Universidade Federal da Paraíba – Campus I
Centro de Informática
Departamento de Sistemas e Computação

Métodos e Projeto de Software
Material 8: Revisão de Conceitos Básicos de O.O – Parte 3

Prof. Raoni Kulesza
raoni@ci.ufpb.br

Objetivos

- Discutir o uso de herança com outros mecanismos de reuso de código, como a *Composição e Polimorfismo*
- São apresentados também os seguintes conceitos relacionados a polimorfismo:
  - Métodos e classes abstratas,
  - Upcasting,
  - Downcasting
  - Interfaces e herança múltipla.

Reuso com Composição

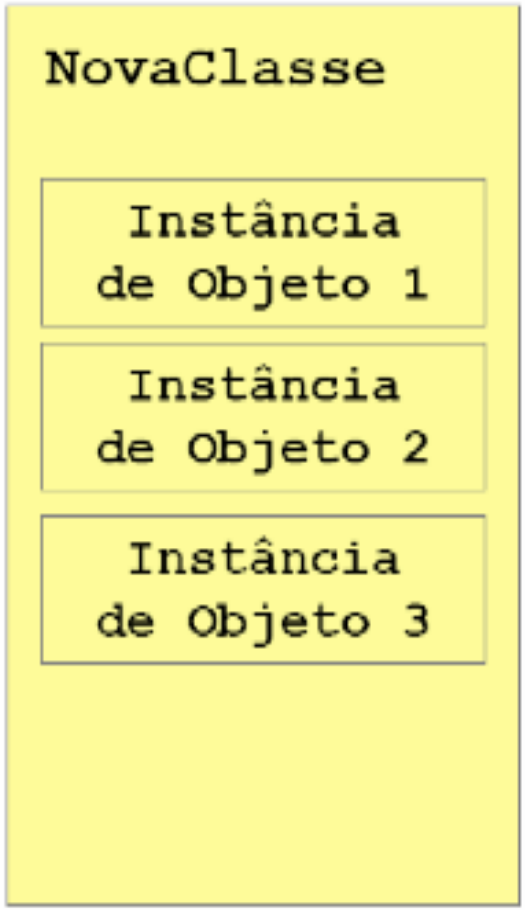
Como aumentar as chances de reuso?

- Separar as partes que podem mudar das partes que que não mudam
  - Descobrindo o que irá mudar, a idéia é encapsular isso, trabalhar com uma interface e usar o código sem a preocupação de ter que reescrever tudo caso surjam versões futuras (fraco acoplamento.)
- Programe para uma interface, e não para uma implementação (concreta)
  - Explorar o polimorfismo e a ligação dinâmica para usar um supertipo e poder trocar objetos distintos em tempo de execução

Composição - Conceito

- A composição estende uma classe pela delegação de trabalho para outro objeto
  - Em vez de codificar um comportamento estaticamente, definimos e encapsulamos pequenos comportamentos padrão e usamos composição para delegar comportamentos
- Muitos autores já consideram a composição muito superior à herança em grande parte dos casos
  - Posso mudar a associação entre classes em tempo de execução (chamadas delegadas)
  - Permite que um objeto assuma mais de um comportamento (composição de vários outros objetos)

Composição em Java




```
class NovaClasse {
    Um um = new Um();
    Dois dois = new Dois();
    Tres tres = new Tres();
}
```

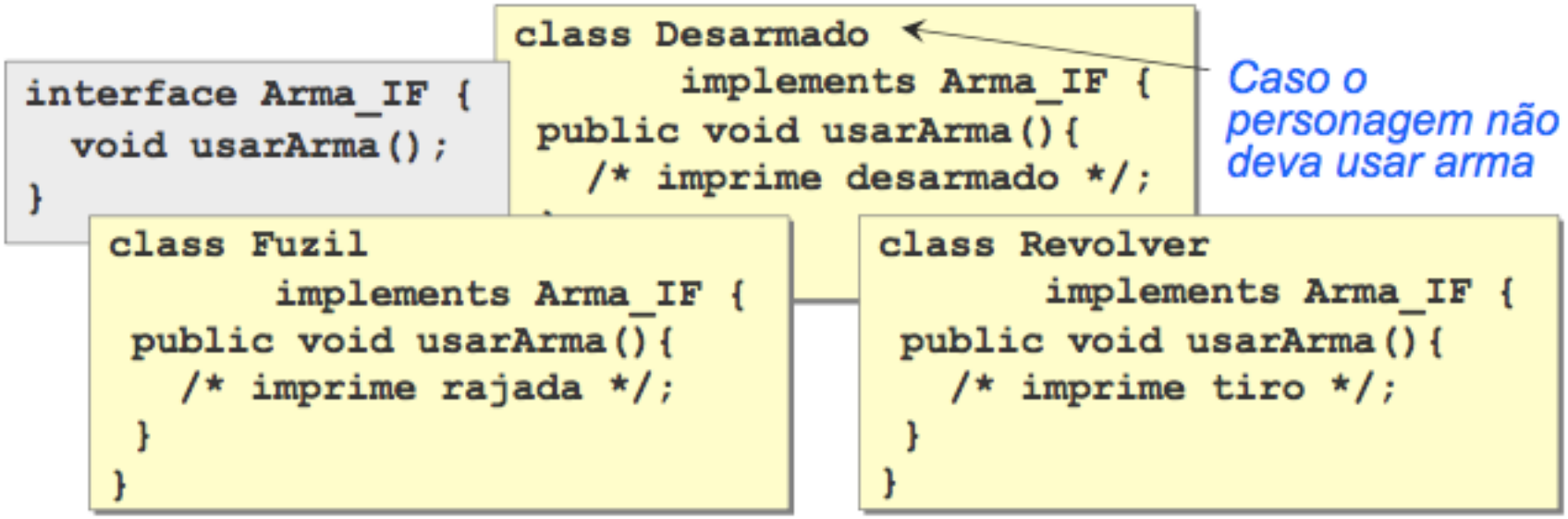
- *Objetos podem ser inicializados no construtor*
- *Flexibilidade*
  - *Pode trocar objetos durante a execução!*
- *Relacionamento*
  - *"TEM UM"*

Composição - Qual é o objetivo?

- Permitir o aparecimento de novas armas no Jogo sem grandes impactos negativos (if-else's) em todas as subclasses concretas
  - Permitir que um personagem mude de arma em tempo de execução (passagem de fases, e.g.)
    - Em vez de já definir a arma estaticamente no código
  - Alguns comportamentos não devem ser compartilhados por todos os personagens
    - Ex.: atirar com um revólver não deve ser executado por um objeto do tipo lutador de Sumô

Cenário de Uso 3

- Se a armas devem mudar, a partir de agora, devemos separá-las das classes concretas (soldado, general)



Cenário de Uso 3

- Todos os personagens concretos devem trabalhar com uma referência para alguma coisa que implemente a interface de armas

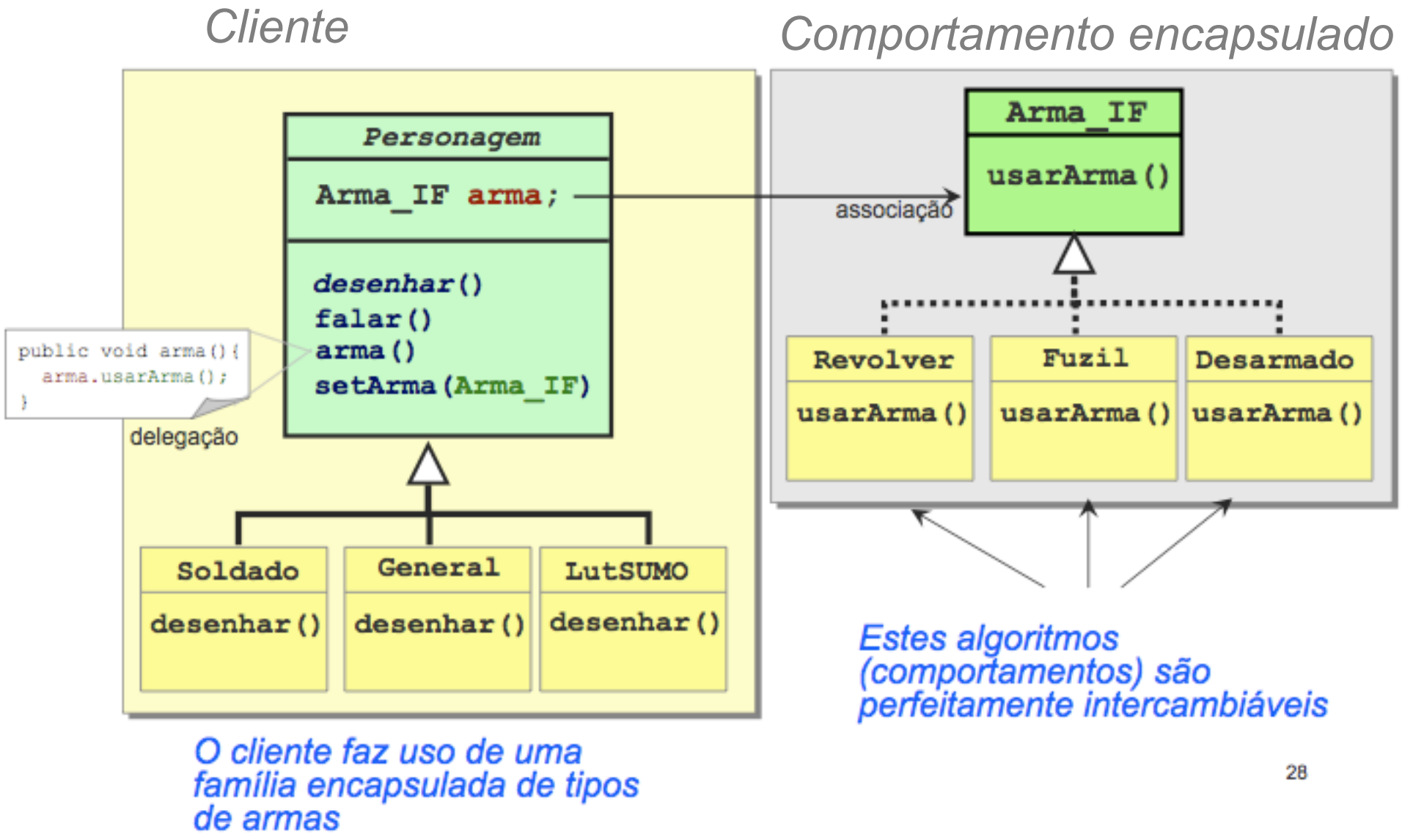
```
public abstract class Personagem {
 Arma_IF arma;

 public abstract void desenhar();
 public void falar(){
   /* código comum para falar */
 }
 public void setArma(Arma_IF a){
    arma = a;
 }
 public void arma(){
    arma.usarArma();
 }
}
```

Reduz o acoplamento de código, já que os personagens interagem com interface (em vez de uma implementação)

Se um personagem deseja usar sua arma, ele simplesmente delega esta tarefa ao objeto (alguém que tem o método usarArma) que está sendo referenciado no momento

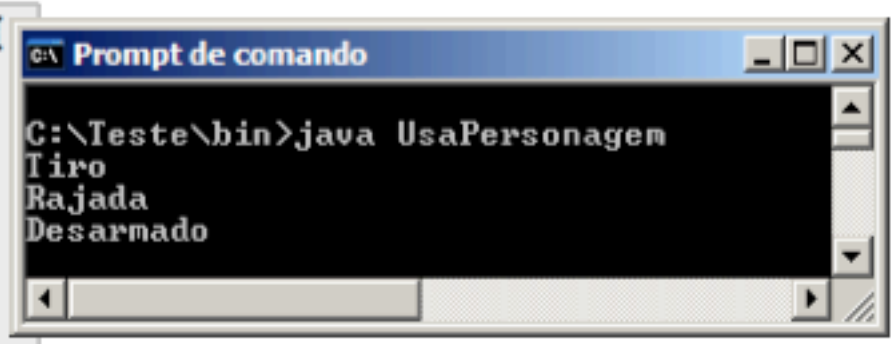
O polimorfismo e a ligação dinâmica irão cuidar de chamar o método (usarArma) correto

Diagrama de classes da solução



Executando o novo código

```
public abstract class Personagem {
 Arma_IF arma;

 public abstract void desenhar();
 public void falar(){
   /* código comum para falar */
 }
 public void setArma(Arma_IF a){
    arma = a;
 }
 public void arma(){
    arma.usarArma();
 }
}
```




```
public class UsaPersonagem {
 public static void main(String[] args){
   Personagem p;

   p = new Soldado();
   p.desenhar();
   p.setArma( new Revolver() ); // define arma
   p.arma(); // imprime "Tiro"
   p.setArma( new Fuzil() ); // trocou arma
   p.arma(); // imprime Rajada

   p = new LutadorSumo();
   p.desenhar();
   p.setArma( new Desarmado() ); // define arma
   p.arma(); // imprime "Desarmado"

}
```

Note a presença de interfaces

Ligação dinâmica

* *E se amanhã surgisse uma nova arma (Faca, por ex.)?*

Considerações: composição e herança (1)

- Uma das principais atividades em um projeto orientado a objetos é estabelecer relacionamentos entre classes
- Duas formas básicas de relacionar classes são: herança e a composição
- Composição e herança **NÃO** são mutuamente exclusivas, ou seja, podem ser utilizadas em conjunto
- A herança, geralmente, ocorre mais no projeto (*design*) de tipos (uma subclasse é um tipo de...)

Considerações: composição e herança (2)

- No desenvolvimento, porém, a composição de objetos é a técnica predominante
  - Separar o que muda do que não muda (e encapsular estes pequenos comportamentos)
  - Trabalhar com uma interface para manipular estes pequenos comportamentos
  - Trocar comportamentos dinamicamente
- Programar para uma interface sempre que possível
  - Garante um fraco acoplamento

Quando usar? Composição ou Herança

- Identifique os componentes do objeto, suas partes
  - Essas partes devem ser agregadas ao objeto via composição (relacionamento do tipo “É PARTE DE”)
- Classifique seu objeto e tente encontrar uma semelhança de identidade com classes existentes
  - Herança só deve ser usada se você puder comparar seu objeto A com outro B dizendo que A “É UM tipo de...” B

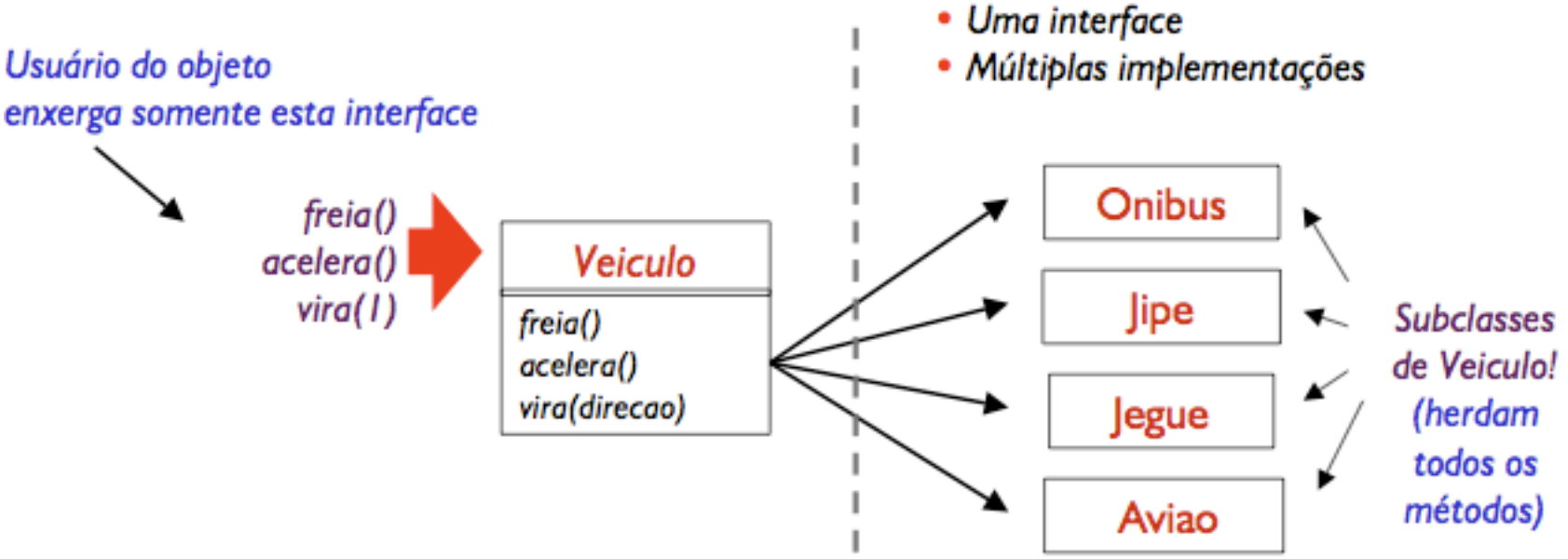
Reuso com Composição

O que é 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

Objetos substituíveis

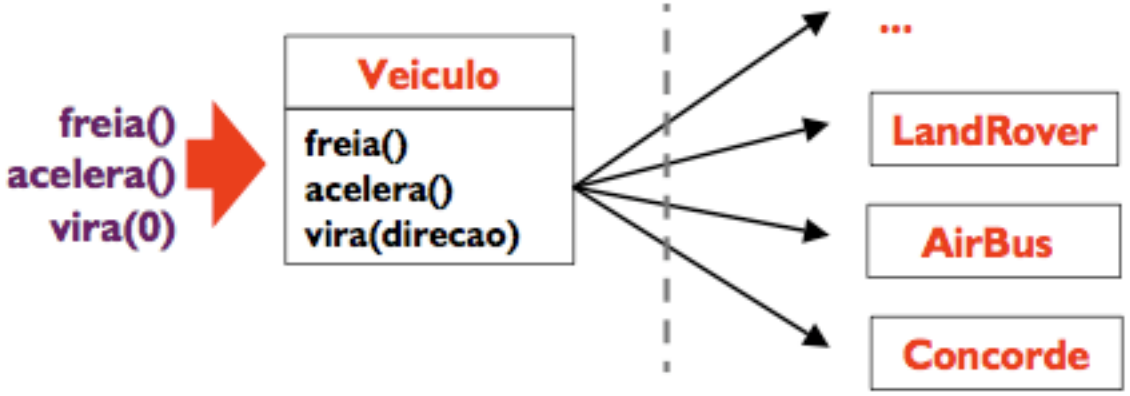
- Polimorfismo significa que um objeto pode ser usado no lugar de outro objeto
- Como funciona?

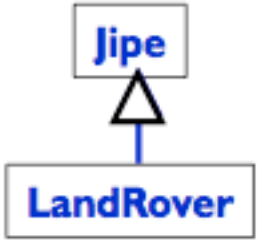


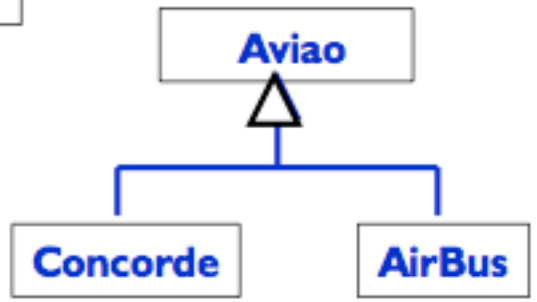
Por exemplo: objeto do tipo Manobrista sabe usar comandos básicos para controlar Veiculo (não interessa a ele saber como cada Veiculo diferente vai acelerar, frear ou mudar de direção). Se outro objeto tiver a mesma interface, Manobrista saberá usá-lo

Programas extensíveis

- 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)





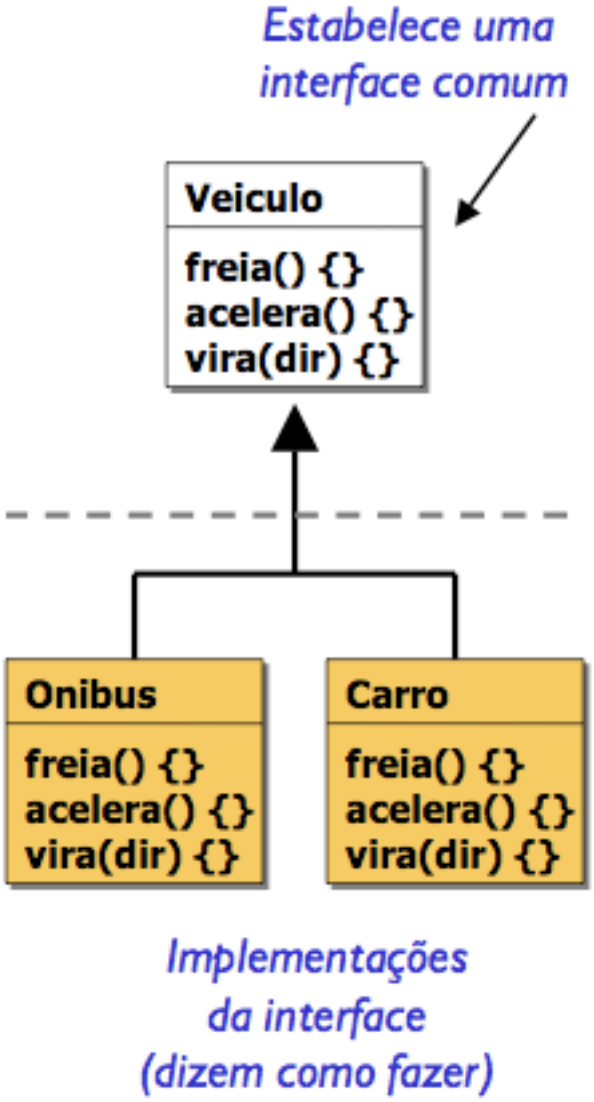


Mesmo nome. Implementações diferentes.

```
Veiculo v1 = new Veiculo();
Veiculo v2 = new Aviao();
Veiculo v3 = new Airbus();
v1.acelera(); // acelera Veiculo
v2.acelera(); // acelera Aviao
v3.acelera(); // acelera AirBus
```

Interfaces vs. Implementação

- 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

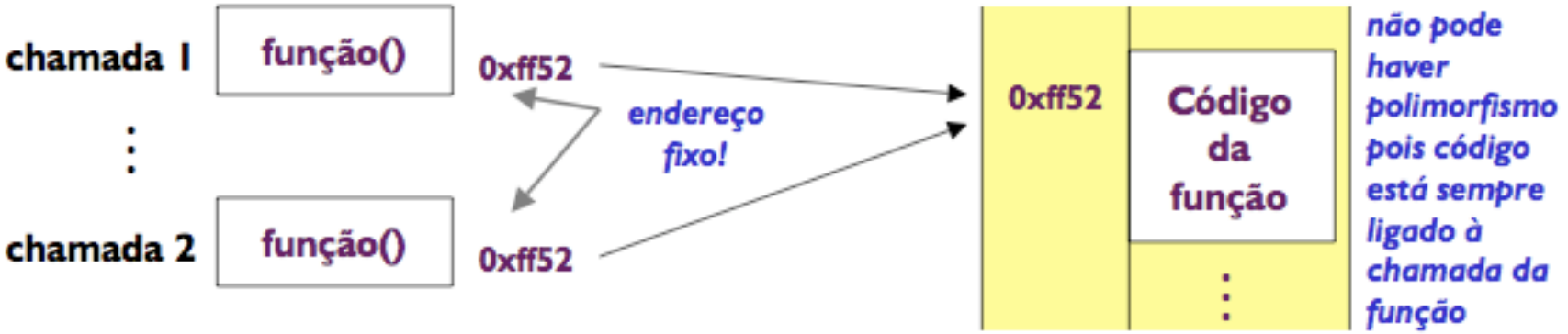


Como funciona?

- 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)

Ligação de chamadas de função

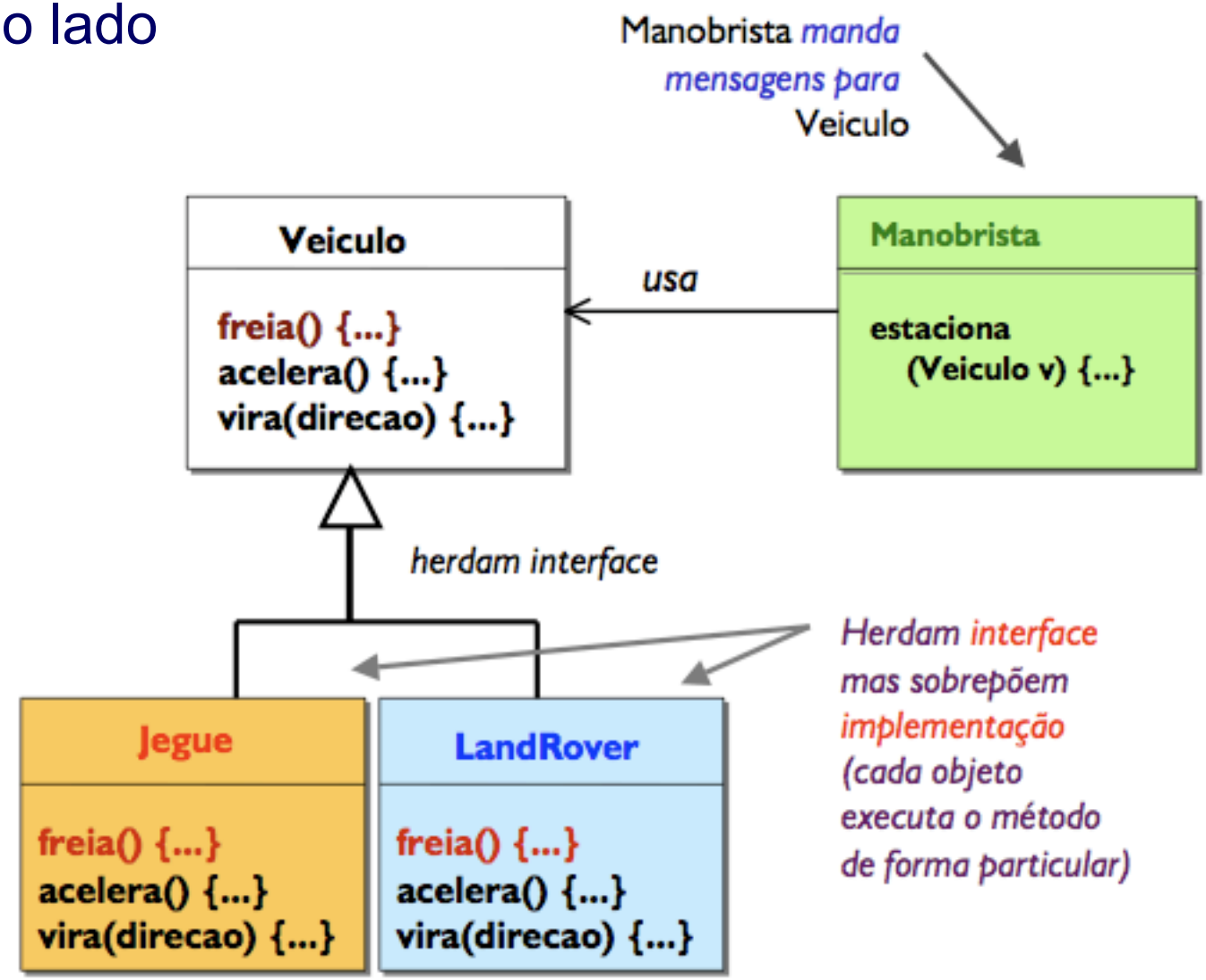
- Em tempo de compilação (*early binding*) – *C*



- Em tempo de execução(*late binding*) – *Java*

Exemplo (1)

- Conside a hierarquia de classes ao lado

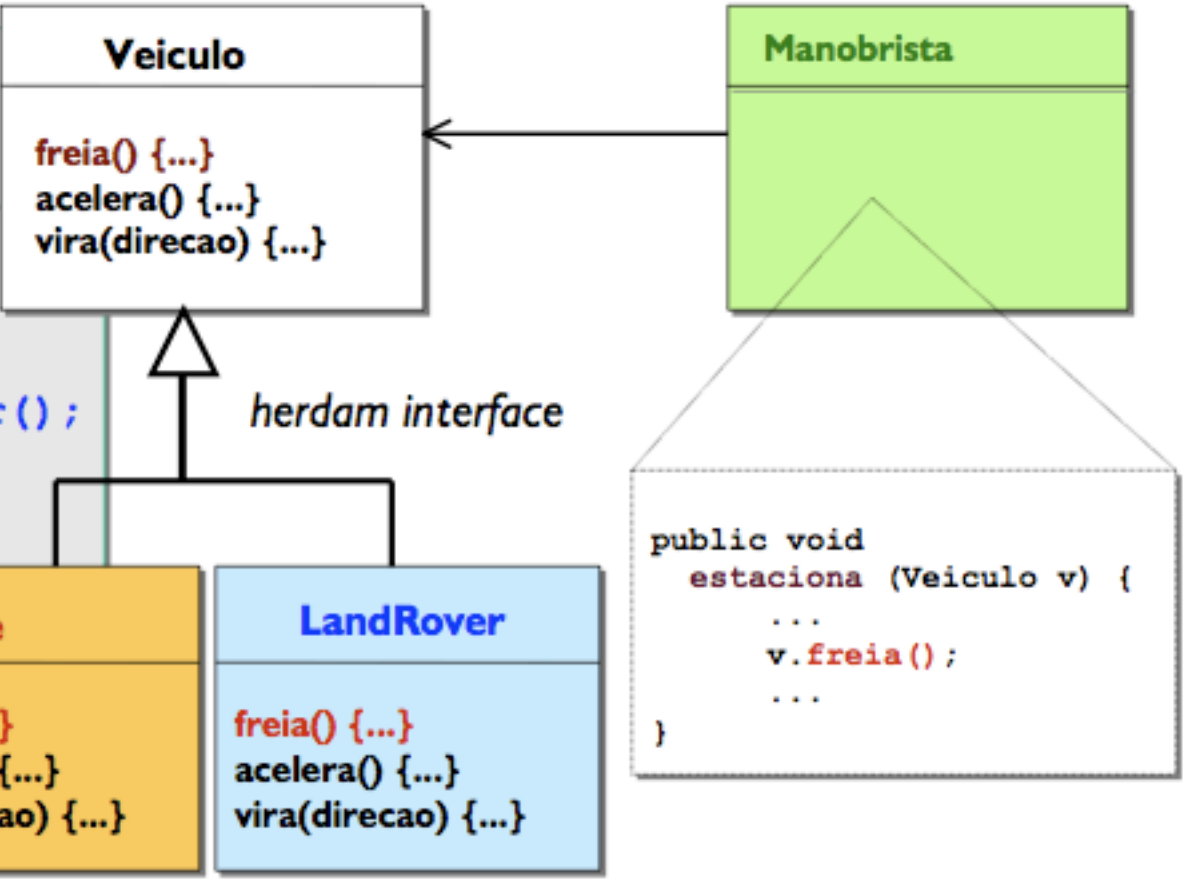


Exemplo (2)

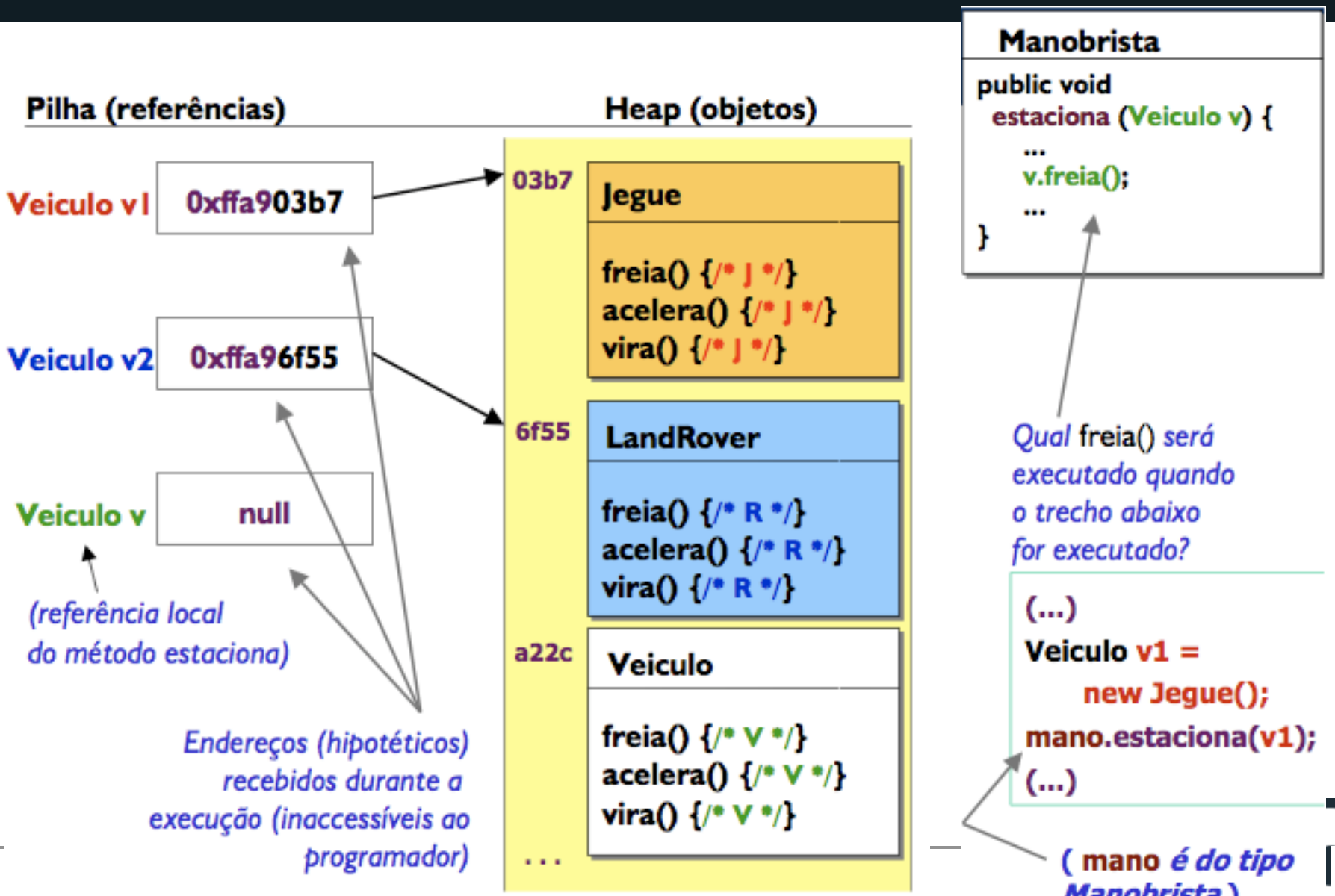
Trecho de programa que usa Manobrista:
Em tempo de execução passa implementação de Jegue e LandRover no lugar da implementação original de Veiculo
(aproveita apenas a interface de Veiculo)

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

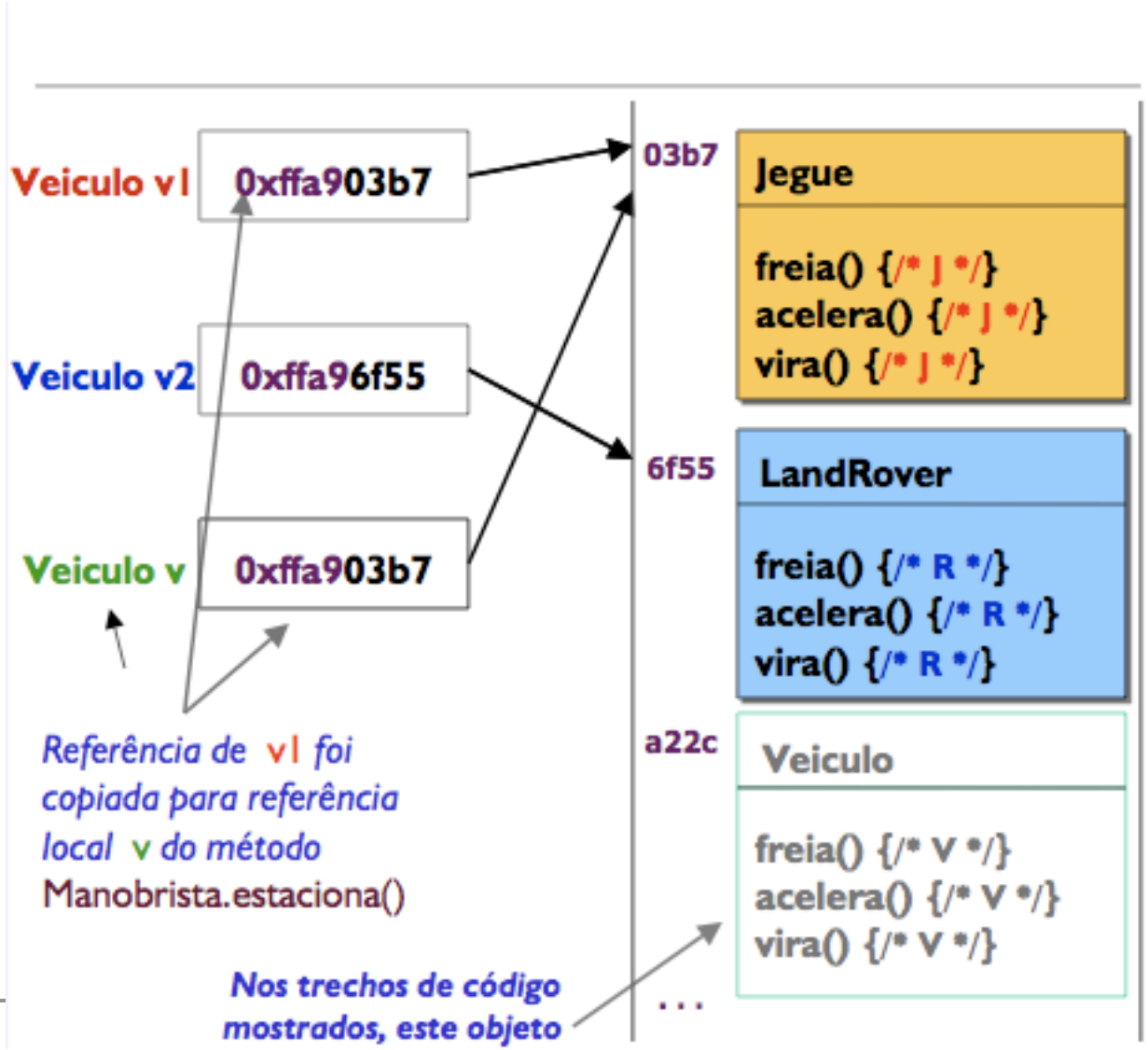
```
(...)
Manobrista mano
      = new Manobrista ();

Veiculo v1 = new Jegue();
Veiculo v2 = new LandRover();

mano.estaciona(v1);
mano.estaciona(v2);
(...)
```



Detalhes (1)


Pilha (referências)
Heap (objetos)
Veiculo v1
0xffa903b7
Veiculo v2
0xffa96f55
Veiculo v
null
(referência local do método estaciona)
Endereços (hipotéticos) recebidos durante a execução (inaccessíveis ao programador)
03b7
Jegue
freia() {/* J */}
acelera() {/* J */}
vira() {/* J */}
6f55
LandRover
freia() {/* R */}
acelera() {/* R */}
vira() {/* R */}
a22c
Veiculo
freia() {/* V */}
acelera() {/* V */}
vira() {/* V */}
. . .
Manobrista
public void
estaciona (Veiculo v) {
...
v.freia();
...
}
Qual freia() será executado quando o trecho abaixo for executado?
(...)
Veiculo v1 =
new Jegue();
mano.estaciona(v1);
(...)
(mano é do tipo Manobrista)

Detalhes (2)



Manobrista

```
public void
  estaciona (Veiculo v) {
     ...
     v.freia();
     ...
}
```

Na chamada abaixo, Veiculo *foi "substituído" com* Jegue. *A implementação usada foi* Jegue.freia()

```
(...)
Veiculo v1 =
      new Jegue();
mano.estaciona(v1);
(...)
```

```
Veiculo v = v1
```

Argumento do método estaciona()

Conceitos Abstratos

- 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 método 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

Métodos e classes abstratas

- 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!

Classes abstratas

- Uma classe pode ter métodos concretos e abstratos
- Se tiver um ou mais método(s) abstrato(s), 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 (2)

- 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

Upcasting

- 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.

Downcasting

- 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;
```

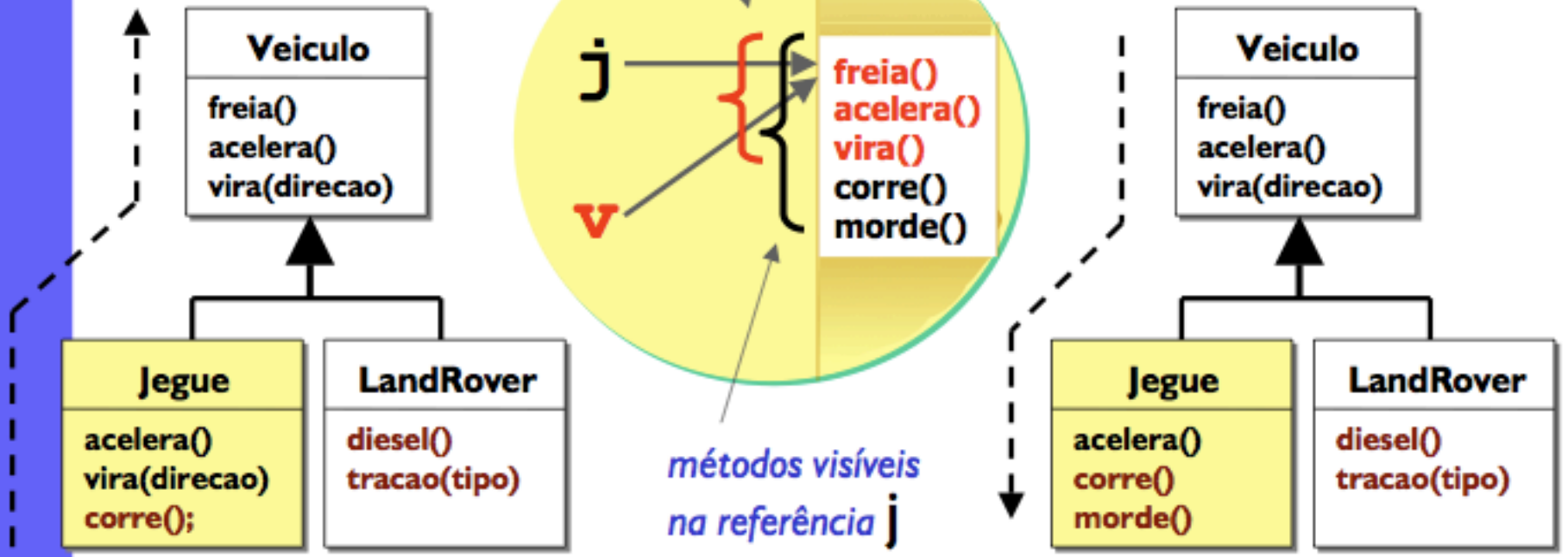
- E se `v` for Onibus e não Carro?

Upcasting x Downcasting

- **Upcasting**
  - sobe a hierarquia
  - não requer cast
- **Downcasting**
  - desce a hierarquia
  - requer operador de cast

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

```
Jegue j = (Jegue) v;
```



ClassCastException

- O downcasting explícito sempre é aceito pelo compilador se o tipo da direita for superclasse do tipo da esquerda

```
Veiculo v = new Onibus();
Carro c = (Carro) v; // passa na compilação
```

  - Object, portanto, pode ser atribuída a qualquer tipo de referência

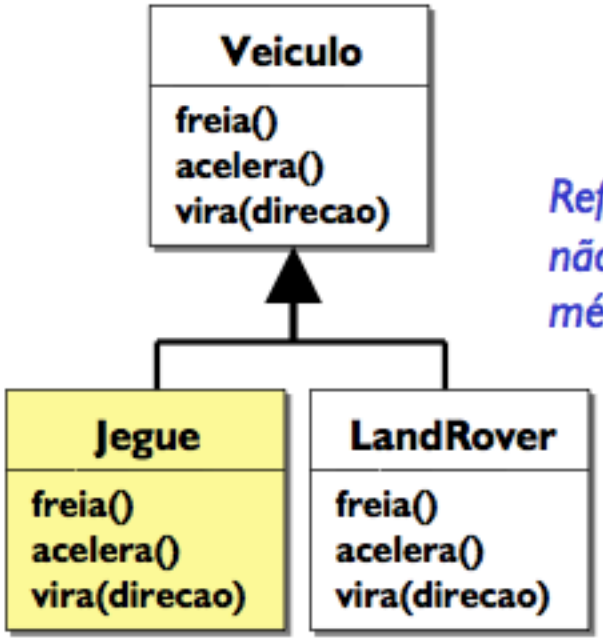
- Em tempo de execução, a referência terá que ser ligada ao objeto
  - Incompatibilidade provocará `ClassCastException`

- Para evitar a exceção, use instanceof

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

Herança Pura x Extensão

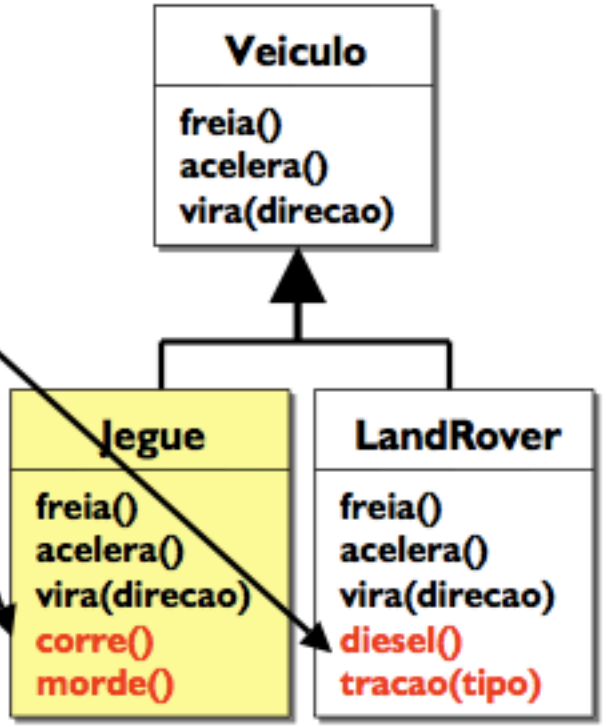
- *Herança pura: referência têm acesso a todo o objeto*




```
Veiculo v = new Jegue();
v.freia() // freia o Jegue
v.acelera(); // acelera o Jegue
```

- *Extensão: referência apenas tem acesso à parte definida na interface da classe base*

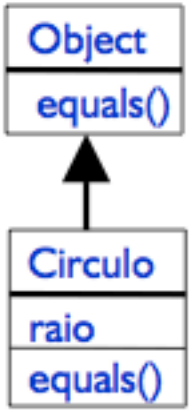
Referência Veiculo não enxerga estes métodos




```
Veiculo v = new Jegue();
v.corre() // ERRADO!
v.acelera(); //OK
```

Ampliação da referência

- *Uma referência pode apontar para uma classe estendida, mas só pode usar métodos e campos de sua interface*
  - *Para ter acesso total ao objeto que estende a interface original, é preciso usar referência que conheça toda sua interface pública*
- *Exemplo*



ERRADO: raio não faz parte da interface de Object

```
class Circulo extends Object {
  public int raio;
  public boolean equals(Object obj) {
    if (this.raio == obj.raio)
      return true;
    return false;
  }
} // CÓDIGO ERRADO!
```

verifica se obj realmente é um Circulo

cria nova referência que tem acesso a toda a interface de **Circulo**

```
class Circulo extends Object {
  public int raio;
  public boolean equals(Object obj) {
    if (obj instanceof Circulo) {
      Circulo k = (Circulo) obj;
      if (this.raio == k.raio)
        return true;
    }
    return false;
  }
}
```

Como k é **Circulo** *possui raio*

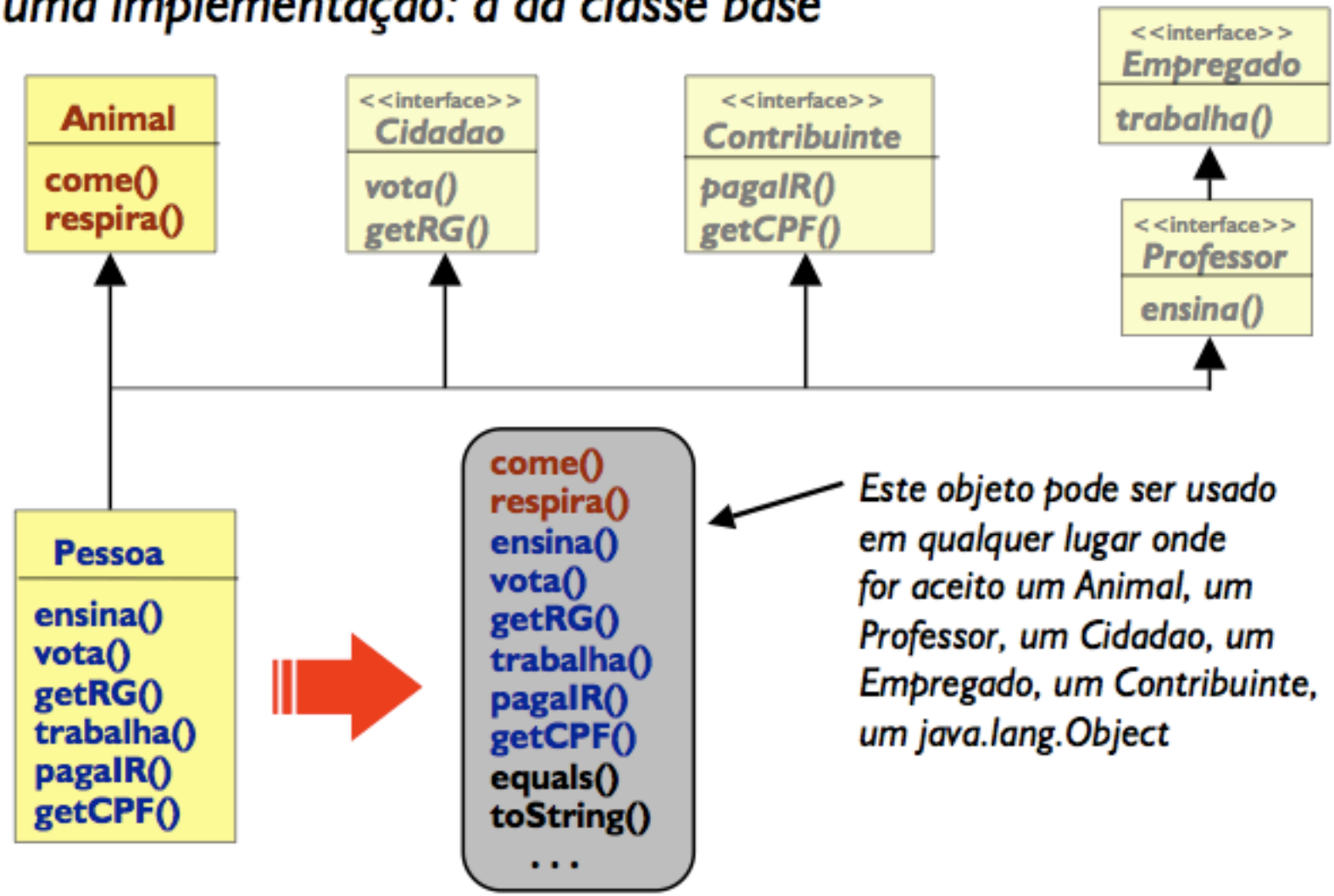
Interface Java

- 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

Herança Múltipla em Java

- *Classe resultante combina todas as interfaces, mas só possui uma implementação: a da classe base*



Exemplo (1)

```
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 inicializadas e são implicitamente
  - static
  - final (constantes)
- Indicar public, static, abstract e final é opcional
- Interface pode ser declarada public (default: package-private)

Exemplo (2)

```
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*

Exemplo (3) – Uso de interfaces

```
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
    }
}
```

Conclusões

- 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.

Dúvidas? Obrigado.

raoni@ci.ufpb.br