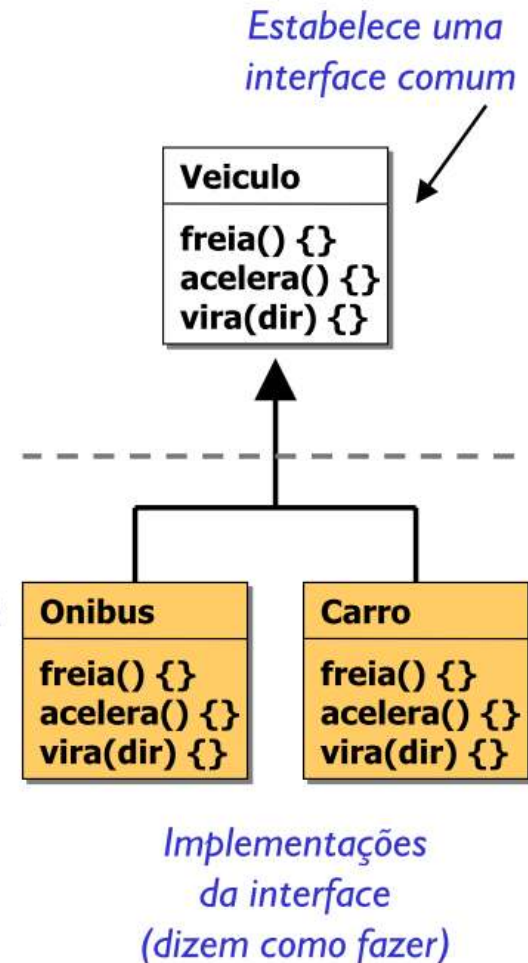
A classe base define a interface comum

- Não precisa dizer **como** isto vai ser feito

Não diz: eu sei como frear um Carro ou um Ônibus

- Diz apenas que os métodos existem, que eles retornam determinados tipos de dados e que requerem certos parâmetros

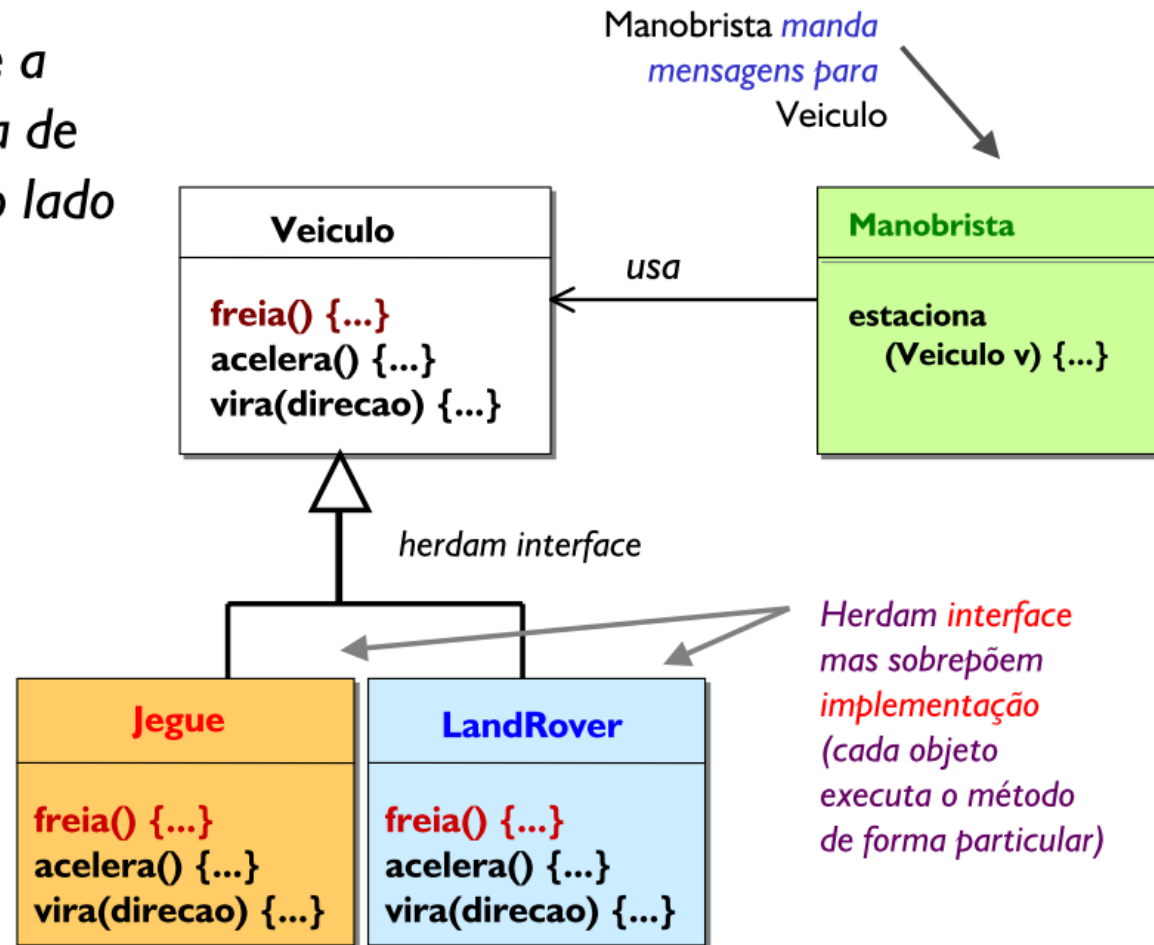
Diz: Veiculo pode acelerar, frear e virar para uma direção, mas a direção deve ser fornecida



Suporte a polimorfismo depende do suporte à **ligação tardia** (*late binding*) de chamadas de função

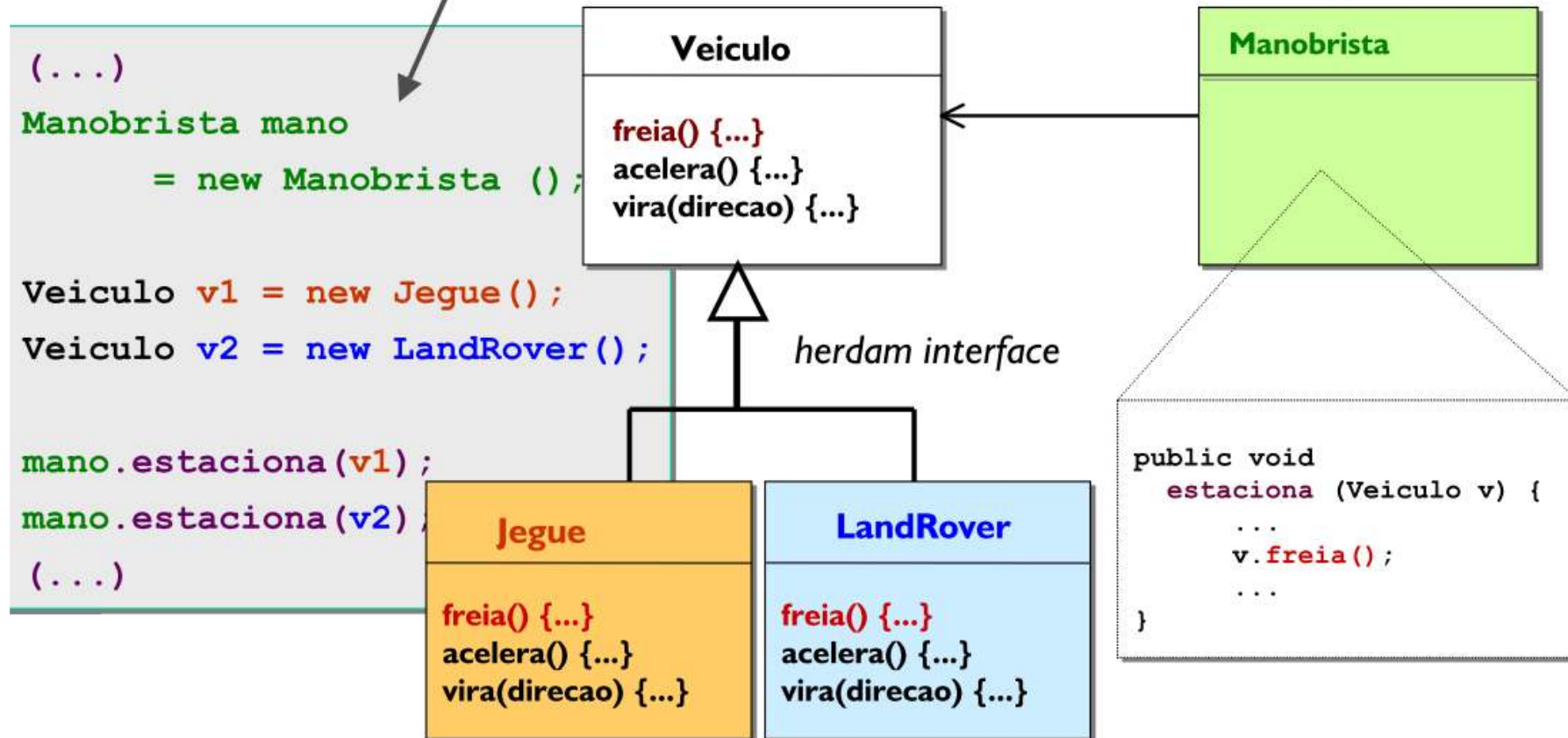
- A referência (interface) é conhecida em **tempo de compilação** mas o objeto a que ela aponta (implementação) não é
- O objeto pode ser da mesma classe ou de uma subclasse da referência (garante que a TODA a interface está implementada no objeto)
- Uma única referência, pode ser ligada, **durante a execução**, a vários objetos diferentes (a referência é polimorfa: pode assumir muitas formas)

Considere a
hierarquia de
classes ao lado



Trecho de programa que **usa** Manobrista:
Em tempo de execução passa implementação de Jegue e LandRover no lugar da implementação original de Veiculo
(aproveita apenas a interface de Veiculo)

Manobrista **usa** a classe Veiculo (e ignora a existência de tipos específicos de Veiculo como Jegue e LandRover)



Como deve ser implementado *freia()* na classe *Veiculo*?

- Faz sentido dizer como um veículo genérico deve frear?
- Como garantir que cada tipo específico de veículo redefina a implementação de *freia()*?

O método *freia()* é um procedimento **abstrato** em *Veiculo*

- Deve ser usada apenas a implementação das subclasses

E se não houver subclasses?

- Como *freia* um *Veiculo* genérico?
- Com que se parece um *Veiculo* generico?

Conclusão: não há como construir objetos do tipo *Veiculo*

- É um conceito genérico demais
- Mas é ótimo como interface! Eu posso saber dirigir um *Veiculo* sem precisar saber dos detalhes de sua implementação

Procedimentos genéricos que têm a finalidade de servir apenas de interface são **métodos abstratos**

- declarados com o modificador **abstract**
- não têm corpo {}. Declaração termina em ";"

```
public abstract void freia();
```

```
public abstract float velocidade();
```

Métodos abstratos não podem ser **usados**, apenas declarados

- São usados através de uma subclasse que os implemente!

Uma classe pode ter métodos concretos e abstratos

- Se tiver **um ou mais método abstrato**, classe não pode ser usada para criar objetos e precisa ter declaração **abstract**

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

- Objetos do tipo Veiculo não podem ser criados
- Subclasses de Veiculo podem ser criados desde que implementem TODOS os métodos abstratos herdados
- Se a implementação for parcial, a subclasse também terá que ser declarada abstract

Classes abstratas são criadas para serem estendidas

Podem ter

- *métodos concretos (usados através das subclasses)*
- *campos de dados (memória é alocada na criação de objetos pelas suas subclasses)*
- *construtores (chamados via `super()` pelas subclasses)*

Classes abstratas "puras"

- *não têm procedimentos no construtor (construtor vazio)*
- *não têm campos de dados (a não ser constantes estáticas)*
- *todos os métodos são abstratos*

Classes abstratas "puras" podem ser definidas como

*"**interfaces**" para maior flexibilidade de uso*

Tipos genéricos (acima, na hierarquia) sempre podem receber objetos de suas subclasses:

upcasting

```
Veiculo v = new Carro();
```

- *Há garantia que subclasses possuem pelo menos os mesmos métodos que a classe*
- *v só tem acesso à "parte Veiculo" de Carro. Qualquer extensão (métodos definidos em Carro) não faz parte da extensão e não pode ser usada pela referência v.*

Tipos específicos (abaixo, na hierarquia) não podem receber explicitamente seus objetos que foram declarados como referências de suas superclasses:

downcasting

```
Carro c = v; // não compila!
```

- *O código acima não compila, apesar de v apontar para um Carro! É preciso converter a referência:*

```
Carro c = (Carro) v;
```

- *E se v for Onibus e não Carro?*

Interface é uma estrutura que representa uma classe abstrata "pura" em Java

- *Não têm atributos de dados (só pode ter constantes estáticas)*
- *Não tem construtor*
- *Todos os métodos são abstratos*
- *Não é declarada como class, mas como interface*

*Interfaces Java servem para fornecer **polimorfismo sem herança***

- *Uma classe pode "herdar" a interface (assinaturas dos métodos) de várias interfaces Java, mas apenas de uma classe*
- *Interfaces, portanto, oferecem um tipo de herança múltipla*

Em linguagens como C++ , uma classe pode herdar métodos de duas ou mais classes

- *A classe resultante pode ser usada no lugar das suas duas superclasses via upcasting*
- *Vantagem de herança múltipla: mais flexibilidade*

Problema

- *Se duas classes A e B estenderem uma mesma classe Z e herdarem um método x() e, uma classe C herdar de A e de B, qual será a implementação de x() que C deve usar? A de A ou de B?*
- *Desvantagem de herança múltipla: ambigüidade. Requer código mais complexo para evitar problemas desse tipo*

```
interface Empregado {  
    void trabalha();  
}
```

```
interface Cidadao {  
    void vota();  
    int getRG();  
}
```

```
interface Professor  
    extends Empregado {  
    void ensina();  
}
```

```
interface Contribuinte {  
    boolean pagaIR();  
    long getCPF();  
}
```

- Todos os métodos são implicitamente
 - *public*
 - *abstract*
- Quaisquer campos de dados têm que ser inicializados e são implicitamente
 - *static*
 - *final (constantes)*
- Indicar *public*, *static*, *abstract* e *final* é opcional
- Interface pode ser declarada *public* (default: *package-private*)

```
public class Pessoa
    extends Animal
    implements Professor, Cidadao, Contribuinte {

    public void ensina() { /* votar */ }
    public void vota() { /* votar */ }
    public int getRG() { return 12345; }
    public void trabalha() {}
    public boolean pagaIR() { return false; }
    public long getCPF() { return 1234567890; }
}
```

- Palavra *implements* declara interfaces implementadas
 - Exige que **cada um dos métodos** de cada interface sejam de fato implementados (na classe atual ou em alguma superclasse)
 - Se alguma implementação estiver faltando, classe só compila se for declarada *abstract*

```
public class Cidade {
    public void contrata(Professor p) {
        p.ensina();
        p.trabalha();
    }
    public void contrata(Empregado e) { e.trabalha(); }
    public void cobraDe(Contribuinte c) { c.pagaIR(); }
    public void registra(Cidadao c) { c.getRG(); }
    public void alimenta(Animal a) { a.come(); }

    public static void main (String[] args) {
        Pessoa joao = new Pessoa();
        Cidade sp = new Cidade();
        sp.contrata(joao); // considera Professor
        sp.contrata( (Empregado) joao); // Empregado
        sp.cobraDe(joao); // considera Contribuinte
        sp.registra(joao); // considera Cidadao
        sp.alimenta(joao); // considera Animal
    }
}
```

Use interfaces sempre que possível

- Seu código será mais **reutilizável!**
- Classes que já herdam de outra classe podem ser facilmente redesenhadas para implementar uma interface sem quebrar código existente que a utilize

Planeje suas interfaces com muito cuidado

- É mais fácil evoluir classes concretas que interfaces
- Não é possível acrescentar métodos a uma interface depois que ela já estiver em uso (as classes que a implementam não compilarão mais!)
- Quando a evolução for mais importante que a flexibilidade oferecido pelas interfaces, deve-se usar classes abstratas.