Universidade Federal da Paraíba

Centro de Informática

Departamento de Informática

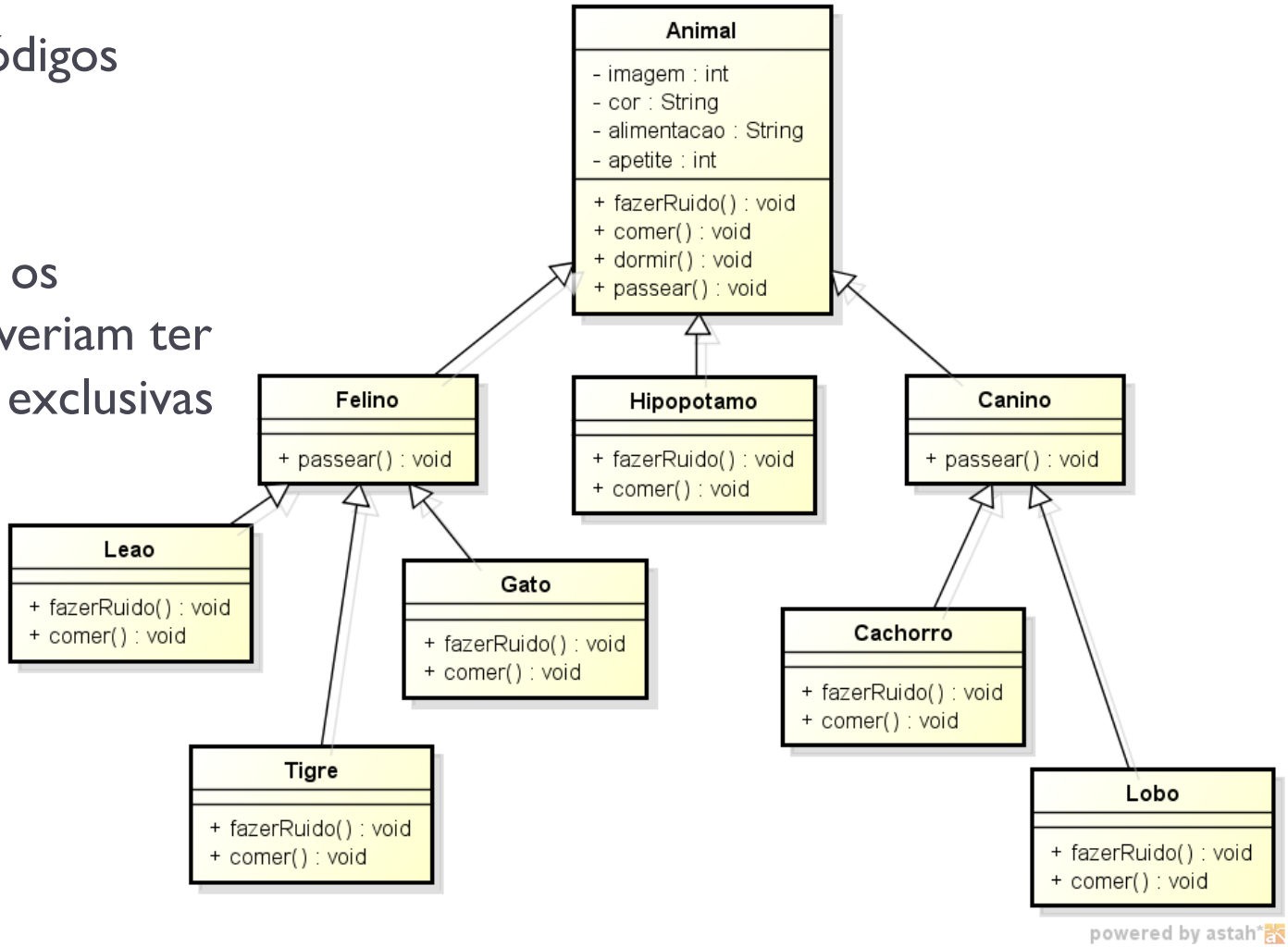
Linguagem de Programação I

Interfaces e Classes Abstratas

- Tiago Maritan
- tiago@ci.ufpb.br

Motivação

- Esse projeto está bom?
  - Reduzimos os códigos duplicados;
  - Sobrescrevemos os métodos que deveriam ter implementações exclusivas nas subclasses



Motivação

- Vimos na aula passada que podemos fazer:

```
Lobo aLobo = new Lobo(); // ou então

Animal aHipo = new Hipopotamo();
```

- Mas tem algo que fica estranho:

```
Animal anim = new Animal();
```

Qual é a aparência desse objeto Animal?

Qual é a cor, tamanho, quantidade de pernas?

Classes Abstratas

- Precisamos de uma classe `Animal`, devido a herança e polimorfismo.
- Queremos objetos `Tigre`, `Lobo`, `Cachorro` e não objetos `Animal`.
  - Ou seja, queremos que apenas as subclasses de `Animal` sejam instanciadas
- Em Java, existe uma forma de impedir que uma classe seja instanciada
  - Ou seja, impedir que alguém use o "**`new`**" com esse tipo
- Para isso, usamos a palavra reservada **`abstract`**
  - Cria um tipo (ou uma classe) abstrato(a)
  - Compilador impede que se crie uma instância desse tipo
  - Pode ser usada para fins de polimorfismo

Classes Abstratas

- Ex: Criando classe abstrata...

```
public abstract class Canino extends Animal{
    public void passear(){}
}
```

- Tentando referenciar...

```
public class FabricaCaninos {
    public void criar(){
        Canino c;
        c = new Canino();    // erro! Canino é abstrato não
                             // pode ser instanciado

        c = new Cachorro(); // ok! pode-se atribuir um objeto
                            // da subclasse a superclasse...
                            // e classe Cachorro é concreta;
    }
}
```

Classes Abstratas

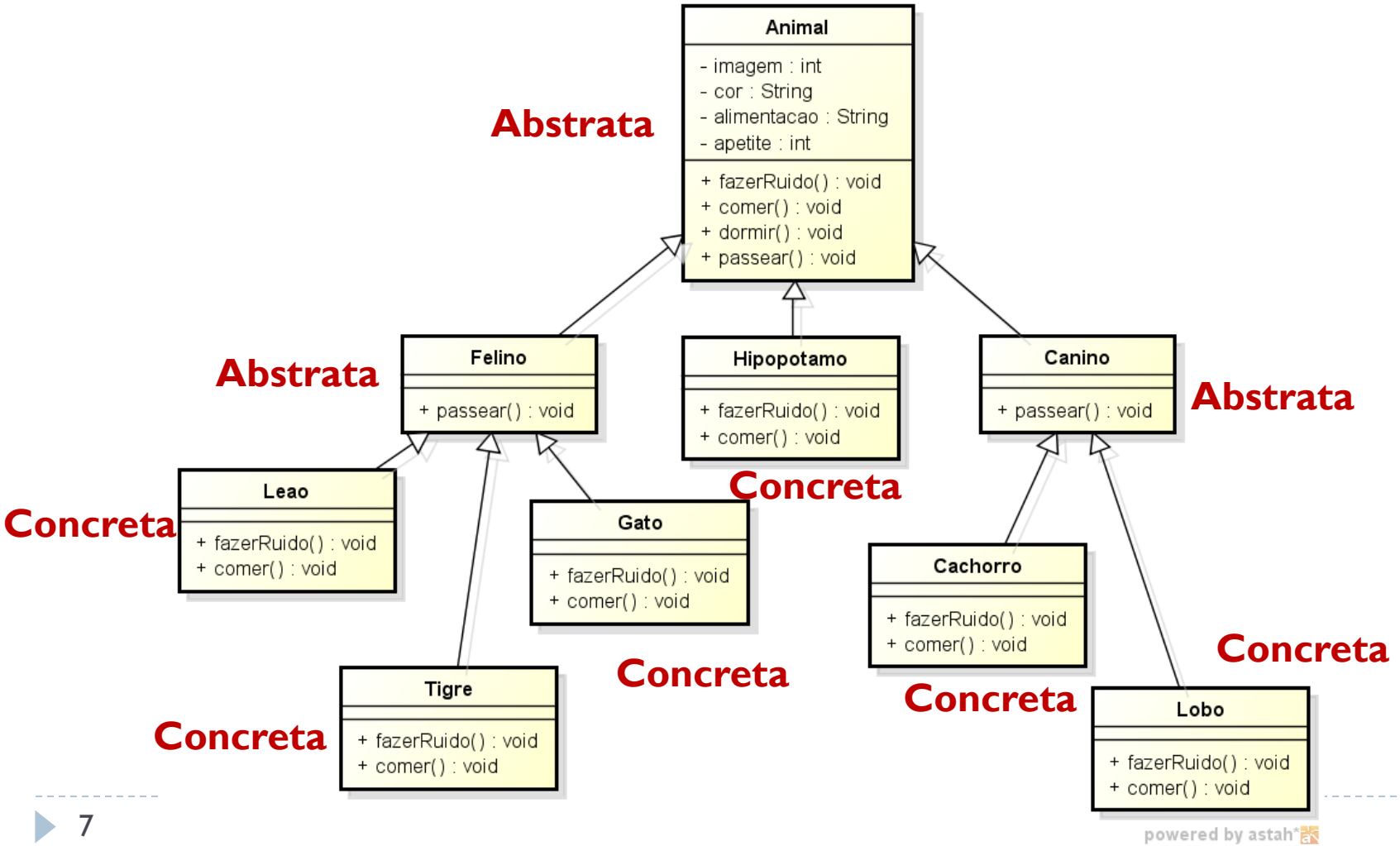
- Quando for projetar suas classes, você deve decidir quais classes serão **concretas** e quais serão **abstratas**.

- Geralmente, **classes abstratas** só tem utilidade se forem estendidas

- API Java tem várias classes abstratas
  - Ex: Pacote **Swing**
    - `Component` é abstrata
    - `Jbutton` estende `Component` e é concreta

Classes Abstratas

- Então, podemos melhorar nosso projeto:



Métodos Abstratos

- Além das classes, os métodos também pode ser abstratos
  - Uma classe abstrata significa que ela deve ser estendida;
  - Um método abstrato significa que ele deve ser sobreposto;

- Em geral, um **método é abstrato** quando não é possível gerar um método genérico para todas as subclasses.
  - Ex: não há como gerar um método genérico `fazerRuido()` para todas as subclasses de `Animal`

- Sintaxe:

```
public abstract void fazerRuido();
```

Método não tem corpo...
Termina com ponto e vírgula

Métodos Abstratos

- Uma **classe abstrata** deve ter ao menos um **método abstrato**
  - Mas também pode ter **métodos concretos**

- Se algum método for abstrato, a classe também DEVE ser abstrata

- Todos os métodos abstratos DEVEM ser implementados nas subclasses;
  - Criar um método não-abstrato com a mesma assinatura e um corpo;
  - Equivalente a sobrescrever o método;

Exemplo:

```
public abstract class Animal{

    public abstract void fazerRuido();
    public abstract void comer();

    public void dormir(){
        System.out.pritln("Dormir");
   }
}
```

```
public abstract class Canino extends Animal{
        public abstract void passear();
}
```

Exemplo

- Implementando subclasse `Cachorro`
  - Implementa os métodos abstratos da superclasse

```
public class Cachorro extends Animal {

    public void fazerRuido(){
        System.out.println("Latir");
    }
    public void comer(){
        System.out.println("Comer ração");
    }
}
```

- Considere agora que você deseja adicionar comportamentos de animais domésticos em alguns dos seus animais.
  - Ex: métodos `serAmigavel()` e `brincar()` em Cachorro, Gato...

- Como fazer isto?
  - **Opção 1**: Inserir esses métodos na classe Animal
  - **Opção 2**: Inserir esses métodos como abstratos na classe Animal
  - **Opção 3:** Inserir os métodos de Pet apenas nas subclasses que pertencem (ex.: Cachorro e Gato)

- Considere agora que você deseja adicionar comportamentos de animais domésticos em alguns dos seus animais.
  - Ex: métodos `serAmigavel()` e `brincar()` em Cachorro, Gato...

- Como fazer isto?
  - **Opção 1**: Inserir esses métodos na classe Animal
    - **Vantagem:** Todos os animais herdarão esse comportamento;
    - **Desvantagem:** Teremos Leões e Lobos domésticos;

- Considere agora que você deseja adicionar comportamentos de animais domésticos em alguns dos seus anmais.
  - Ex: métodos `serAmigavel()` e `brincar()` em Cachorro, Gato...

- Como fazer isto?
  - **Opção 2**: Inserir esses métodos como abstratos na classe Animal
    - **Vantagem**: Igual a 1;
    - **Desvantagem**: Animais Selvagem teriam uma implementação inútil e um **contrato** que define que eles tem essas características.

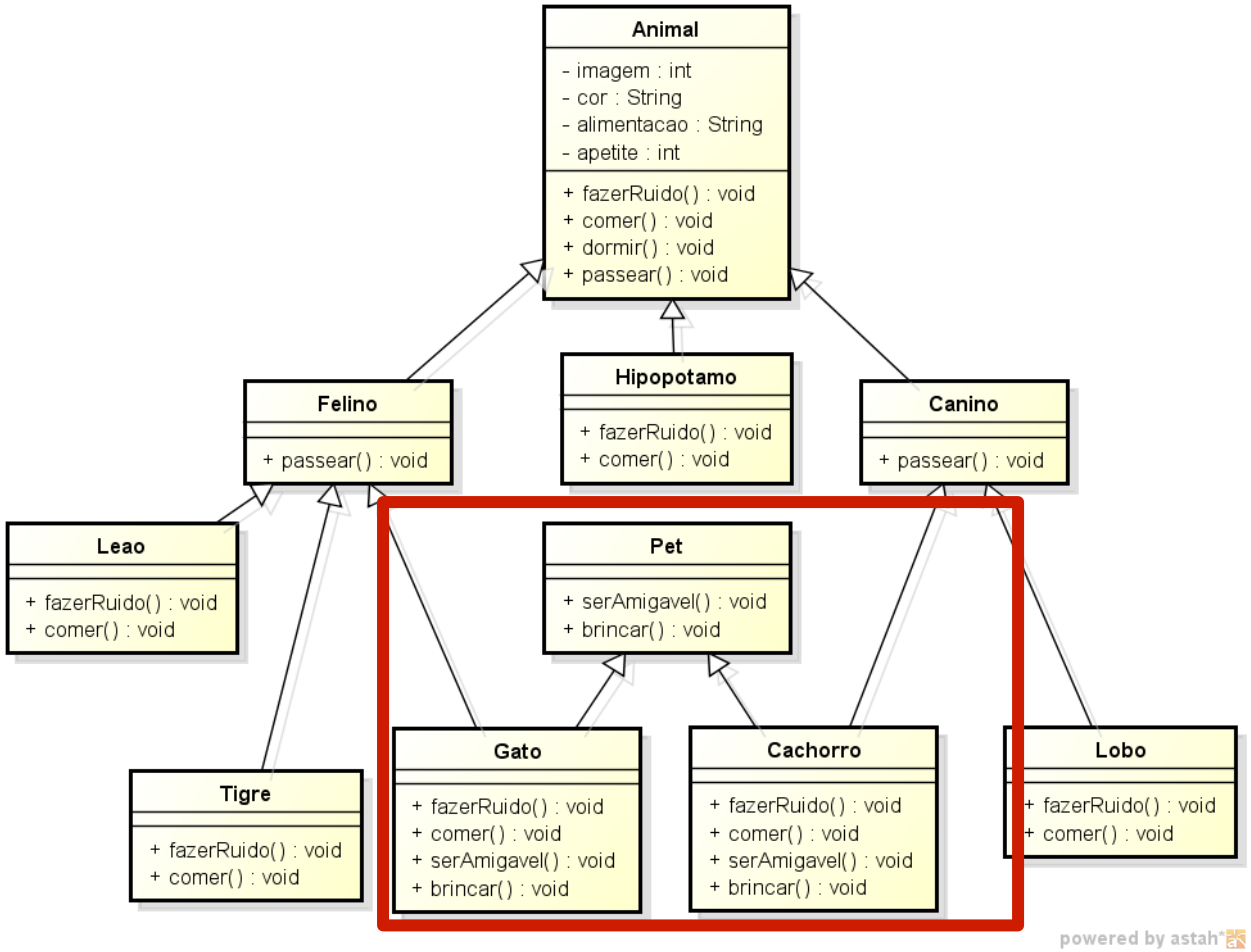
- Considere agora que você deseja adicionar comportamentos de animais domésticos em alguns dos seus anmais.
  - Ex: métodos `serAmigavel()` e `brincar()` em Cachorro, Gato...

- Como fazer isto?
  - **Opção 3:** Inserir os métodos de Pet apenas nas classes que pertencem (ex: Cachorro, Gato, etc)
    - **Vantagem**: Não teremos mais hipópotamos brincalhões =P
    - **Desvantagem**:
      - **Falta de um protocolo para os métodos:** não dá pra garantir que os programadores usarão a mesma assinatura sempre;
      - **Não dá pra implementar polimorfismo com esses métodos**

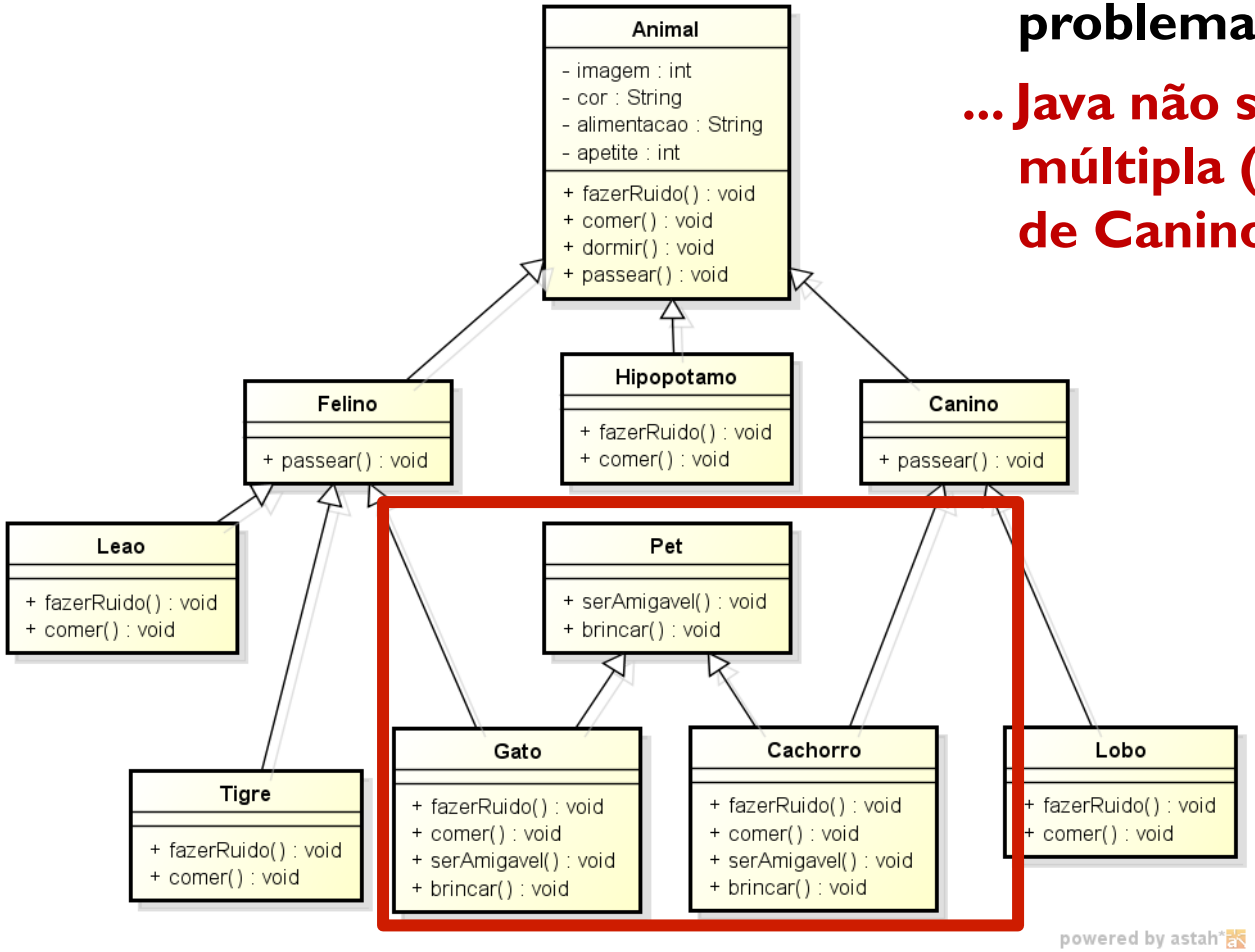
```
Animal a = new Cachorro();
a.brincar(); // errado, pois 'a' é do tipo Animal
```

- Resumindo, precisamos de...
  1. Adicionar o comportamento apenas em algumas classes (Cachorro, Gato, etc)
  2. Garantir que todas as classes tenham os mesmos métodos definidos
  3. Uma forma de usar polimorfismo para todos os métodos de animais domésticos.

Uma outra superclasse abstrata resolveria o problema?



Uma outra superclasse abstrata resolveria o problema?



Resolveria, mas tem um outro problema...

... Java não suporta herança múltipla (Cachorro herda de Canino e de Pet)

Interfaces

- A interface vem nos socorrer!

- Uma **interface** é como uma **classe 100% abstrata**
  - Todos os métodos são abstratos;
  - Ou seja, a subclasse deve implementar todos os métodos;

- Resolve o problema da herança múltipla
  - Embora uma classe só possa **herdar de uma única classe...**
  - ela pode **implementar várias interfaces**

- Fornece os mesmos benefícios polimórficos!

Criação de um Interface

- Sintaxe: Uso da palavra reservada **`interface`**

```
<qualificacao> interface <nomeInterface>{
    // assinatura dos métodos
}
```

- Exemplo:

```
public interface Pet{
    public abstract void serAmigavel();
    public abstract void brincar();
}
```

Implementação de uma Interface

- Uso da palavra reservada `implements`

```
public class <nomeClasse> implements <nomeInterface>{
    // implementação dos métodos de <nomeInterface>
}
```

- Exemplo: Cachorro "é um" Canino e "é um" Pet

```
public class Cachorro extends Canino implements Pet{
    public void fazerRuido(){...}

    public void comer(){...}

    public void serAmigavel(){...}

    public void brincar(){...}
}
```

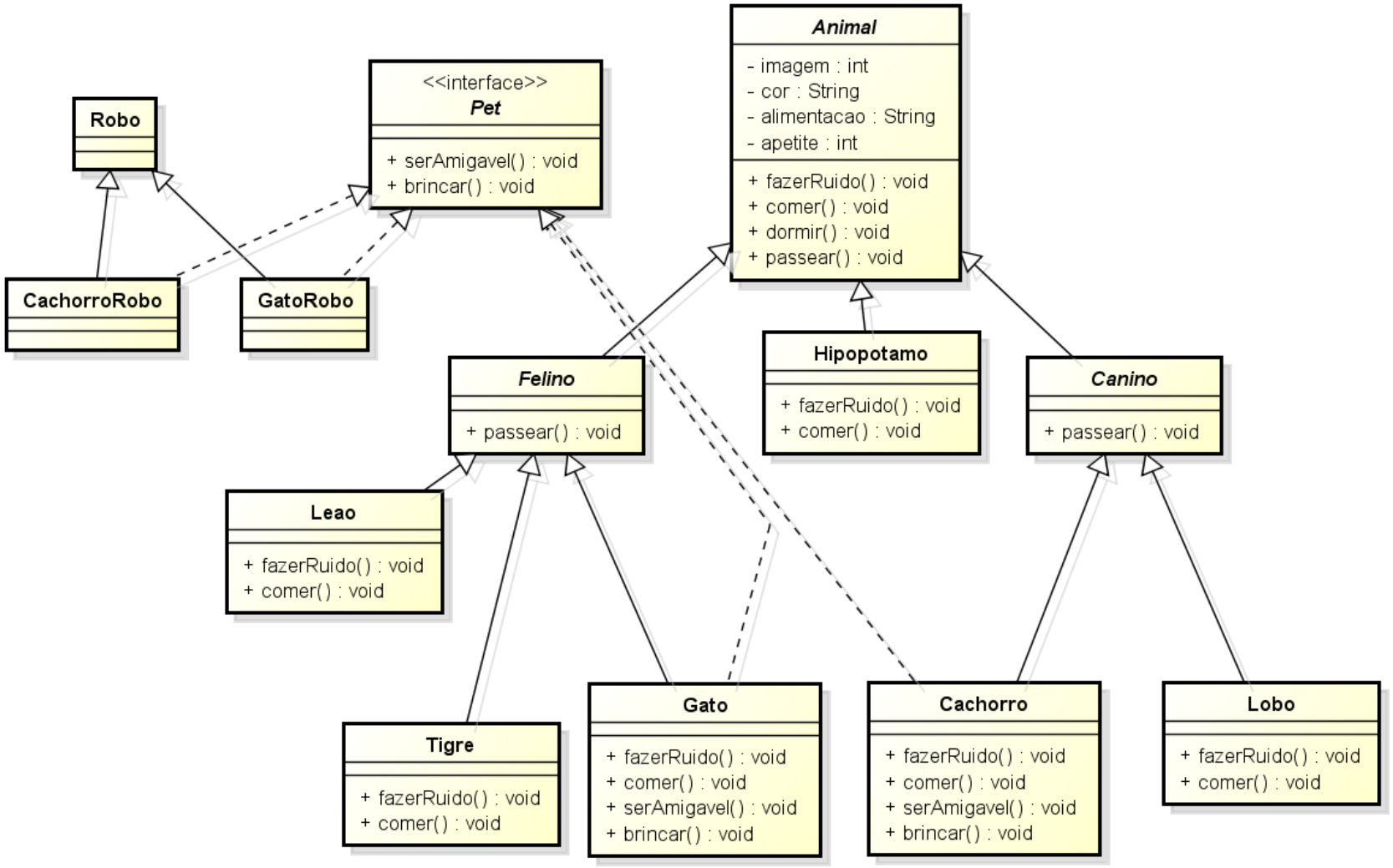
Interfaces

- Mas por que usar interfaces?
  - Polimorfismo, polimorfismo e polimorfismo;
  - Se usadas como parâmetros de métodos, pode-se passar qualquer coisa que implemente essa interface

```
public void brincarComPet(Pet p) {
        p.brincar();
}
```

  - Outras classes de outras hierarquias de herança também podem implementar a interface!
    - Ex: CachorroRobo e GatoRobo que implementam Pet

Exemplo



Universidade Federal da Paraíba

Centro de Informática

Departamento de Informática

Linguagem de Programação

Interfaces e Classes Abstratas I

- Tiago Maritan
- tiago@ci.ufpb.br