Programação Orientada a Objetos I

CÁSSIO CAPUCHO PEÇANHA – 07

Reuso / reutilização

- Para entregar software de qualidade em menos tempo, é preciso reutilizar;
  - Reuso é uma das principais vantagens anunciadas pela Orientação a Objetos;
  - “Copy & paste” **não** é reuso!
- É necessário o bom entendimento dos mecanismos de **dependências de classes** baseado no conceito de classes que são:
  - Associação, Composição e Agregação
  - Herança ou derivação

Dependências de classes

- Até agora aprendemos a construir relacionamentos de: Composição, Agregação e Associação
  - Ex: Uma conta tem um dono (cliente), etc..
- As suas principais características são que:
  - Usa objetos de outras classes
    - Uso de outra classe como atributo de uma nova classe
  - Não altera comportamentos
    - Reuso como Cliente
  - Cria dependências entre classes

Herança

- A outra forma de relacionamento entre as classes que temos em OO é a Herança,
  - É identificado como (“é-um”) ou (“é um tipo de”)
- Define uma hierarquia de abstrações
  - Na qual uma subclasse herda propriedades e comportamentos de uma superclasse

Herança

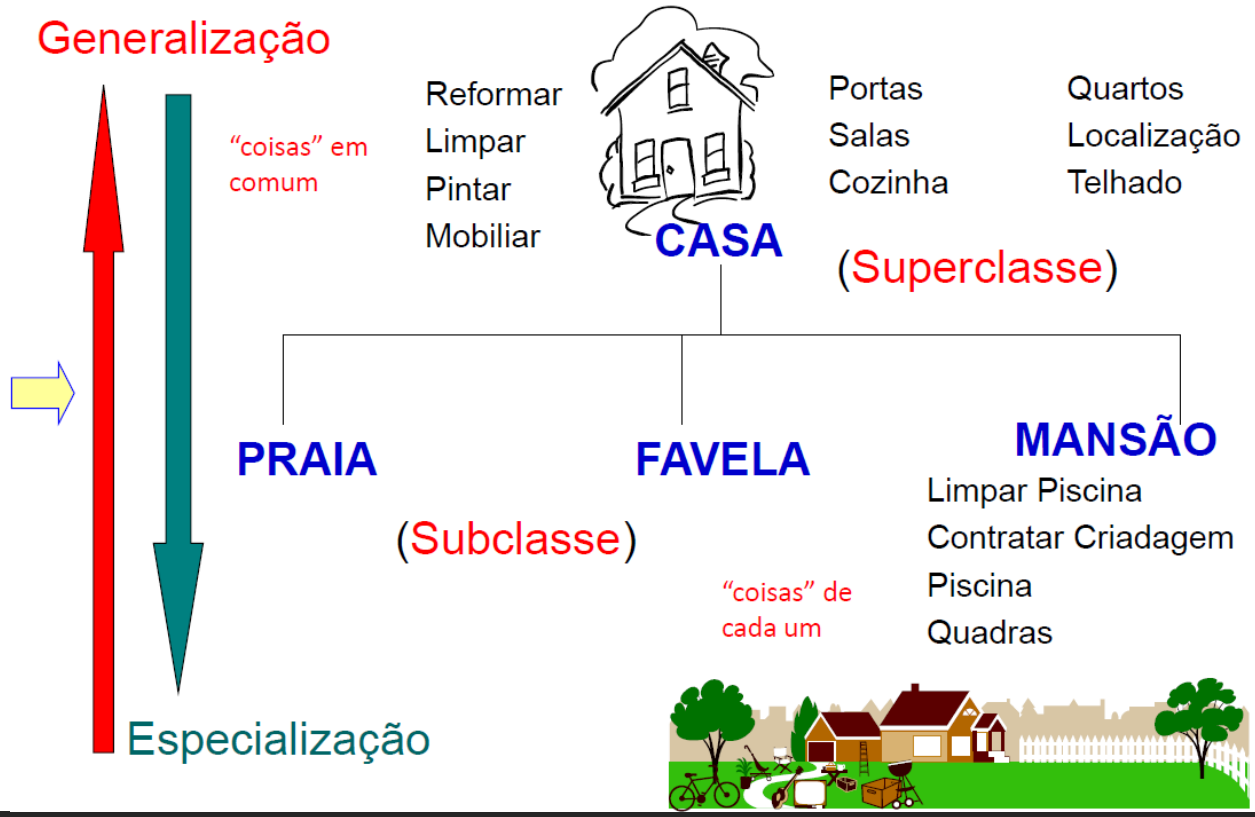
- E o que é herdado em uma classe em OO ?
  - Atributos,
  - Métodos
  - Relacionamentos
- A classe derivada pode ainda:
  - Adicionar atributos, métodos e relacionamentos
  - Redefinir métodos

Composição

- Criação de uma nova classe usando classes existentes como atributos;
- Relacionamento “tem um”: uma conta tem um titular (cliente), um cliente tem um nome (string);
- Vimos como fazer isso anteriormente:
  - Atributos primitivos e referências a objetos;
  - Operadores de seleção;
  - Inicialização (zerar e atribuir valor inicial);
  - O valor null;
  - Atributos estáticos.

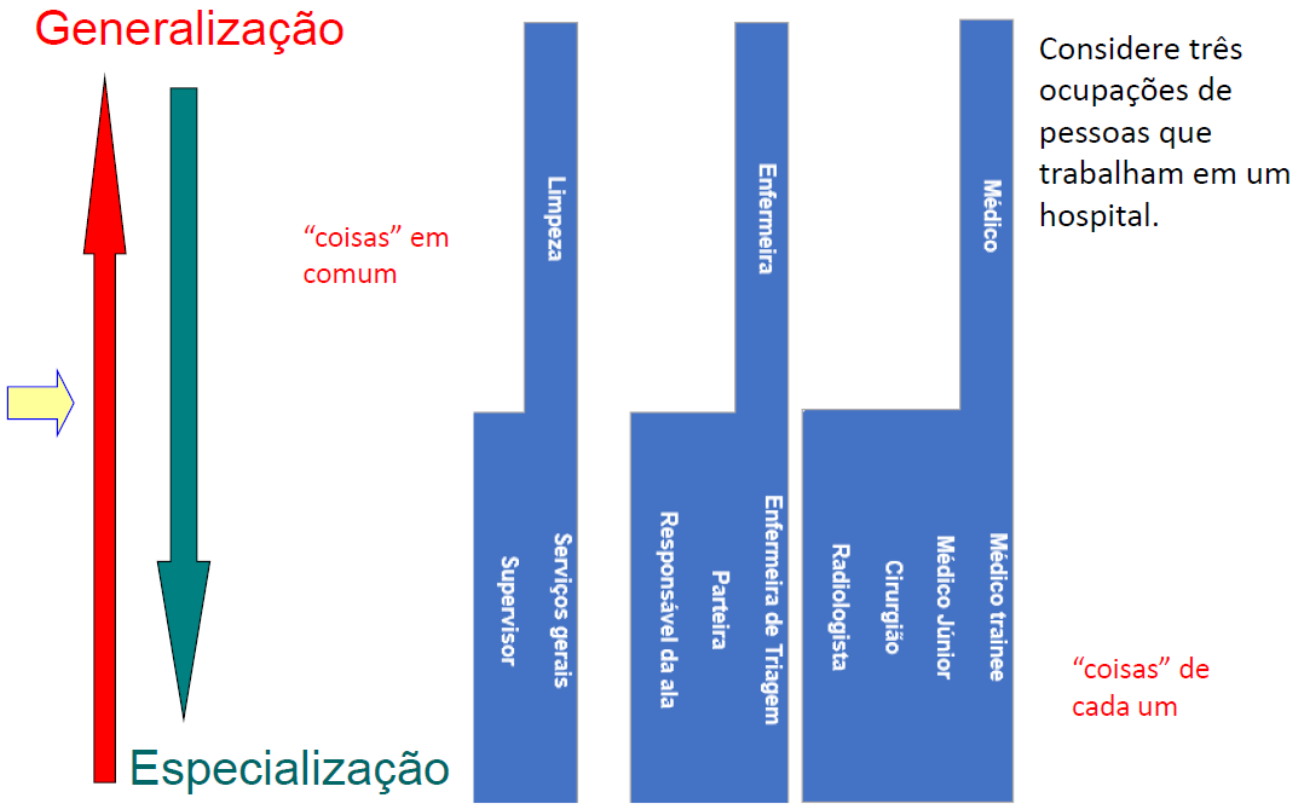
Herança

- Criação de novas classes derivando classes existentes;
- Relacionamento “é um [subtipo de]”: um livro é um produto, um administrador é um usuário;
- Uso da palavra-chave **extends**;
- A palavra-chave é sugestiva – a classe que está sendo criada “estende” outra classe:
  - Partindo do que já existe naquela classe...
  - Pode adicionar novos recursos;
  - Pode redefinir recursos existentes.



Generalização x Especialização

- A outra forma de relacionamento entre as classes que temos em OO é a Herança,
- –É identificado como ("é-um") ou ("é um tipo de")



Generalização x Especialização

- A outra forma de relacionamento entre as classes que temos em OO é a Herança,
- –É identificado como ("é-um") ou ("é um tipo de")

Herança

- Classes com elementos (atributos, métodos) repetidos:

```
class Produto {
    String nome;
    double preco;
    Produto() { } // Precisa?
    public Produto(String nome, double preco) {
        this.nome = nome; this.preco = preco;
    }
    public boolean ehCaro() {
        return (preco > 100);
    }
    // Eventuais outros métodos...
}
```

Herança

- Classes com elementos (atributos, métodos) repetidos:

```
class Livro {
    String nome;
    double preco;
    String autor;
    int paginas;
    public Livro(String n, double p, String a, int pg) {
        nome = n; preco = p; autor = a; paginas = pg;
    }
    public boolean ehCaro() { return (preco > 100); }
    public boolean ehGrande() { return (paginas > 200); }
    // Eventuais outros métodos...
}
```

Motivação

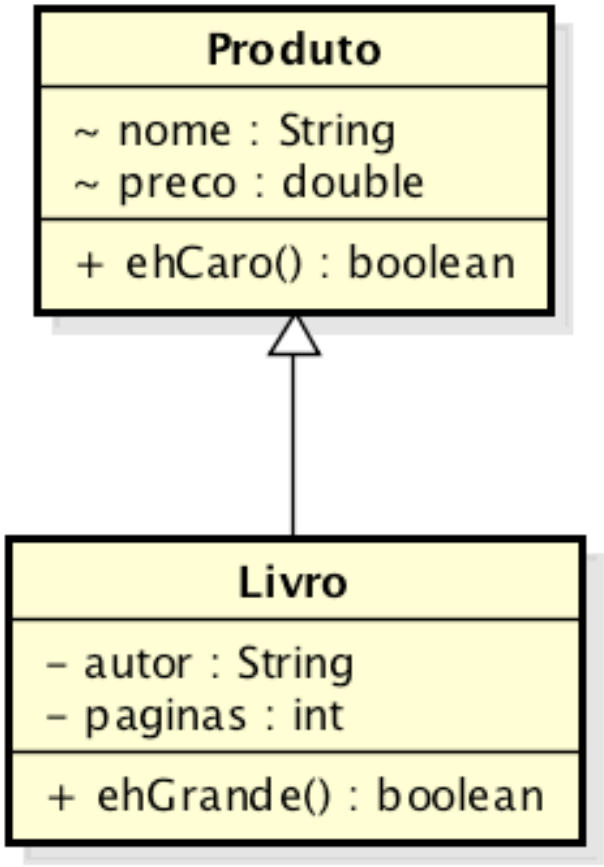
- Código repetido = problema de manutenção;
  - Se surge um novo tipo de produto?
  - Se muda alguma coisa em todos os produtos?
- Colocar os atributos extras em Produto, porém só utilizá-los em objetos que representem livros?
  - Solução confusa, desperdiça memória, ainda mais se a hierarquia crescer (CDs, DVDs, eletrônicos, etc.);
- Usar composição?
  - Também causa confusão. Um livro tem um produto ou um livro é um produto?
- Solução OO: herança!

Solução com herança

- Livro estende produto (adiciona novos membros):

```
class Livro extends Produto {
    //private String nome; // Não preciso repetir.
    //private double preco; // Herdo de Produto
    private String autor;
    private int paginas;
    public Livro(String n, double p, String a, int pg) {
        nome = n; preco = p; autor = a; paginas = pg;
    }
    // Também não preciso repetir:
    // public boolean ehCaro() { return (preco > 100); }
    public boolean ehGrande() { return (paginas > 200); }
    // Eventuais outros métodos...
}
```

Herança em UML



Solução com herança

- Podemos chamar métodos do Produto no Livro:

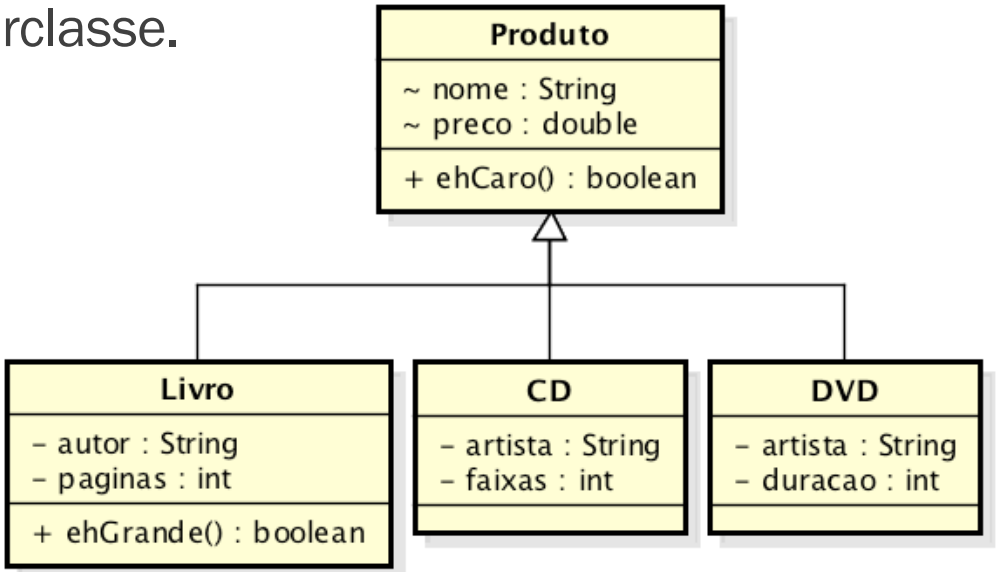
```
public class Loja {
    public static void main(String[] args) {
        Livro l = new Livro("Linguagens de Programação",
                74.90, "Flávio Varejão", 334);
        System.out.println(l.ehCaro());
        System.out.println(l.ehGrande());
    }
}
```

Produto (Superclasse)
- Classe base;
- Classe pai/mãe;
- Classe ancestral, etc.

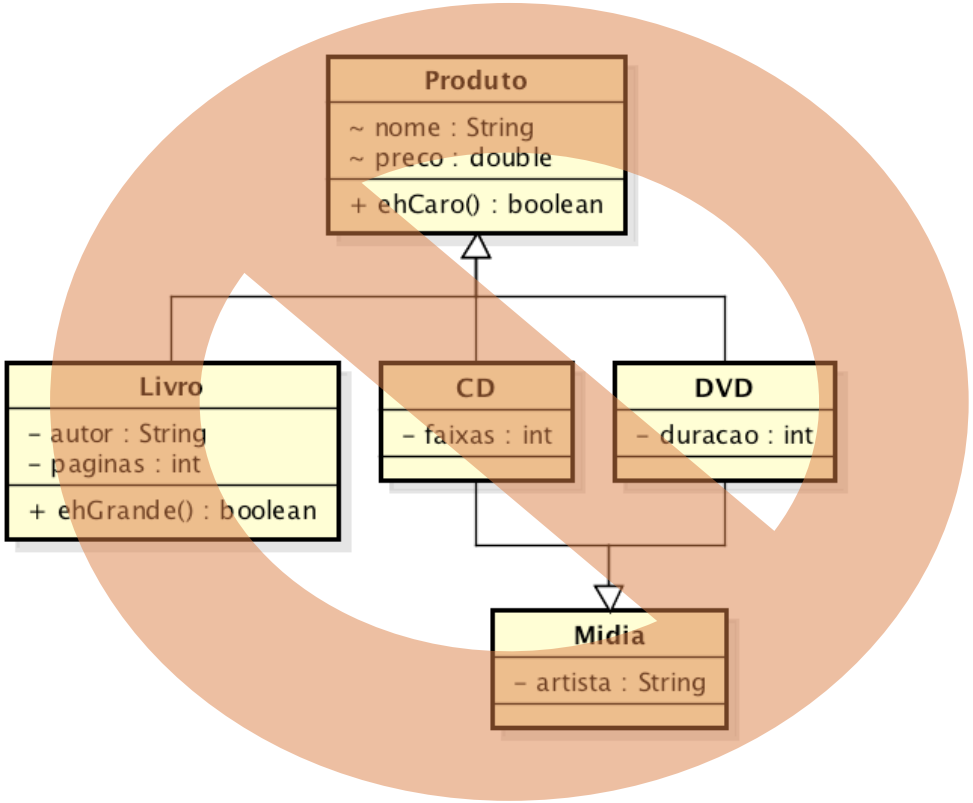
Livro (Subclasse)
- Classe derivada;
- Classe filha;
- Classe descendente, etc.

Java suporta herança simples

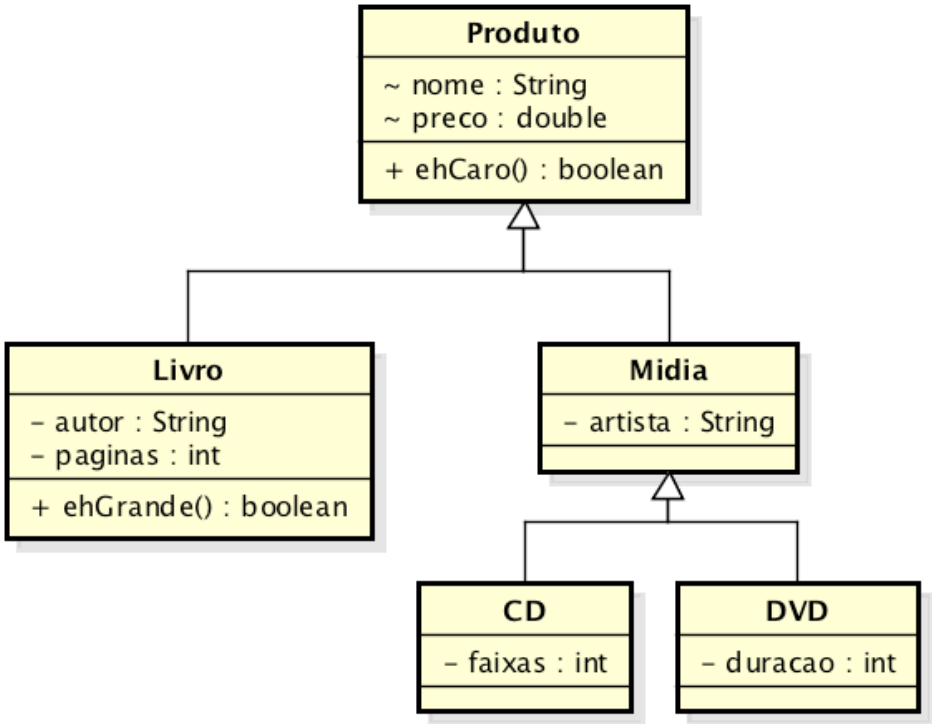
- Uma classe pode ter muitas subclases;
- Uma classe só pode ter uma superclasse.



Java não suporta herança múltipla



Hierarquias de qualquer tamanho



Sintaxe

- Sintaxe:

```
class Subclasse extends Superclasse {
        /* ... */
}
```

- Semântica:
  - A subclasse herda todos os atributos e métodos que a superclasse possuir;
  - Subclasse é uma derivação, um subtipo, uma extensão da superclasse.

Subclasses herdam membros

- Livro possui autor e paginas (definidos na própria classe);
- Livro possui nome e preco (definidos na superclasse);
- Livro pode receber mensagens ehGrande() (definida na própria classe);
- Livro pode receber mensagens ehCaro() (definida na superclasse).

Visibilidade dos atributos herdados

- Acesso privativo é só para a própria classe:

```
class Produto {
    private String nome;
    private double preco;
    // Restante da classe...
}
class Livro extends Produto {
    private String autor;
    private int paginas;
    public Livro(String n, double p, String a, int pg) {
        nome = n; preco = p; autor = a; paginas = pg;
    }
    // Restante da classe...
}
```

Relembrando modificadores de acesso

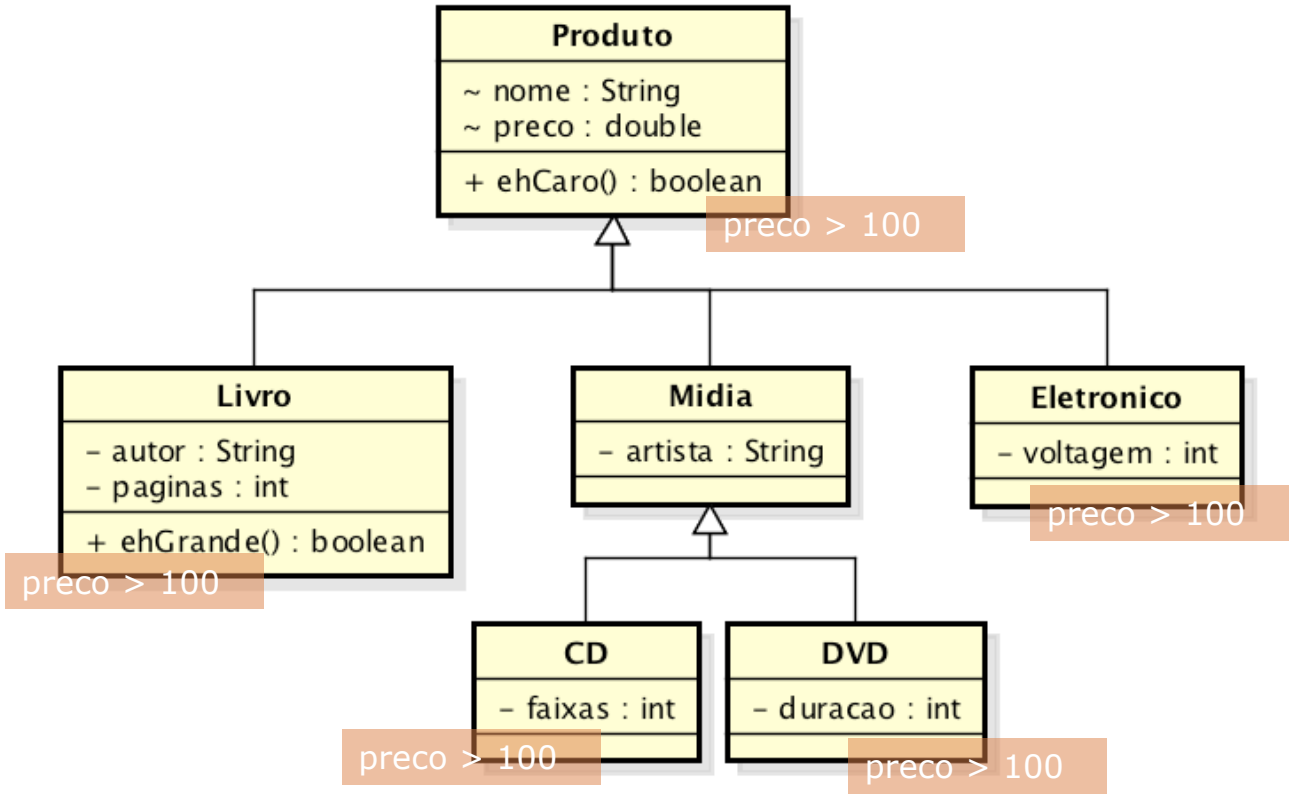
Acesso	Público	Protegido	Amigo	Privado
A própria classe	Sim	Sim	Sim	Sim
Classe no mesmo pacote	Sim	Sim	Sim	Não
Subclasse em pacote diferente	Sim	Sim	Não	Não
Não-subclasse em pacote diferente	Sim	Não	Não	Não

Herança

- Custos da Herança
  - Velocidade da execução
    - Necessário identificar qual método no nível hierárquico que está se referindo
- Overhead de Mensagem
  - Há uma maior troca de mensagens devido aos níveis hierárquicos
- Isto não significa que não deve-se usar a herança,
  - Mas que deve-se compreender melhor os benefícios, e pesá-los em relação aos custos.

Reescrita/sobrescrita de método

- Um método herdado pode não fazer total sentido:



Herança

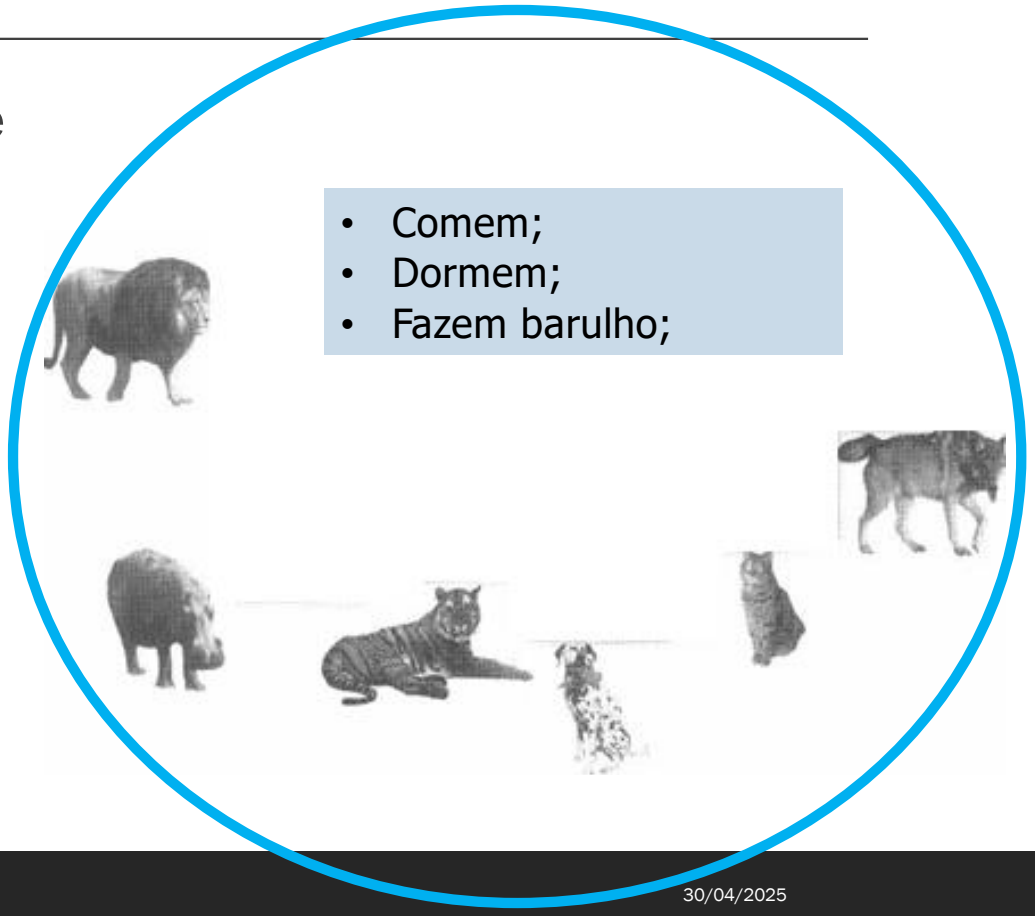
Seja o seguinte Contexto:

- Criar um programa para simular comportamento de vários animais em um ambiente;
- Possui conjunto de animais (não todos);
- Cada animal => 1 objeto;
- Cada animal move-se no ambiente a seu modo; fazendo qualquer coisa;
  - Ou seja, tem seu próprio comportamento também..
- Novos animais podem ser adicionados ao programa.

Herança

Passos para definir o relacionamento de Herança:

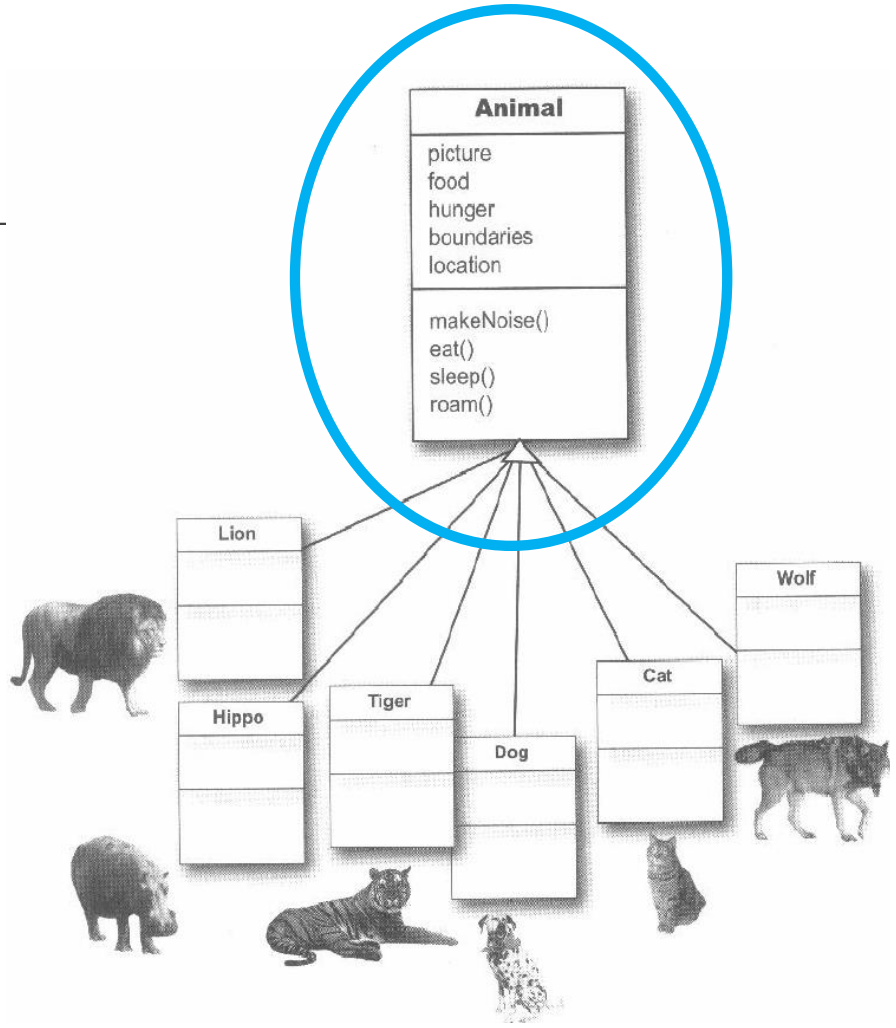
- 1º Passo:
  - Definir o que cada objeto animal tem em comum, como Atributos e Comportamentos;
  - Definir como esses tipos de animais se relacionam



Herança

Passos para definir o relacionamento de Herança:

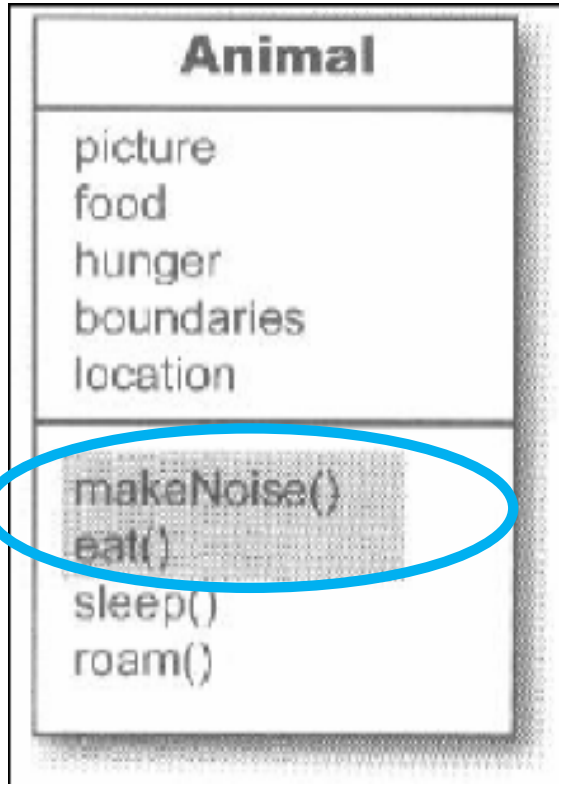
- 2º Passo:
  - Projetar a classe que representa o estado e comportamento em comum.
  - Superclasse.



Herança

Passos para definir o relacionamento de Herança:

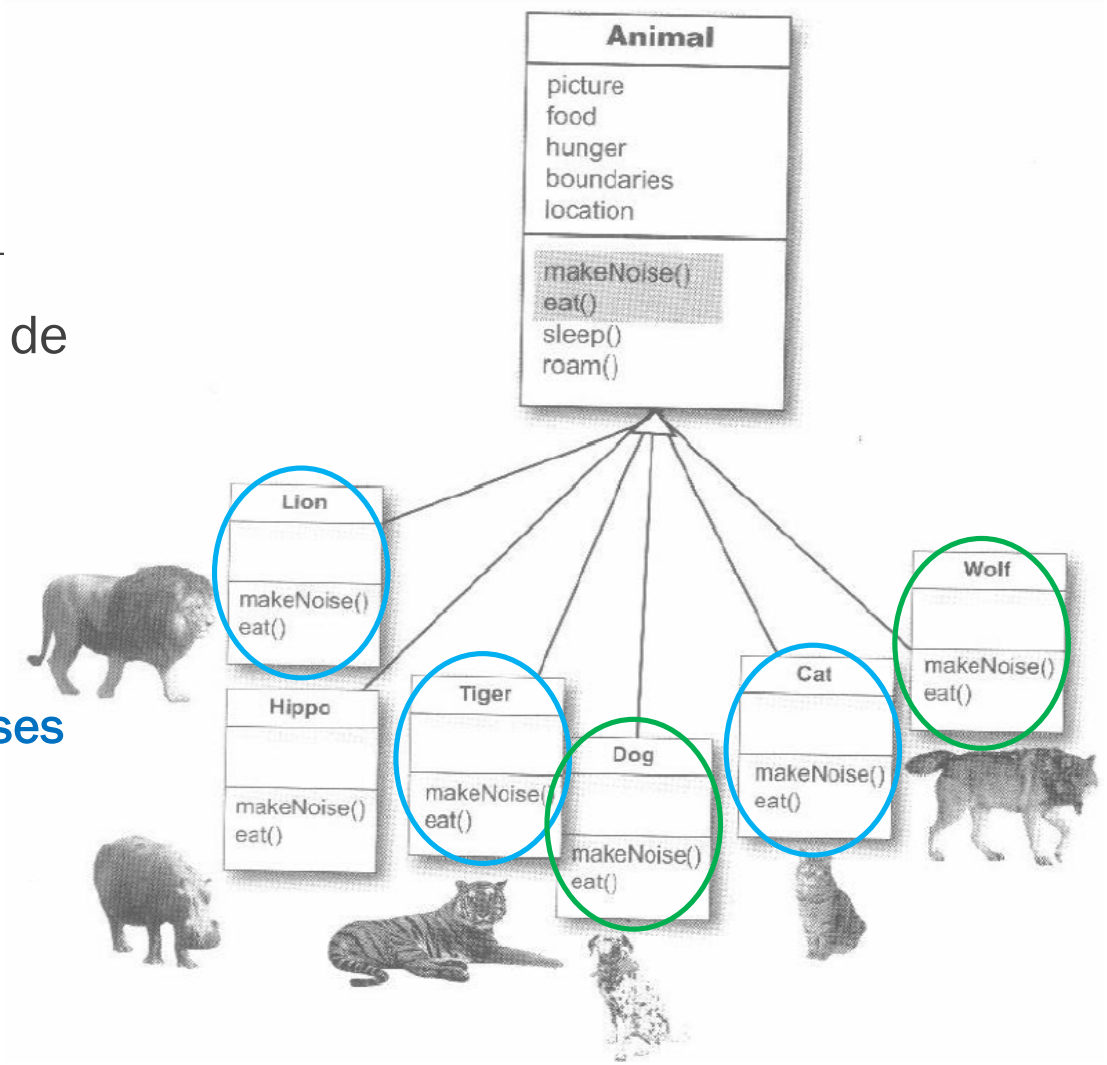
- 3º Passo:
  - Decidir **se a subclasse necessita de comportamentos** (métodos) que são **específicos** do tipo particular da subclasse.



Herança

Passos para definir o relacionamento de Herança:

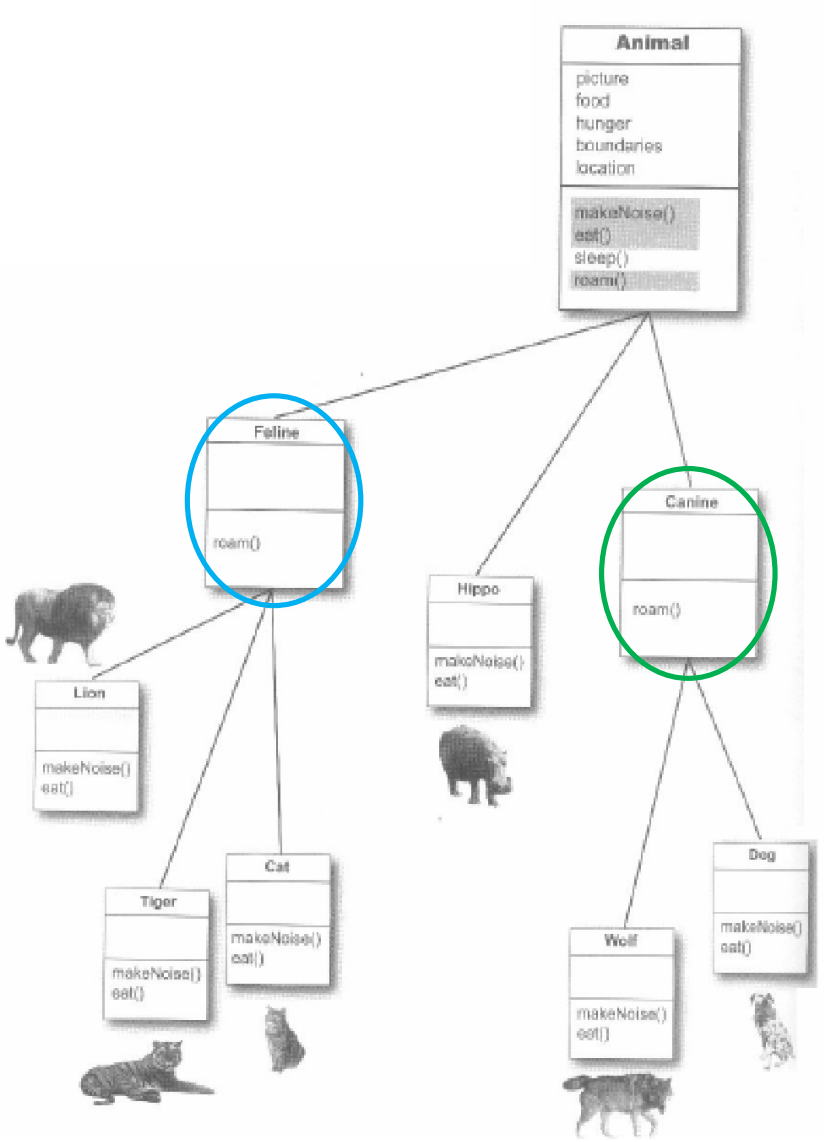
- 4º Passo:
  - Procurar mais oportunidades para se usar a abstração.
  - Ou seja, **encontrar** duas ou **mais classes** que tenham **comportamento em comum.**



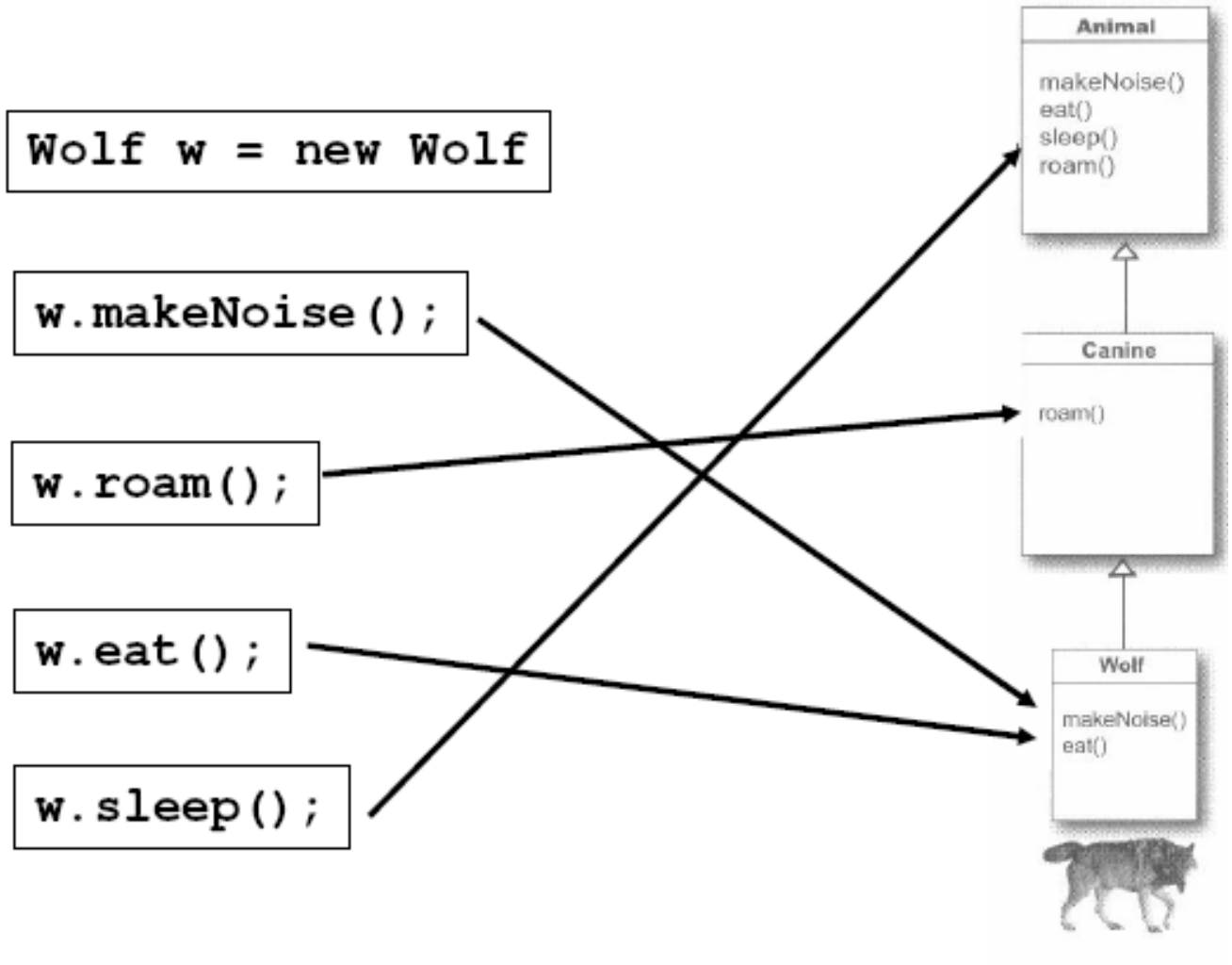
Herança

Passos para definir o relacionamento de Herança:

- 5º Passo:
  - Finalizar a **Hierarquia de classes.**



Herança


Wolf w = new Wolf
w.makeNoise();
w.roam();
w.eat();
w.sleep();
Animal
makeNoise()
eat()
sleep()
roam()
Canine
roam()
Wolf
makeNoise()
eat()

Heraça

- Sintaxe em Java:

```
class Subclasse extends Superclasse {
}
```

- Semântica:
  - Podemos declarar um membro que, embora não seja acessível por outras classes, é herdado por suas sub-classes
    - Para isso usamos o modificador de controle de acesso PROTECTED
  - A visibilidade na herança seria:
    - private: Membros são vistos só pela própria classe e não são herdados por nenhuma outra
    - protected: Membros são vistos pelas classes do pacote e herdados por qualquer outra classe
    - public: membros são vistos e herdados por qualquer classe

Polimorfismo

- Sobrecarregando (Overloading)
  - Pode-se sobrecarregar um método de uma classe, criando-se assim diversas maneiras de invocá-lo.
  - Ou seja, é possível definir dois ou mais métodos com o mesmo nome, porém com assinaturas diferentes;
  - **O método deve ter assinatura DIFERENTE pode ser: número ou tipos de parâmetros diferentes.**
- Sobrescrevendo (Overriding)
  - Pode-se sobrescrever um método de mesmo nome numa classe derivada.
  - Uma classe filha pode fornecer uma outra implementação para um método herdado, caracterizando uma redefinição do método
  - Esta construção é utilizada quando a classe derivada age diferente de sua classe pai.
  - **O método deve ter a MESMA assinatura (nome, argumentos), senão não se trata de uma redefinição e sim, de sobrecarga.**

Polimorfismo

- Sobrescrever um método de uma superclasse com um particular comportamento na classe derivada
  - Se um método herdado não satisfaz, podemos redefini-lo (sobrescrevê-lo)

Polimorfismo

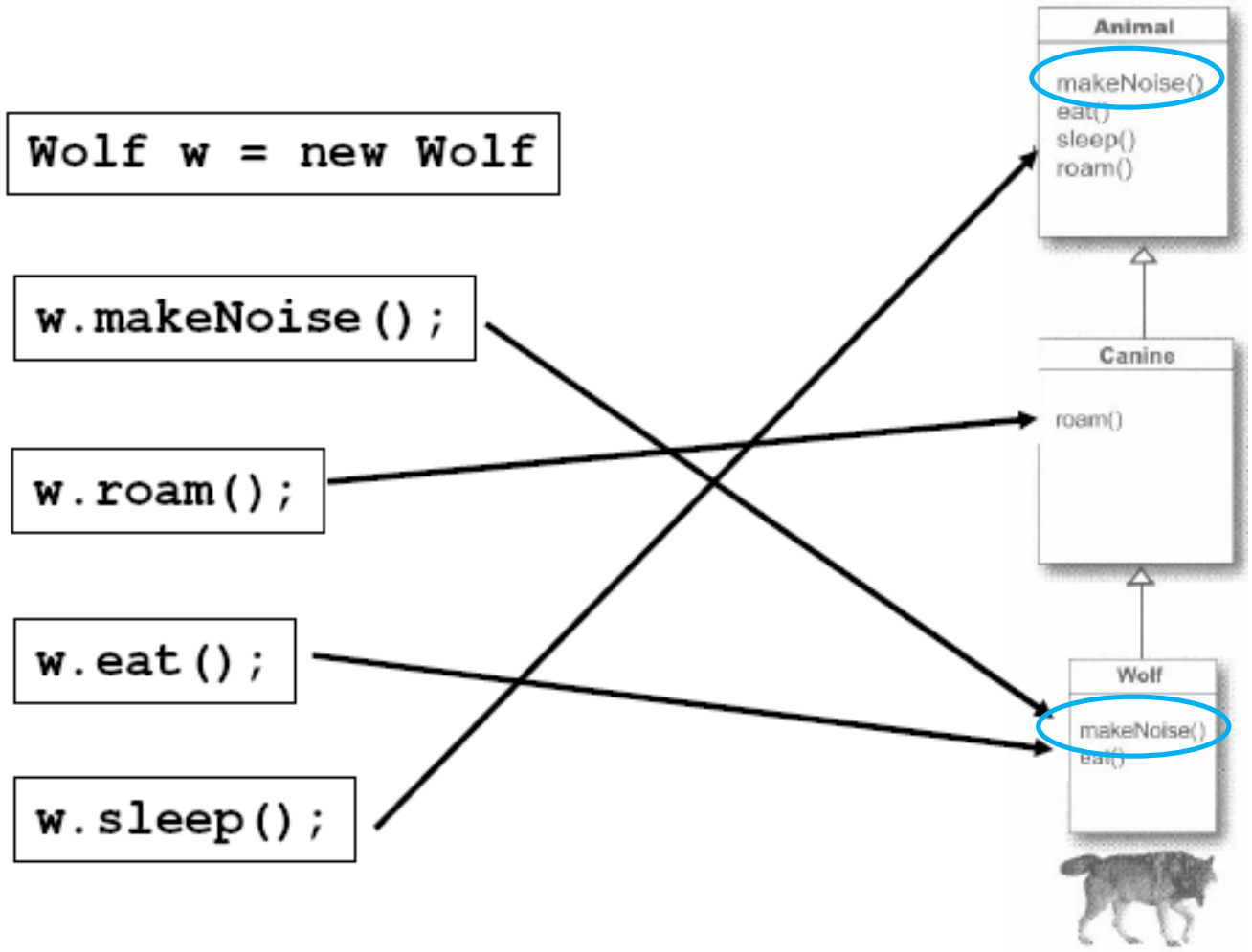
O polimorfismo pode ser útil em muitos casos.

Por exemplo, poderíamos criar um jogo em que teríamos diferentes tipos de jogadores, com cada jogador tendo um comportamento separado para o método Ataque.

Nesse caso, o ataque seria um método virtual da classe base Jogador e cada classe derivada o substituiria.

Herança

- Qual método é chamado????



Sobrescrita x Sobrecarga

```
class Forma {
   public void aumentar(int t) {
      System.out.println("Forma.aumentar()");
   }
}
class Linha extends Forma {
   // Foi feita sobregarga e não sobrescrita!
   public void aumentar(double t) {
      System.out.println("Linha.aumentar()");
   }
}
```

Herança em Java - Final

- O identificador **final** além de:
  - Poder ser utilizado para definir constantes
  - É utilizado para identificar uma classe que não pode ser subclassificada.
    - Por consequência, todos os métodos de uma classe final são automaticamente finais.

Classe que não pode ser subclassificada

```
class Telefone { }

final class TelefoneCelular extends Telefone
{ }

// Erro: TelefoneCelular é final!
class TelefoneAtomico extends TelefoneCelular
{ }
```

Métodos e Classes Abstratas

- Abstração: Capacidade de determinar o problema de maneira geral, dando importância apenas aos seus aspectos relevantes e ignorando os detalhes
- Ao criarmos uma classe para ser estendida, às vezes codificamos alguns métodos para os quais não sabemos dar uma implementação.
  - Um método que só subclasses saberão implementar
- Uma **classe** deste tipo **NÃO** pode ser instanciada pois sua funcionalidade está incompleta.

Métodos e Classes Abstratas

- Uma classe abstrata é utilizada quando deseja-se:
  - Fornecer uma interface comum a diversos membros de uma hierarquia de classes.
  - Não se preocupar COMO, apenas QUAL o contrato que deve ser obedecido.

Métodos e Classes Abstratas

- Classes abstratas:
  - Podemos declarar uma classe abstrata usando o modificador **abstract**
- Métodos abstratos:
  - Usamos o modificador **abstract** e **omitimos** a implementação desse método.

Método sem corpo { }

```
abstract class Veiculo
{
    abstract void Abastece ();
}

class Onibus extends Veiculo
{
    void Abastece ()
    {
        EncheTanqueComOleoDiesel();
    }
}

...
```

Todos os veículos possuem a ação de abastecer, mas a forma de abastecer (álcool, gasolina ou diesel) só é conhecida pelas filhas da herança

```
Veiculo v = new Veiculo();   // Erro!
Veiculo v;                   // Ok
v = new Onibus;              // Ok
```

Runtime Type Information (RTTI)

- É possível verificar o tipo de um objeto em tempo de execução usando o operador instanceof
- Sintaxe: <objeto> instanceof <Classe>
  - Retorna true se o objeto for instância (direta ou indireta) da classe especificada; Retorna false caso contrário.


```
1  abstract class Animal {
2      public abstract void comer();
3  }
4  class Cachorro extends Animal {
5      public void comer() {
6          System.out.println("Comendo um osso...");
7      }
8
9      public void latir() {
10         System.out.println("Au Au!");
11     }
12 }
13
14 class Gato extends Animal {
15     public void comer() {
16         System.out.println("Comendo um peixe...");
17     }
18
19     public void miar() {
20         System.out.println("Miau!");
21     }
22 }
23
24 public class AnimalTesteinstanceOf {
25     public static void main(String[] args) {
26         Animal[] vet = new Animal[] {
27                         new Cachorro(), new Gato(),
28                         new Gato(), new Cachorro()
29                     };
30
31         /*
32         for (int i = 0; i < vet.length; i++) {
33             vet[i].comer();
34             // Erro:
35             vet[i].latir();
36         }
37         */
38
39         for (int i = 0; i < vet.length; i++) {
40             if (vet[i] instanceof Cachorro)
41                 ((Cachorro)vet[i]).latir();
42             else if (vet[i] instanceof Gato)
43                 ((Gato)vet[i]).miar();
44         }
45     }
46 }
```

Runtime Type Information (RTTI)

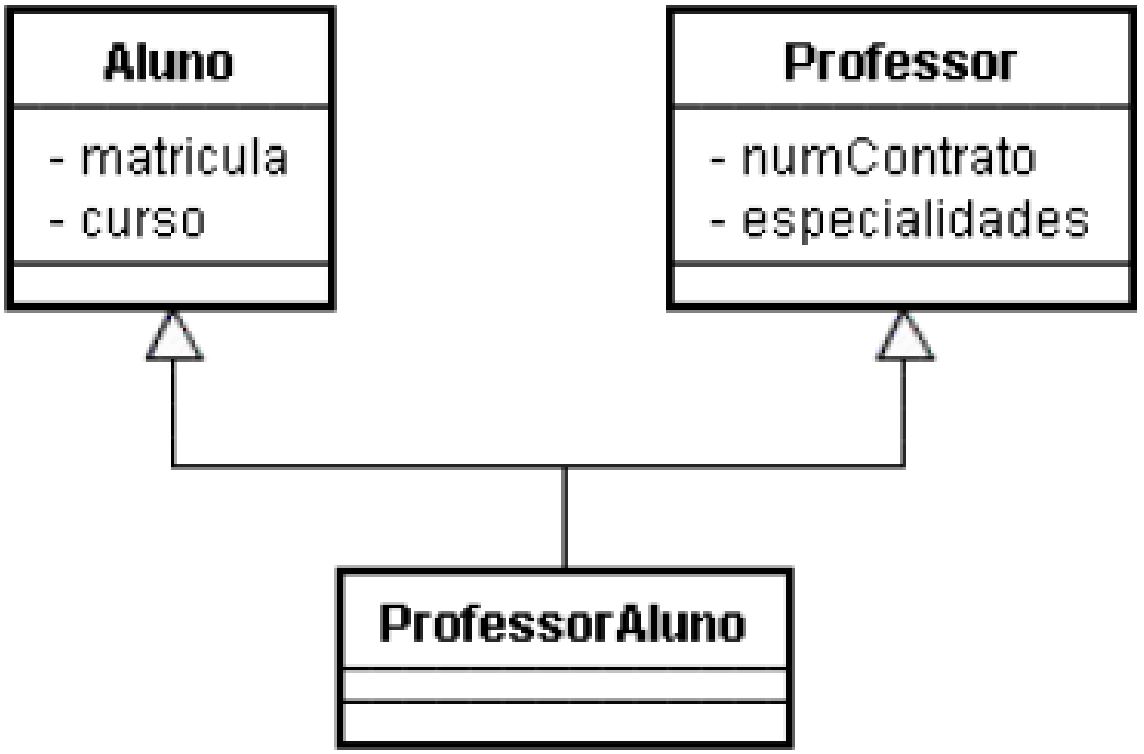
- É possível verificar o tipo de um objeto em tempo de execução usando o operador instanceof
- Sintaxe: <objeto> instanceof <Classe>
  - Retorna true se o objeto for instância (direta ou indireta) da classe especificada; Retorna false caso contrário.


```
1  import java.util.*;
2
3  abstract class Animal {
4      public abstract void comer();
5  }
6  class Cachorro extends Animal {
7      public void comer() {
8          System.out.println("Comendo um osso...");
9      }
10
11     public void latir() {
12         System.out.println("Au Au!");
13     }
14 }
15 class Gato extends Animal {
16     public void comer() {
17         System.out.println("Comendo um peixe...");
18     }
19
20     public void miar() {
21         System.out.println("Miau!");
22     }
23 }
24 public class AnimalTesteGenerics {
25     public static void main(String[] args) {
26
27         ArrayList<Animal> vet = new ArrayList<Animal> ();
28
29         vet.add(new Cachorro());
30         vet.add(new Gato());
31         vet.add(new Gato());
32         vet.add(new Cachorro());
33
34         for(Animal a : vet){
35             a.comer();
36
37             if (a instanceof Cachorro)
38                 ((Cachorro)a).latir();
39             else if (a instanceof Gato)
40                 ((Gato)a).miar();
41         }
42     }
43 }
```

Herança Múltipla

- Em algumas linguagens de programação OO as classes podem ter mais de um superclasse.
  - Herdando variáveis e métodos combinados de todas essas superclasses. Isto é chamado de herança múltipla.
- Herança Múltipla fornece margem para praticamente qualquer comportamento imaginável.
  - Contudo complica bastante as definições de classes e o código para produzi-la, além do entendimento.

Herança Múltipla



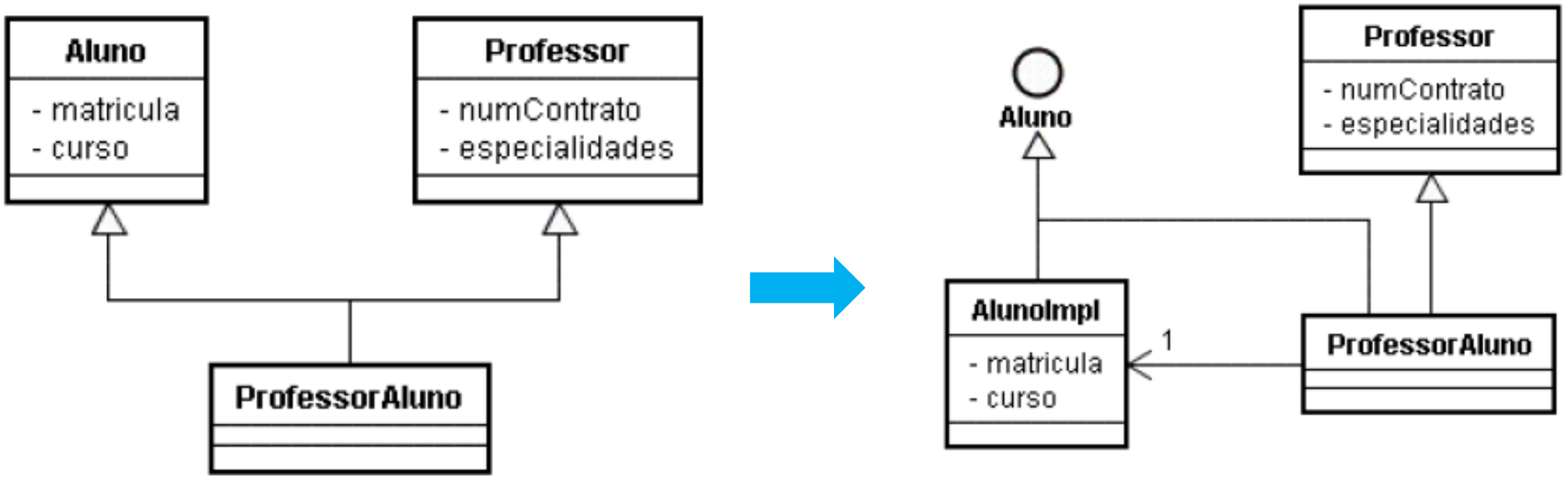
Herança Múltipla

- Java simplifica a herança.
- Permitindo diretamente apenas a herança simples ou única das classes
- Java NÃO permite múltipla herança de classe.
- Verificado em outras linguagens que sua implementação pode ser muito confusa.

Interface

- Em Java, os objetos podem implementar uma INTERFACE.
- As interfaces são como classes, mas não têm nenhuma implementação.
- Define apenas as assinaturas de operações externamente visíveis que uma classe pode implementar, sem conter nenhuma especificação ou estrutura interna
- Na prática representa:
  - Um comportamento compartilhado por várias classes
  - Onde cada classe deve implementar os métodos

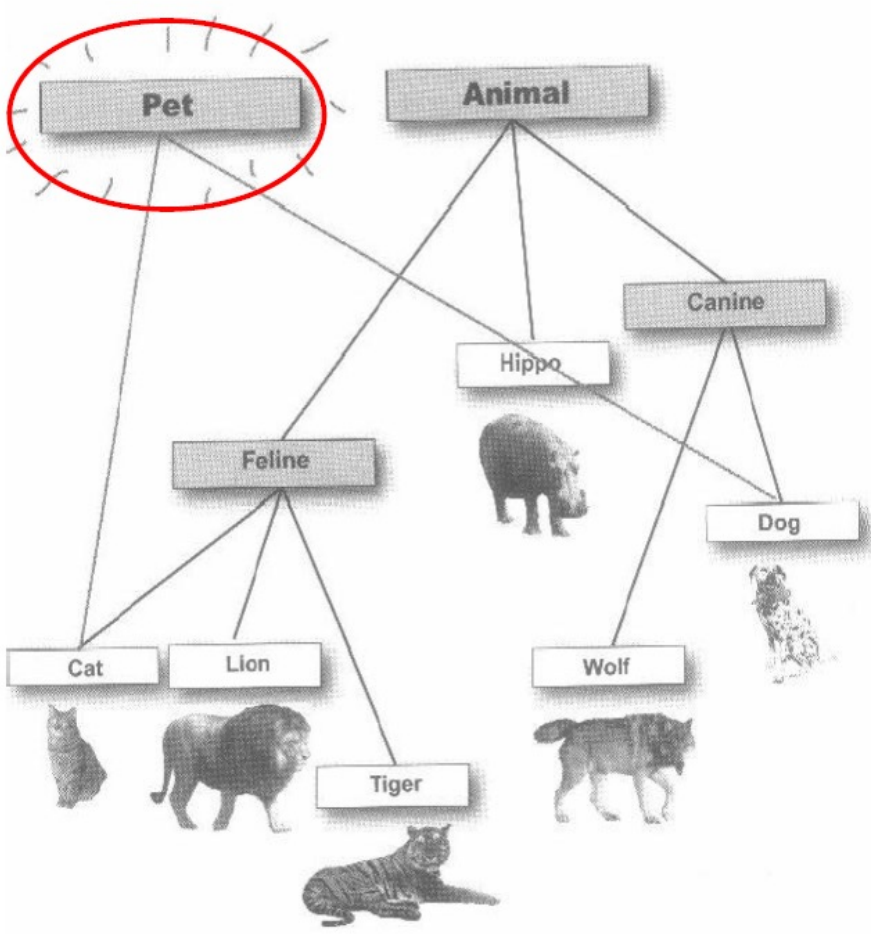
Interface


Aluno
- matricula
- curso
Professor
- numContrato
- especialidades
ProfessorAluno
Aluno
AlunoImpl
- matricula
- curso
1
Professor
- numContrato
- especialidades
ProfessorAluno

Interface

Opção: Novo recurso com comportamentos de animais de estimação

Uma classe pode herdar de apenas uma classe base, mas pode implementar várias interfaces



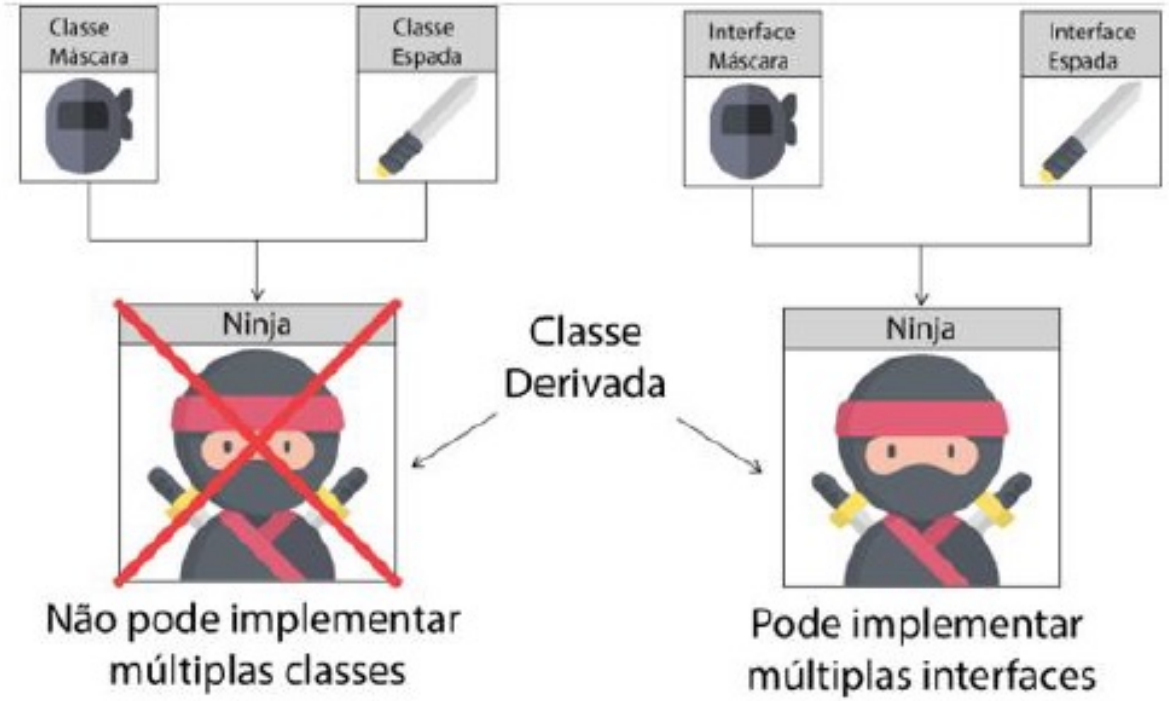
Java - Class x Interface

```
public class MinhaClasse
{
        <construtor, métodos, atributos>
}
```

```
public interface MinhaInterface
{
    <métodos sem implementação, atributos constantes>
}
```

Interfaces não permitem declaração de atributos variantes (somente estáticos), enquanto que classes permitem outros.

Atenção!


Classe
Máscara
Classe
Espada
Interface
Máscara
Interface
Espada
Ninja
Classe
Derivada
Ninja
Não pode implementar
múltiplas classes
Pode implementar
múltiplas interfaces