Orientação por Objetos

Polimorfismo

Aulas anteriores

Encapsulamento

Implementado através do controle do escopo de visibilidade das propriedades de uma classe.

Prover método público para manipular atributo privado.

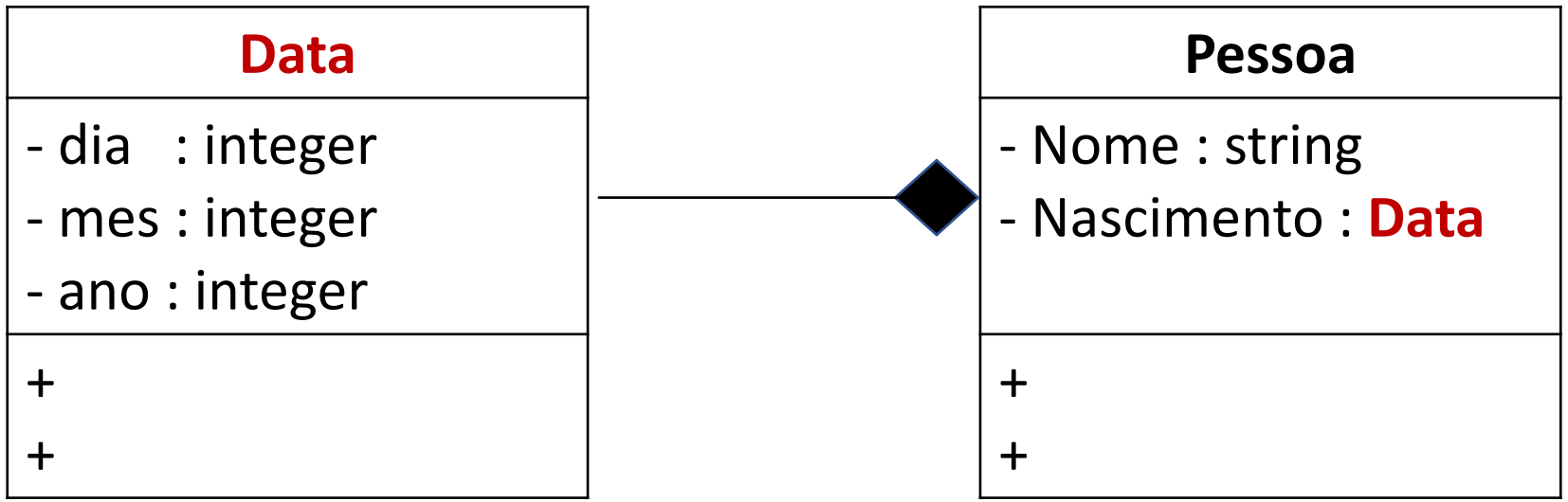
Isto traz uma alternativa para implementar regras de negócio.

Data
- dia : integer - mes : integer - ano : integer
+ +

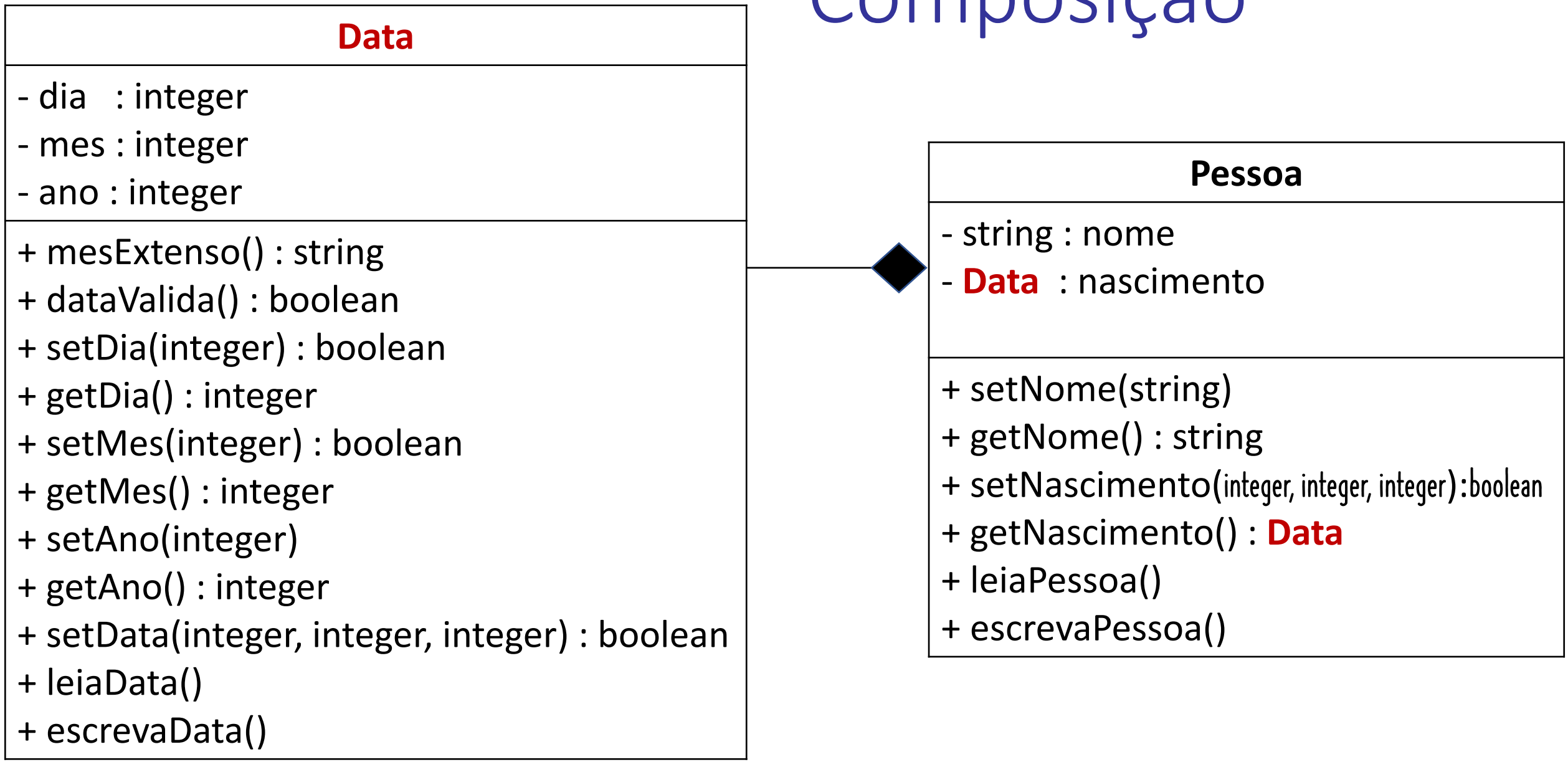
Composição

Uma instância de Pessoa contém uma instância de Data.

Relacionamento do tipo: ***has a***



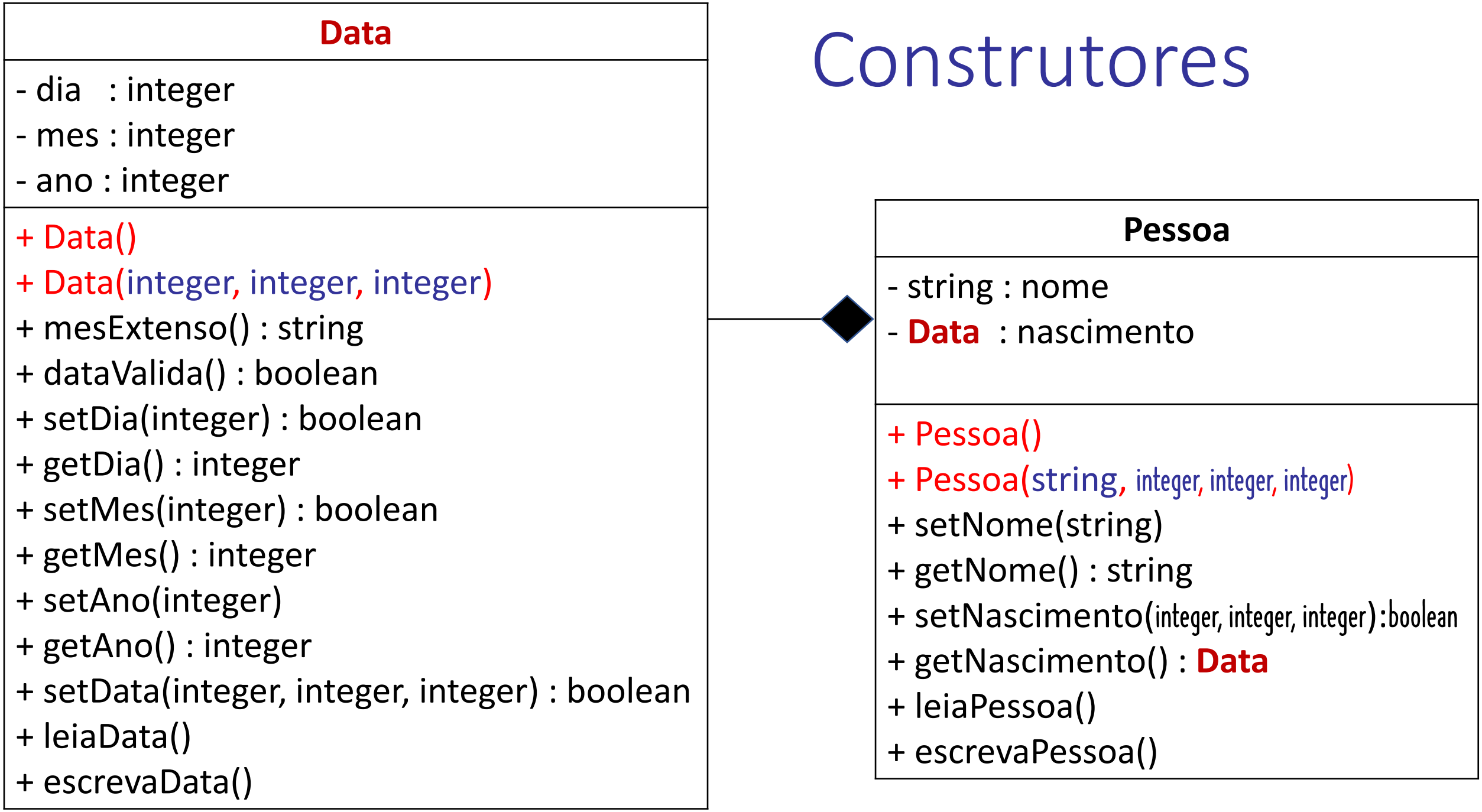
Composição


Data
- dia : integer
- mes : integer
- ano : integer
+ mesExtenso() : string
+ dataValida() : boolean
+ setDia(integer) : boolean
+ getDia() : integer
+ setMes(integer) : boolean
+ getMes() : integer
+ setAno(integer)
+ getAno() : integer
+ setData(integer, integer, integer) : boolean
+ leiaData()
+ escrevaData()
Pessoa
- string : nome
- Data : nascimento
+ setNome(string)
+ getNome() : string
+ setNascimento(integer, integer, integer):boolean
+ getNascimento() : Data
+ leiaPessoa()
+ escrevaPessoa()

```
class Pessoa
{
   private :
      string nome;
      Data   nascimento;
   public :
      void setNome(string nome);
      string getNome();
      bool setNascimento(int dia,int mes,int ano);
      Data getNascimento();
      void lePessoa();
      void escrevePessoa();
};
 void Pessoa::setNome(string nome)
{
     this->nome= nome;
}

string Pessoa::getNome()
{
   return this->nome;
}
```

```
bool Pessoa::setNascimento(int dia, int mes, int ano){
   this-> nascimento.setData(dia,mes,ano);
}

Data Pessoa::getNascimento(){
   return this->nascimento;
}

void Pessoa::leiaPessoa(){
     string nome;
     cout << "\nNome: ";
     cin >> nome;
    setNome(nome);
     cout << "\nData de nascimento: ";
     this->nascimento.leiaData();
}

void Pessoa::escrevaPessoa(){
     cout << "\nNome: " << getNome();
     cout << "\nData de Nascimento: ";
     nascimento.escrevaData();
}
```

Construtores



Construtor

Método executado sempre que uma nova instância for criada

Não se aplica o conceito de tipo do método por não haver chamada explícita

Implementa uma regra para criação dos objetos daquela classe

```
class Data
{
   private :
      int dia;
      int mes;
      int ano;

   public :
     Data() { }
      Data(int dia, int mes, int ano) {
          setData(dia,mes,ano);
      }
};
```

```
...
Data* data1 = new Data();
Data* data2 = new Data(21, 11, 2022);
...
```

Destrutor

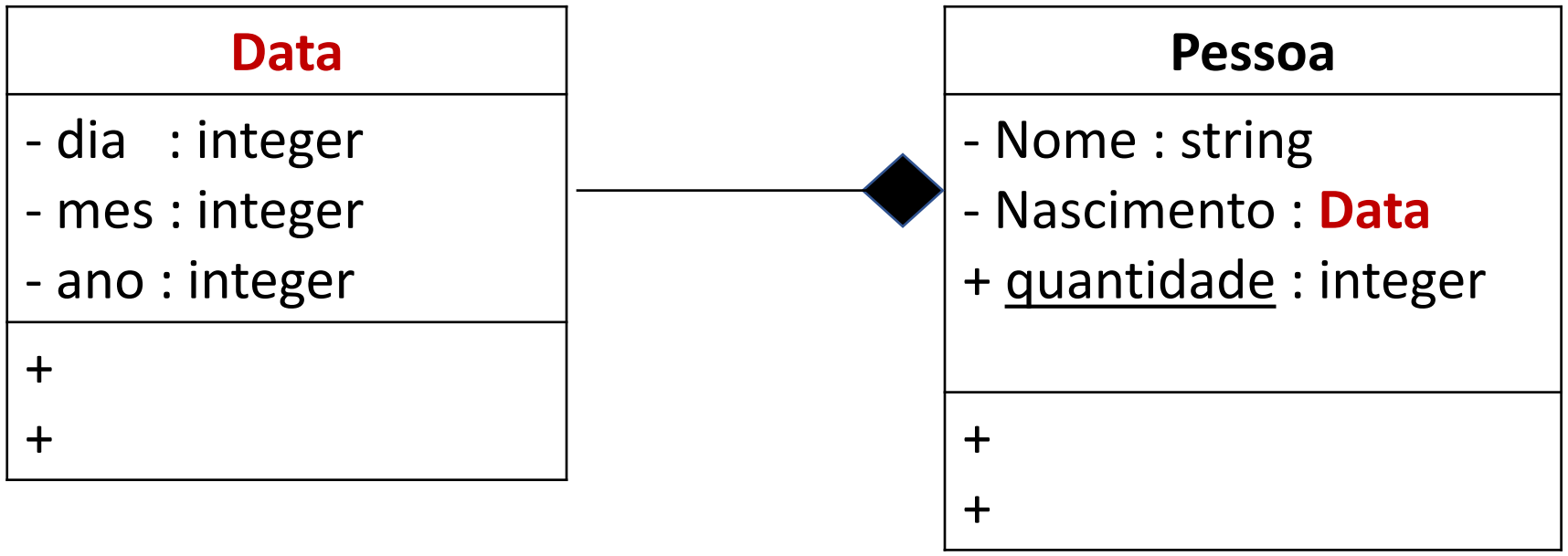
Método executado sempre que uma instância for excluída.

```
...
Data* data1 = new Data();
Data* data2 = new Data(21, 11, 2022);

delete data1;
delete data2;
...
```

```
class Data
{
   private :
      int dia;
      int mes;
      int ano;

   public :
     Data() { }
      Data(int dia, int mes, int ano) {
          setData(dia,mes,ano);
      }
      ~Data() {
          ...
      }
};
```

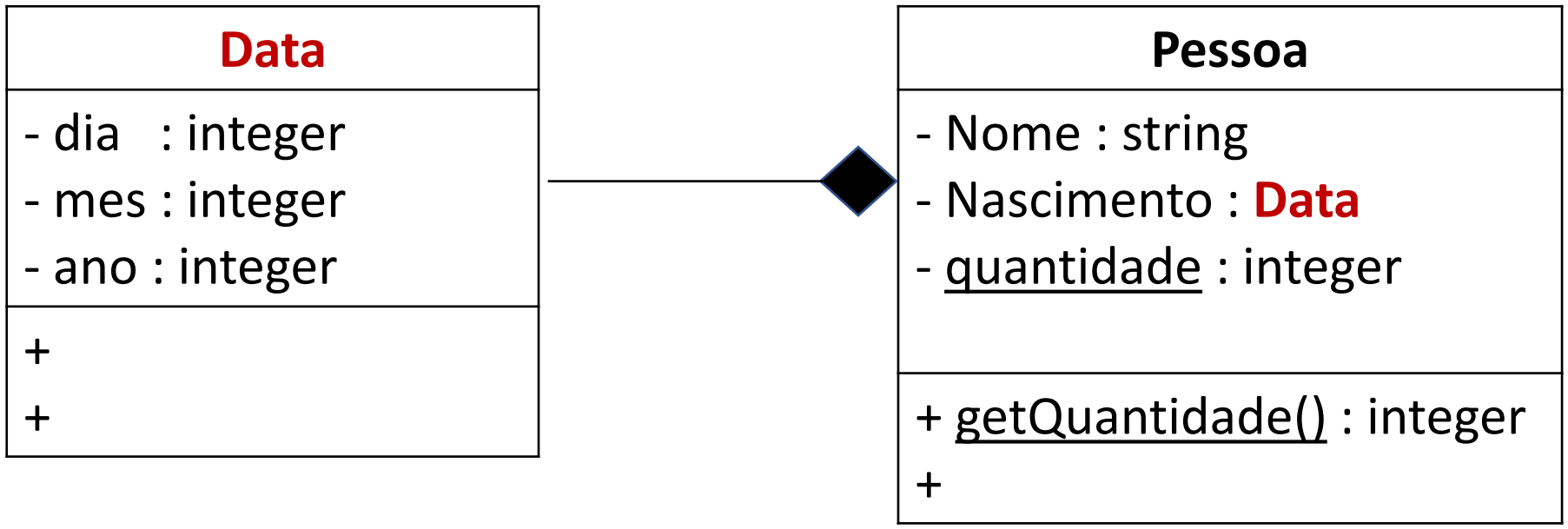
Propriedade Estática

Atributo ou método que pertence à classe, e não às suas instâncias.



Propriedade Estática

Atributo ou método que pertence à classe, e não às suas instâncias.



Atributo estático em C++

Declarado na classe, inicializado em sua manipulação.

Não requer a criação de uma primeira instância para existir

```
class Qualquer
{
   private :
      int a;
      int b;

   public :
      static int x;
};

int Qualquer::x=0;
```

```
int main()
{
   Qualquer qualquer;
   ...

}
```

Exemplo: contando instâncias com estático e público

```
class Qualquer
{
   private :
      int a;
      int b;

   public :
      static int quantidade;
      Qualquer() {
            quantidade++;
      }
};

int Qualquer::quantidade=0;
```

```
int main()
{
   Qualquer qualquer;

   cout << endl << Qualquer::quantidade;

}
```

Exemplo: contando instâncias com estático e privado

```
class Qualquer
{
   private :
      int a;
      int b;
      static int quantidade;

   public :
      Qualquer() {
            quantidade++;
      }
};

int Qualquer::quantidade=0;
```

```
int main()
{
   Qualquer qualquer;

   cout << endl << Qualquer::quantidade;

}
```

Exemplo: contando instâncias com estático e privado

```
class Qualquer
{
   private :
     int a;
     int b;
     static int quantidade;

   public :
     Qualquer() {
          quantidade++;
     }
     static int getQuantidade(){
           return quantidade;
     }
};
```

```
int main()
{
   Qualquer qualquer;

   cout << endl << Qualquer::getQuantidade();

}
```

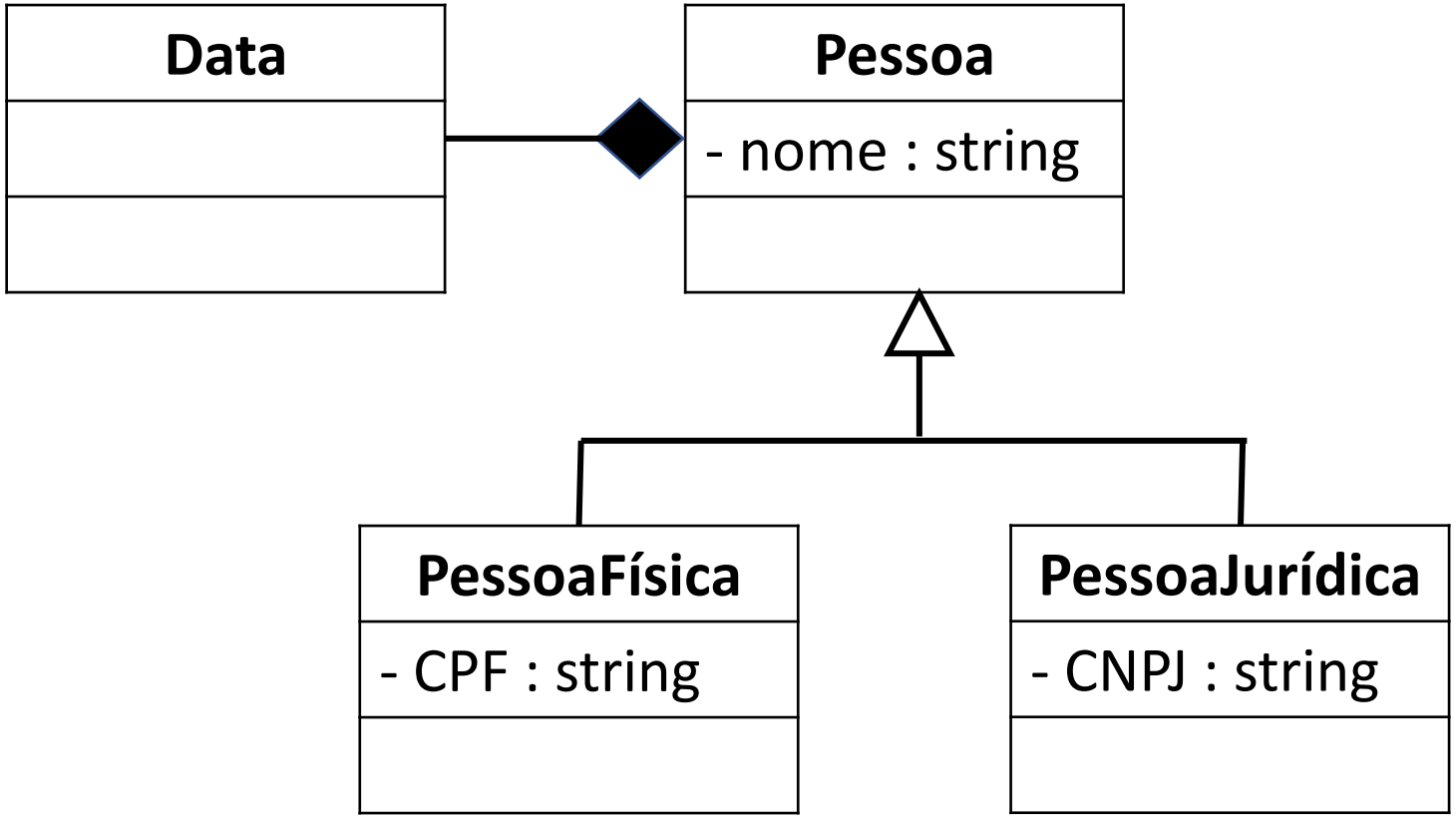
Generalização (herança)

Uma classe (base) pode generalizar as propriedades comuns de outras (derivadas)

Uma classe (base) pode se especializar em outras (derivadas)

Uma classe (derivada) pode herdar as propriedades descritas em outra (base)

Relacionamento do tipo: ***is a***

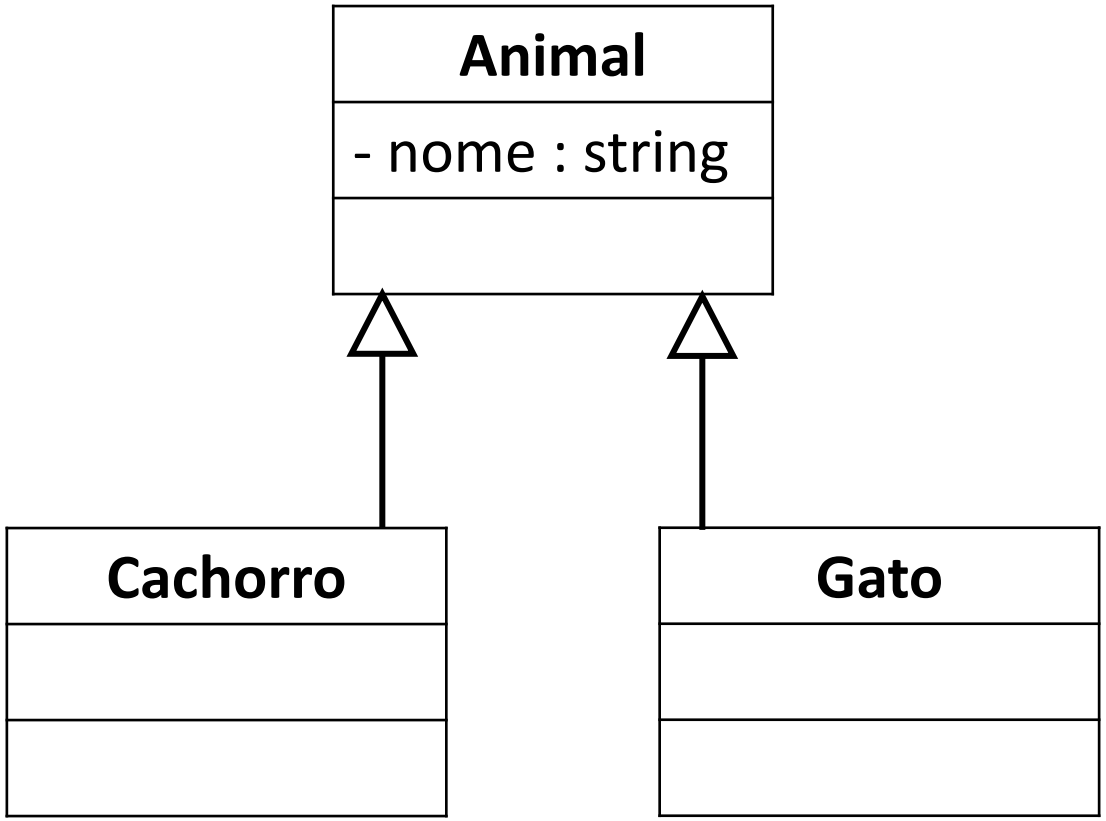


Generalização

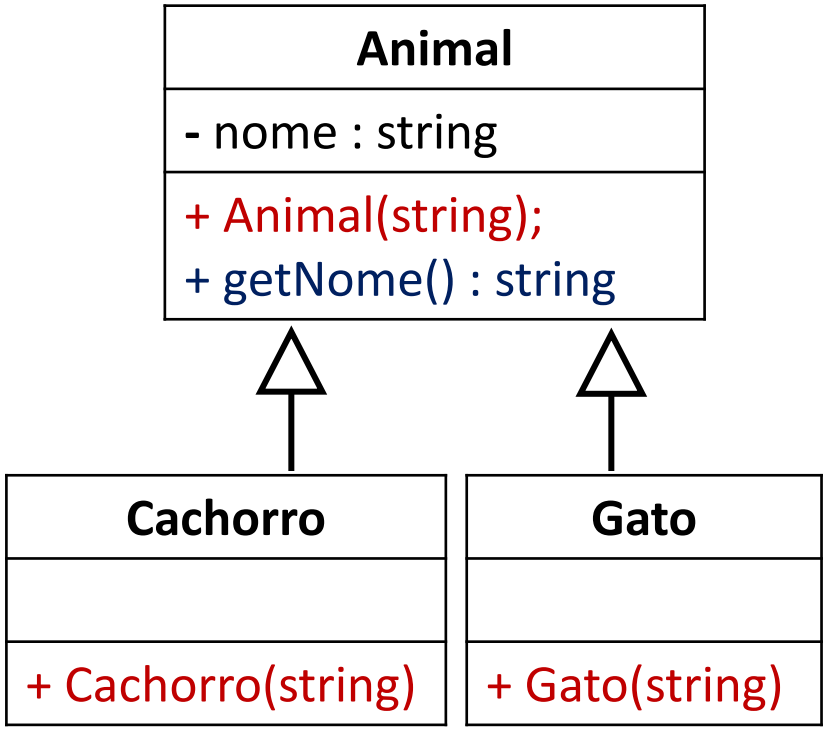
Relacionamento do tipo: ***is a*** *(é um)*

Classe-base (Superclasse) generaliza propriedades comuns (ou pura abstração)

Classes-derivadas (Subclasses) representam suas especializações



Generalização

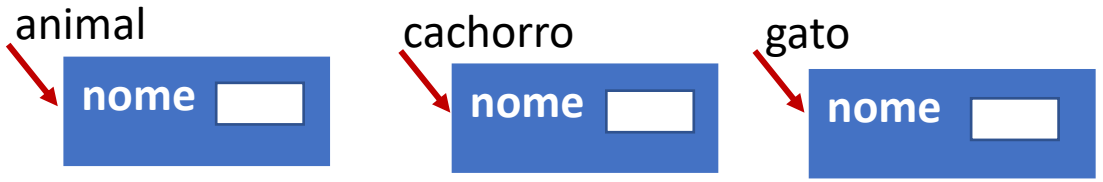



```
class Cachorro : public Animal {
  public :
    Cachorro(string nome) : Animal(nome)
    {  }
};
```

```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
          this->nome= nome;
     }
     string getNome() {
          return nome;
     }
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome)
      {  }
   };
```

```
...
Animal* animal;
animal = new Animal("bichinho");
...
Cachorro * cachorro;
cachorro = new Cachorro("sol");
...
Gato * gato;
gato = new Gato("lua");
...
cout << animal->getNome();
cout << cachorro->getNome();
cout << gato->getNome();
```



Polimorfismo

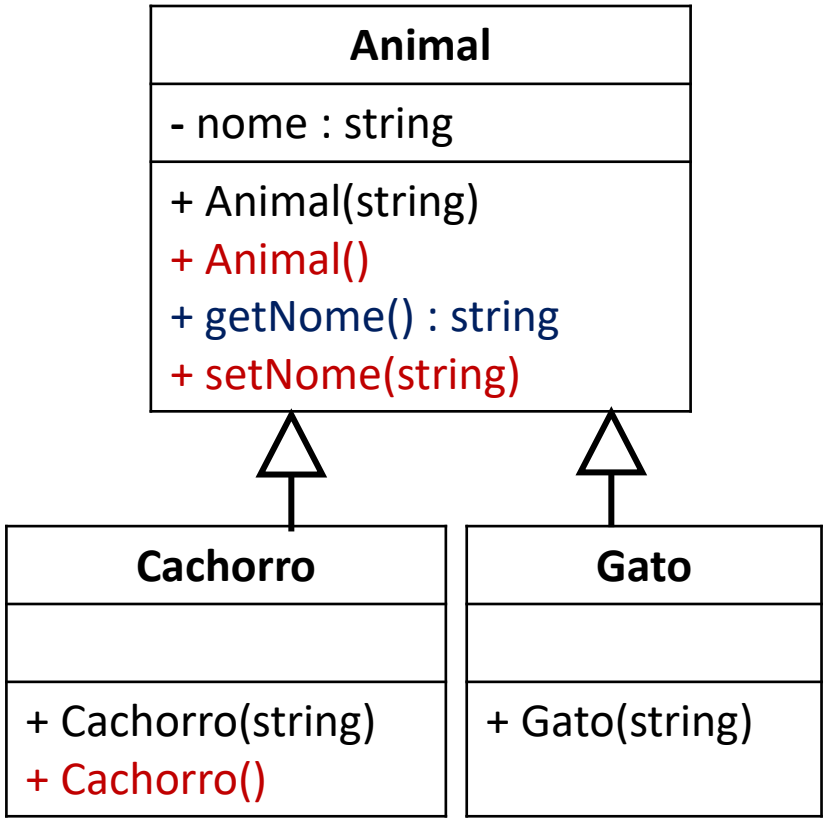
Polimorfismo fraco: *em tempo de compilação*
Funções sobrecarregadas: *mesmo nome, leve diferença na assinatura*

```
double area(double base, double altura){
    return base * altura;
}

double area(double lado)
    return lado * lado;
}
```

Polimorfismo forte

Utiliza características advindas da generalização.

Explora a relação entre *tipos* e *subtipos.*

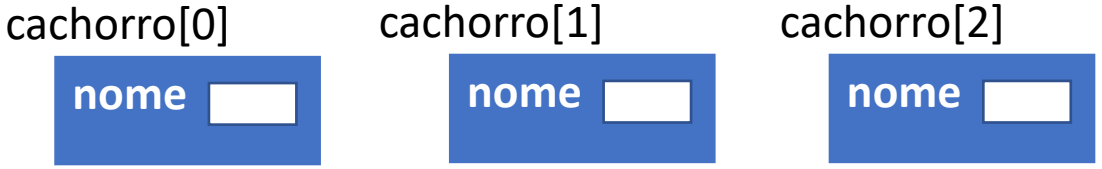


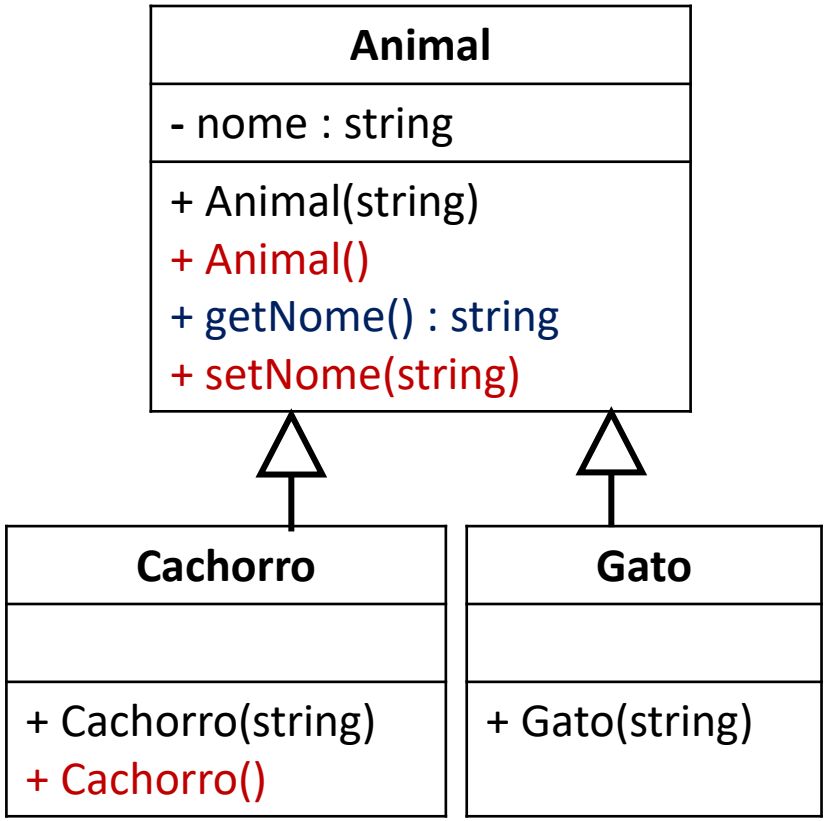

```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
           setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {   }
     Cachorro() {  }
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
};
```

```
...
Cachorro cachorro[3];
...
```



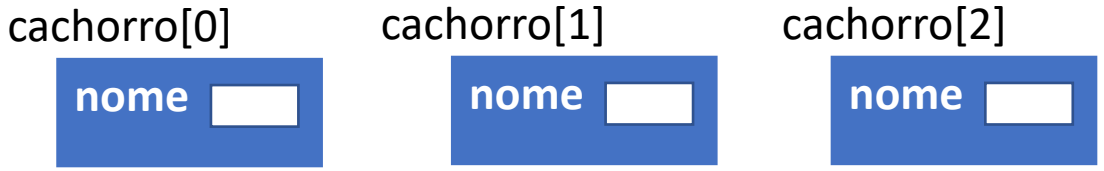


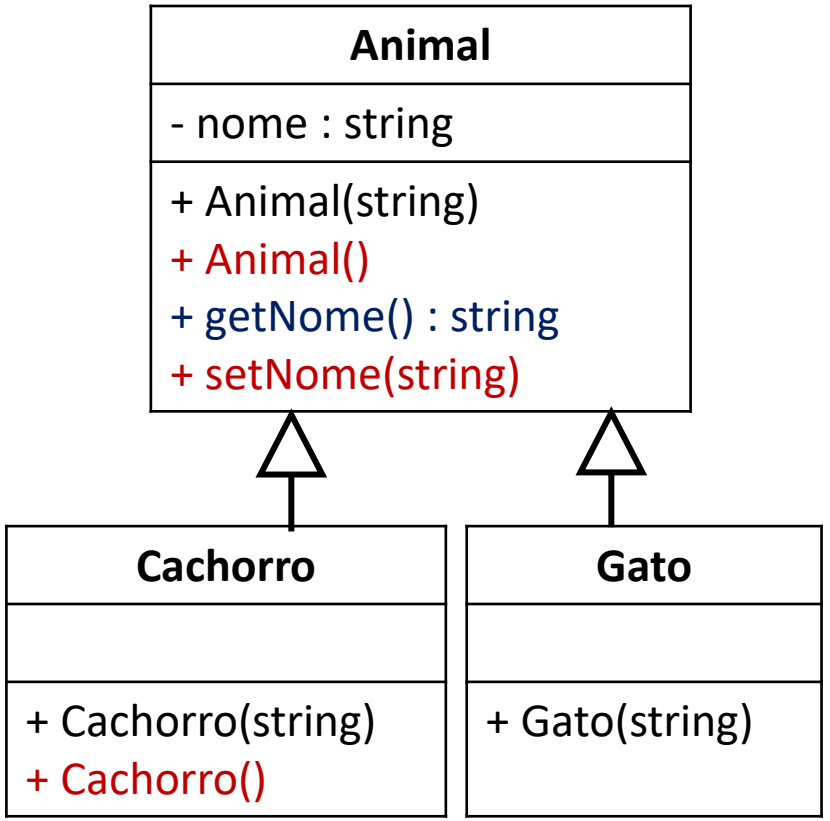

```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
           setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {   }
     Cachorro() {  }
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
};
```

```
...
Cachorro cachorro[3];
leCachorros(cachorro);
...
```





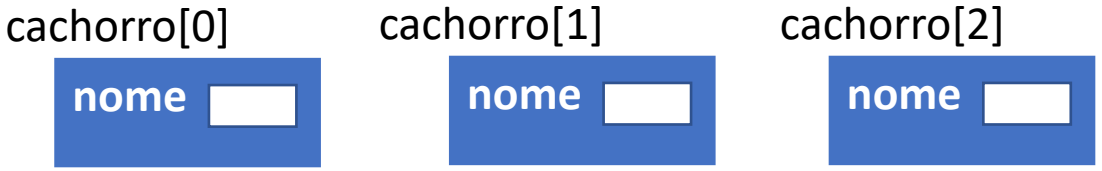

```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
           setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

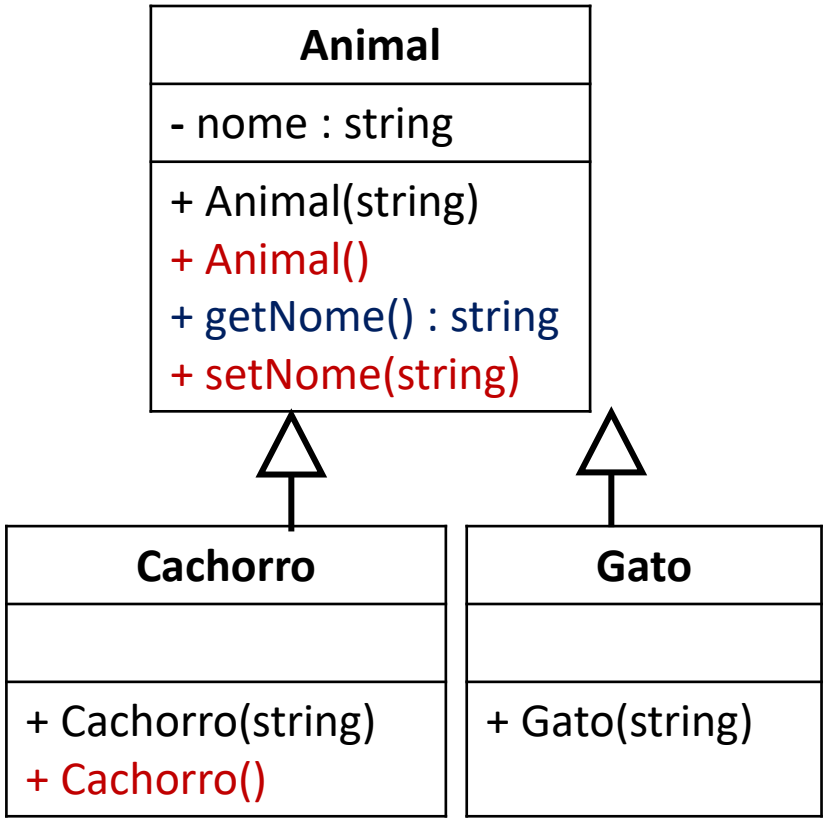
```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {   }
     Cachorro() {  }
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
};
```

```
...
Cachorro cachorro[3];
leCachorros(cachorro);
...
```

```
void leCachorros(Cachorro cachorro[]){
   string nome;
   for(int i=0; i<3; i++){
       cout << "\nCachorro " << i+1 << ": ";
       cin >> nome;
       cachorro[i].setNome(nome);
     }
}
```



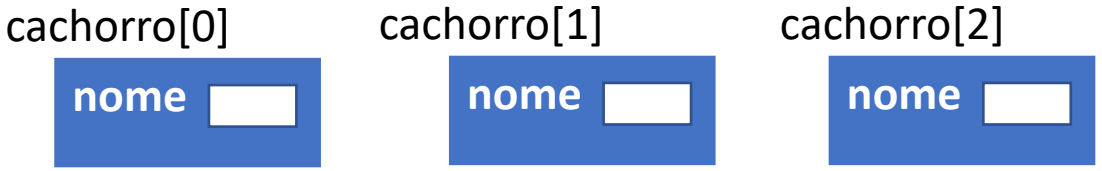


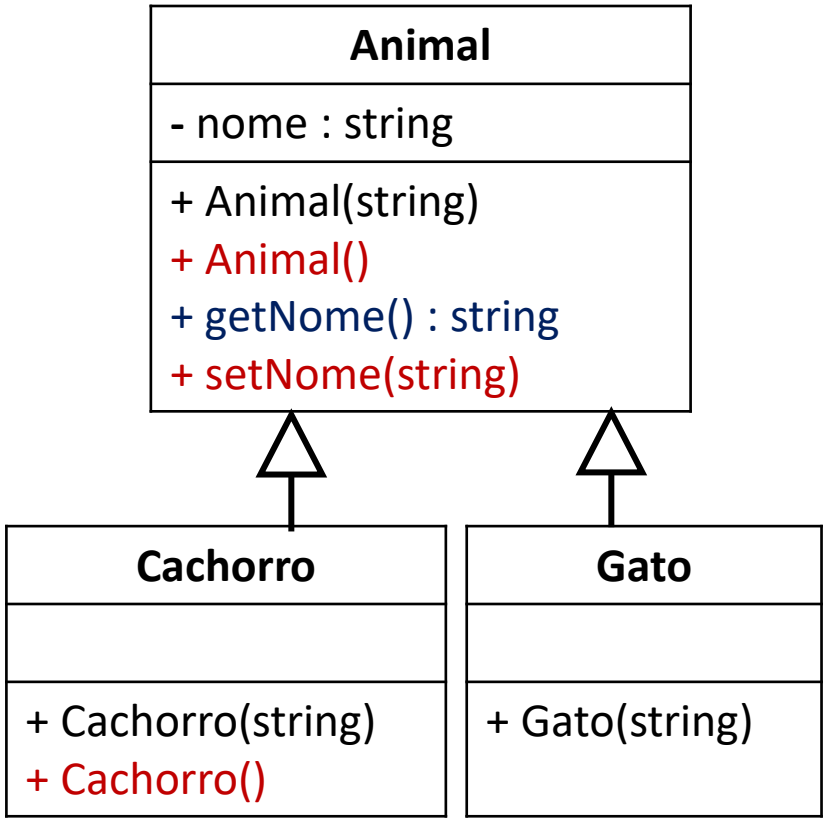

```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
          setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {   }
     Cachorro() {  }
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
};
```

```
...
Cachorro cachorro[3];
leCachorros(cachorro);
listaCachorros(cachorro);
...
```





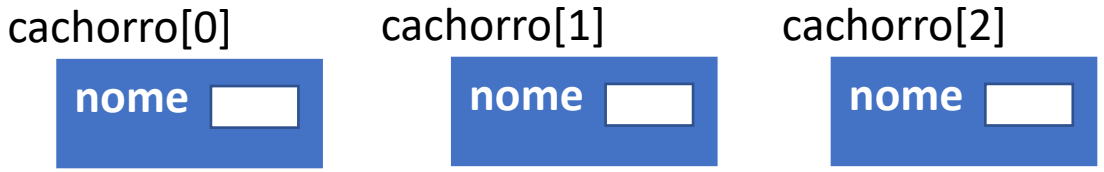

```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
           setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

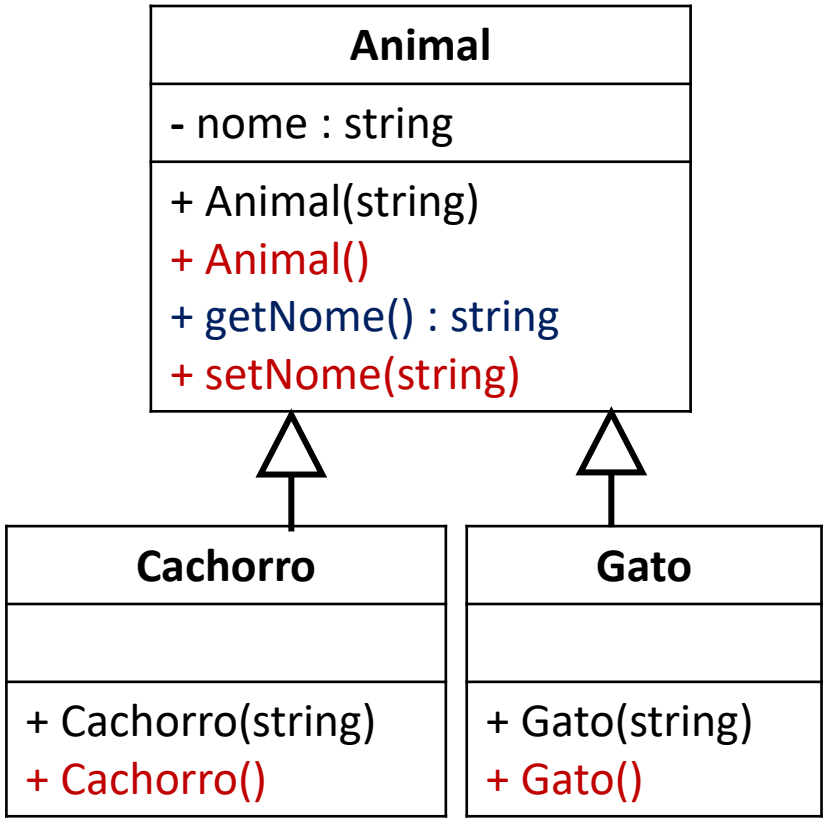
```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {   }
     Cachorro() {  }
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
};
```

```
...
Cachorro cachorro[3];
leCachorros(cachorro);
listaCachorros(cachorro);
...
```

```
void listaCachorros(Cachorro cachorro[]){
  for(int i=0; i<3; i++){
      cout << endl
           << cachorro[i].getNome();
   }
}
```



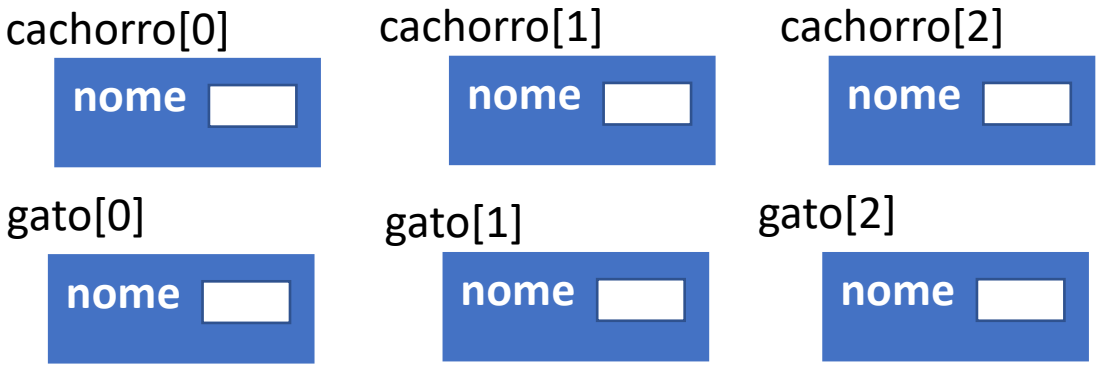


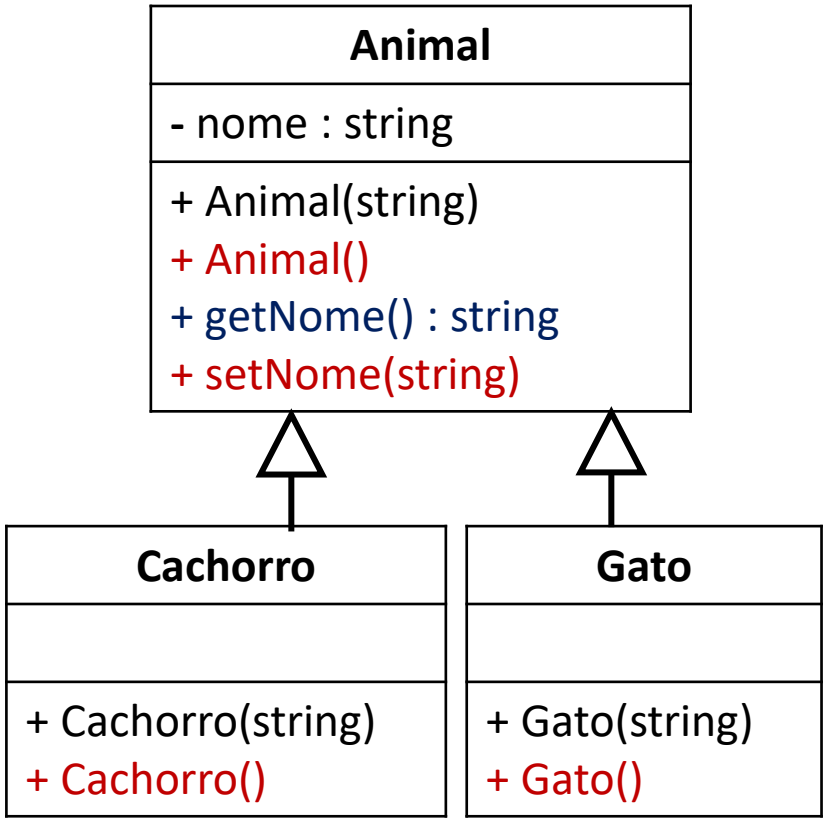

```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
           setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {   }
     Cachorro() {  }
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
      Gato() { }
};
```

```
...
Cachorro cachorro[3];
leCachorros(cachorro);
listaCachorros(cachorro);
...
Gato gato[3];
```



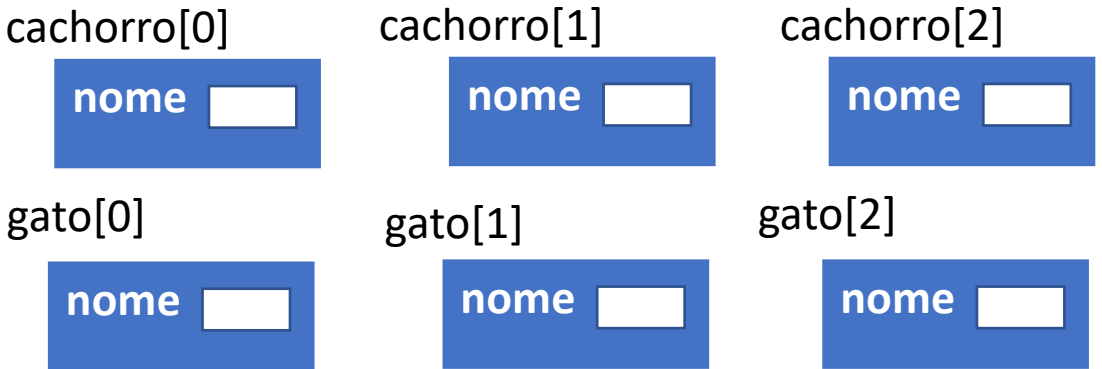


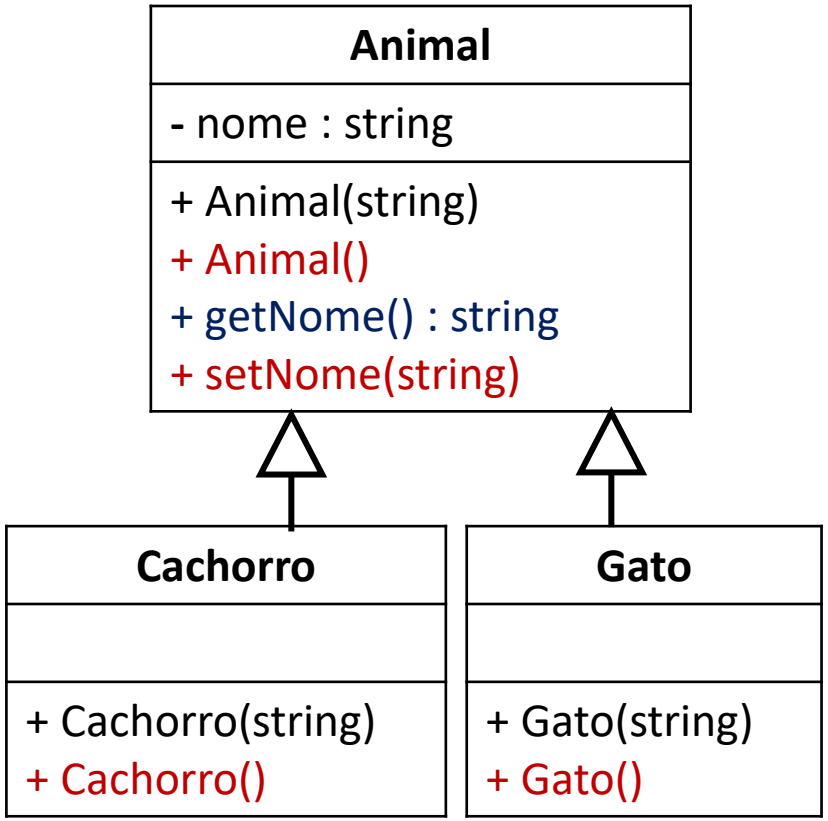

```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
           setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {   }
     Cachorro() {  }
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
      Gato() { }
};
```

```
...
Cachorro cachorro[3];
leCachorros(cachorro);
listaCachorros(cachorro);
...
Gato gato[3];
leGatos(gato);
```





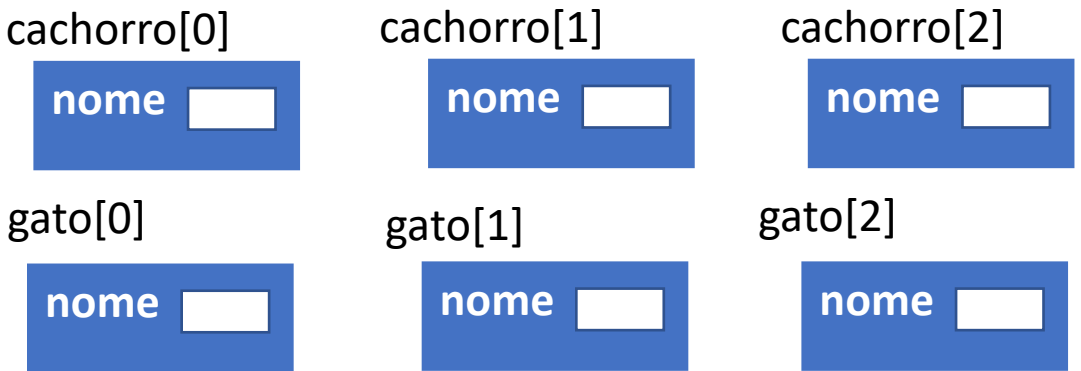

```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
           setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

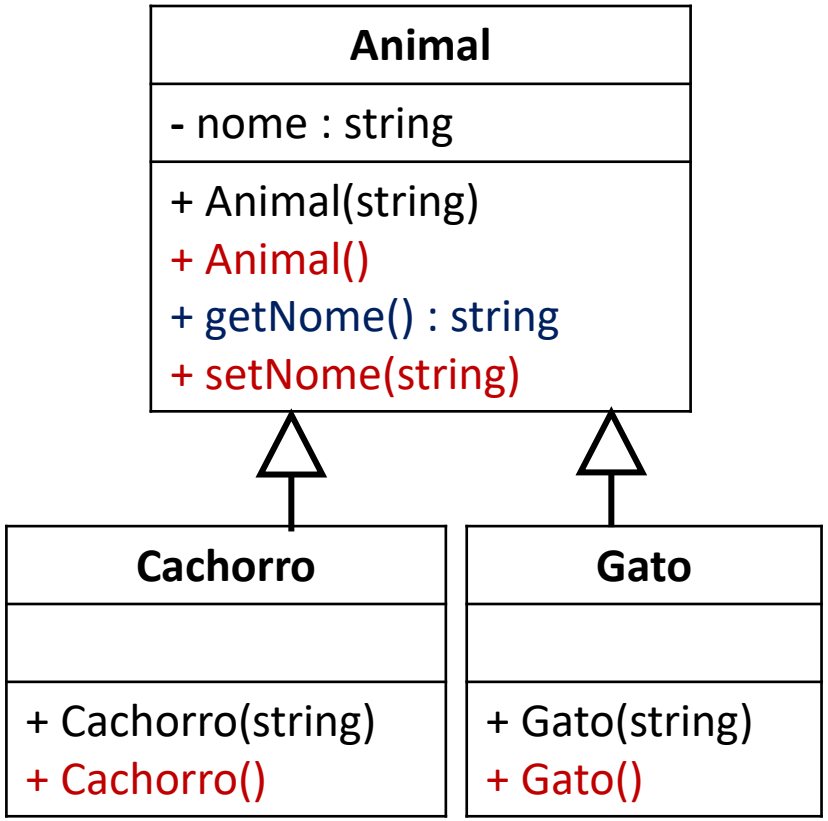
```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {   }
     Cachorro() {  }
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
      Gato() { }
};
```

```
...
Cachorro cachorro[3];
leCachorros(cachorro);
listaCachorros(cachorro);
...
Gato gato[3];
leGatos(gato);
```

```
void leGatos(Gato gato[]){
   string nome;
   for(int i=0; i<3; i++){
       cout << "\nGato " << i+1 << ": ";
       cin >> nome;
       gato[i].setNome(nome);
     }
}
```

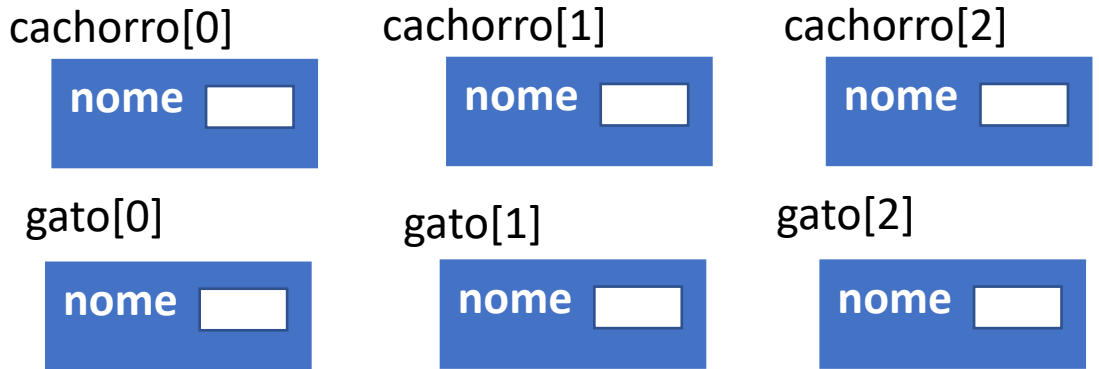


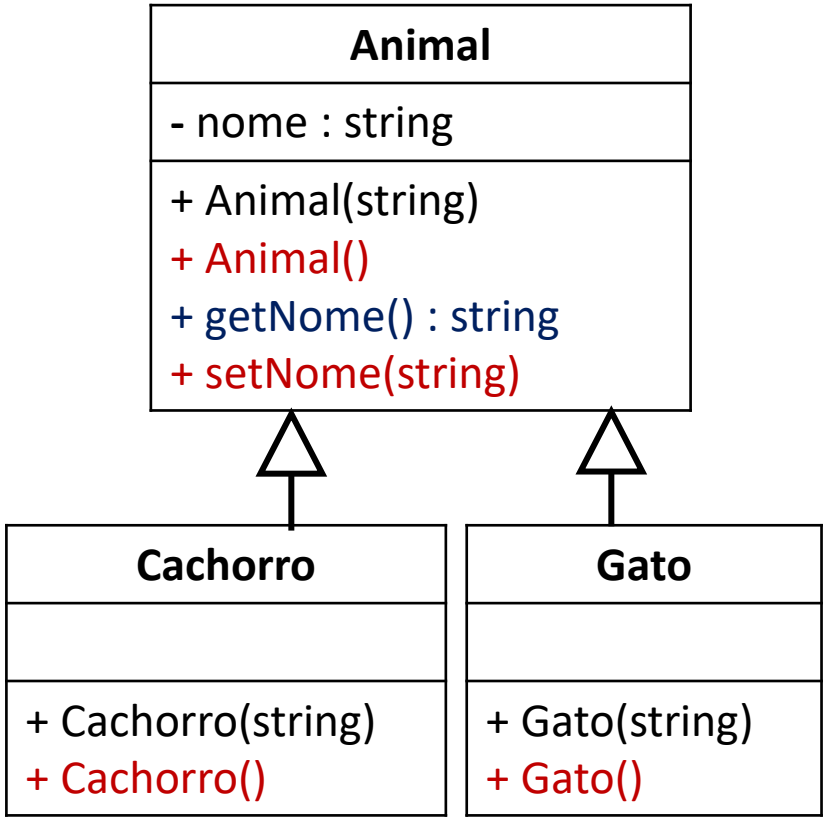



```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
           setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {   }
     Cachorro() {  }
};

class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
      Gato() { }
};
```

```
...
Cachorro cachorro[3];
leCachorros(cachorro);
listaCachorros(cachorro);
...
Gato gato[3];
leGatos(gato);
listaGatos(gato);
```





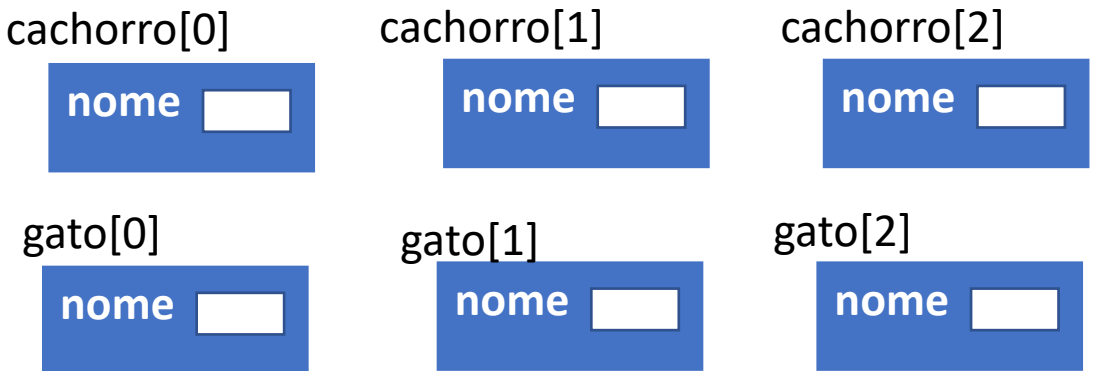

```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
           setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

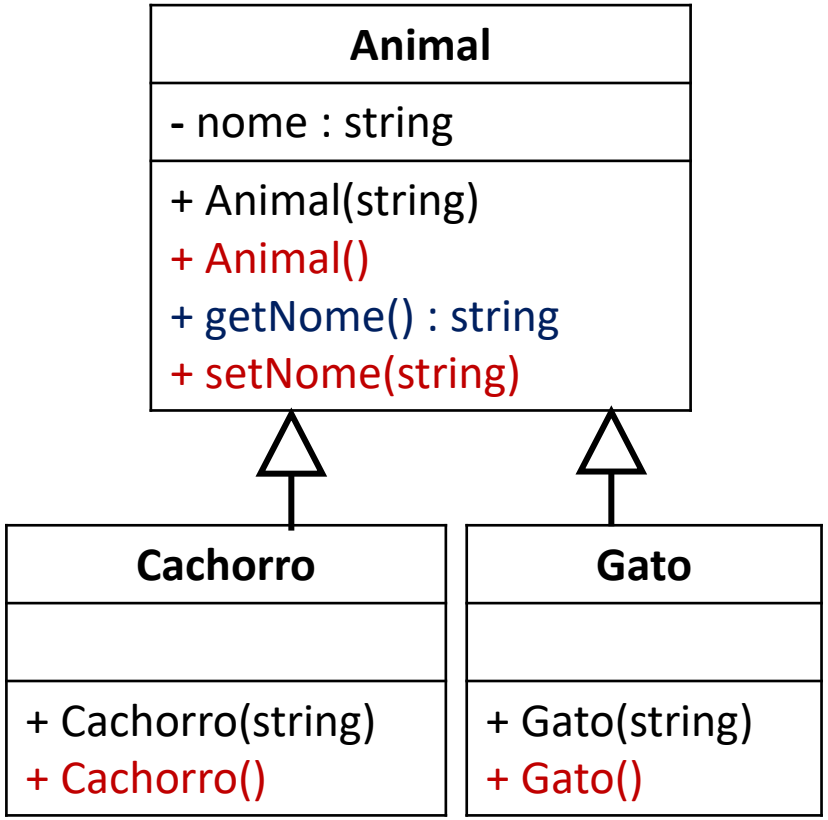
```
...
Cachorro cachorro[3];
leCachorros(cachorro);
listaCachorros(cachorro);
...
Gato gato[3];
leGatos(gato);
listaGatos(gato);
...
```

```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {  }
     Cachorro() {  }
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {  }
      Gato() { }
};
```

```
void listaGatos(Gato gato[]){
  for(int i=0; i<3; i++){
     cout << endl << gato[i].getNome();
   }
}
```



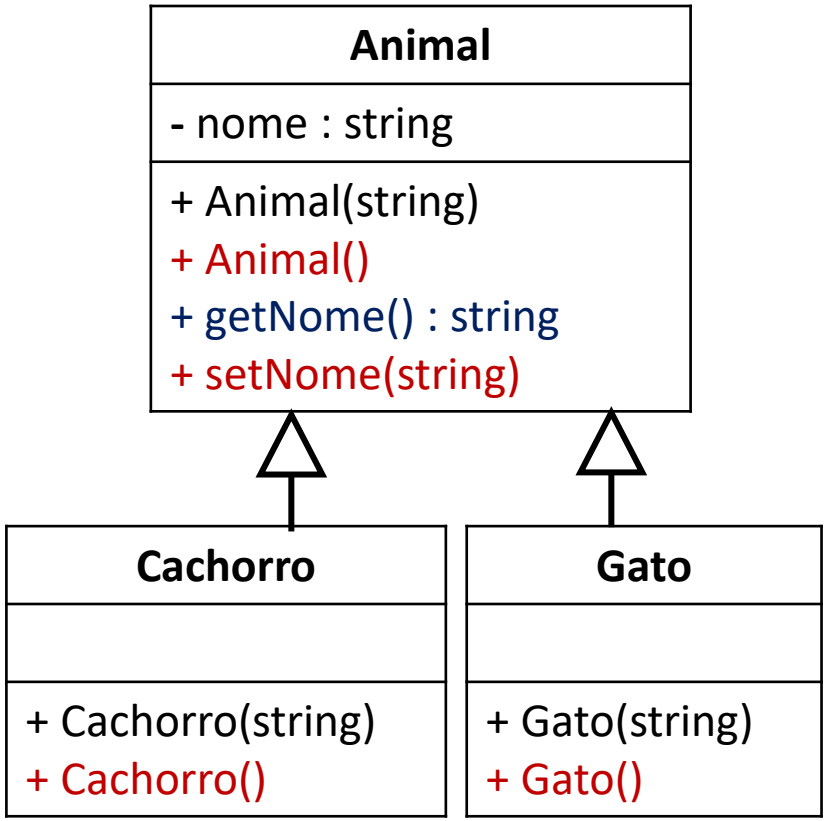



```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
          setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {   }
     Cachorro() {  }
};

class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
      Gato() { }
};
```

```
...
Cachorro cachorro[3];
leCachorros(cachorro);
listaCachorros(cachorro);
...
Gato gato[3];
leGatos(gato);
listaGatos(gato);
...
```

```
void listaCachorros(Cachorro cachorro[]){
  for(int i=0; i<3; i++){
      cout << endl
           << cachorro[i].getNome();
   }
}

void listaGatos(Gato gato[]){
  for(int i=0; i<3; i++){
     cout << endl << gato[i].getNome();
   }
}
```

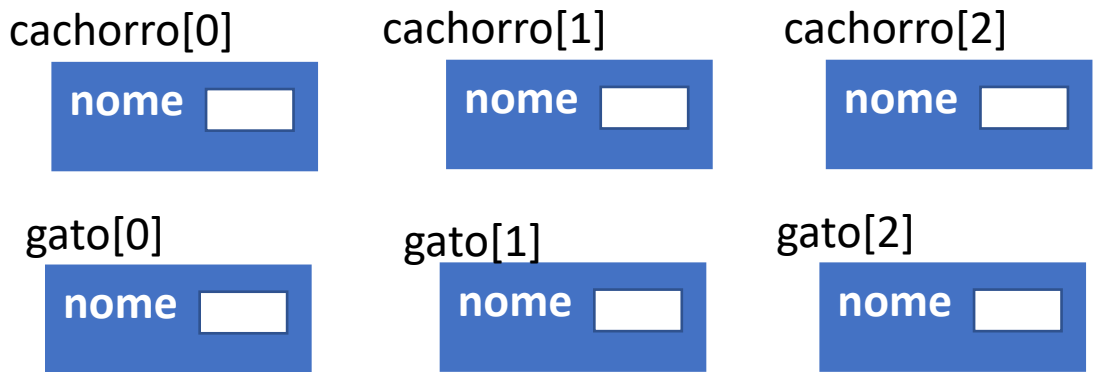


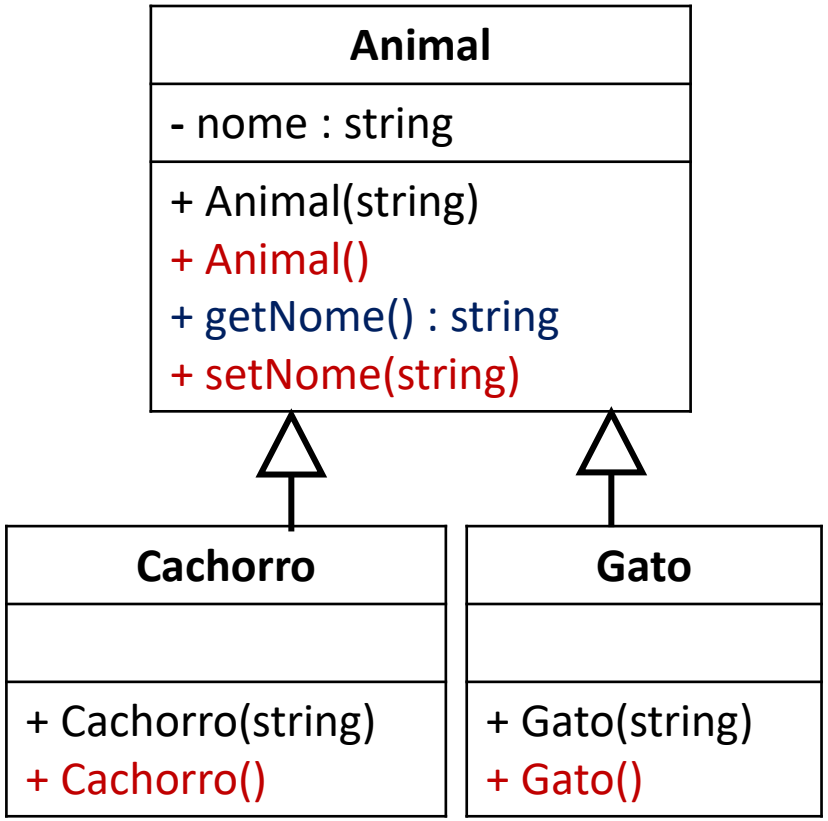

```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
           setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {   }
     Cachorro() {  }
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
      Gato() { }
};
```

```
...
Cachorro cachorro[3];
leCachorros(cachorro);
listaCachorros(cachorro);
...
Gato gato[3];
leGatos(gato);
listaGatos(gato);
...

listaAnimais(cachorro);
listaAnimais(gato);
```



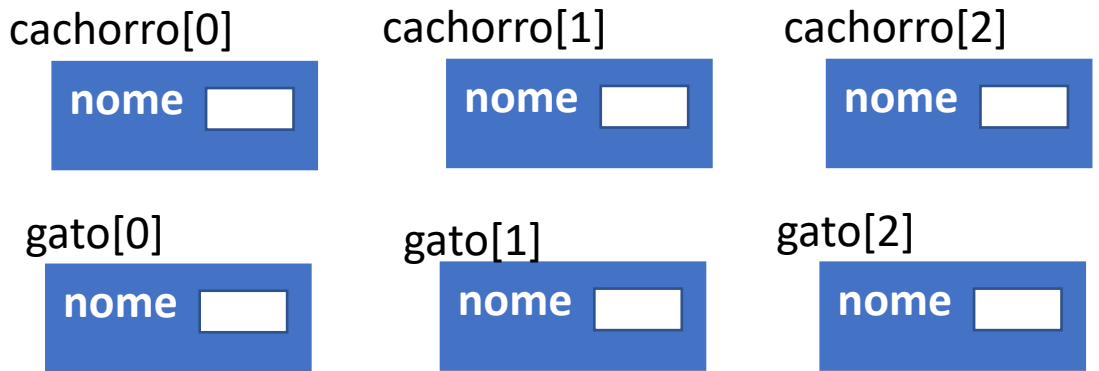


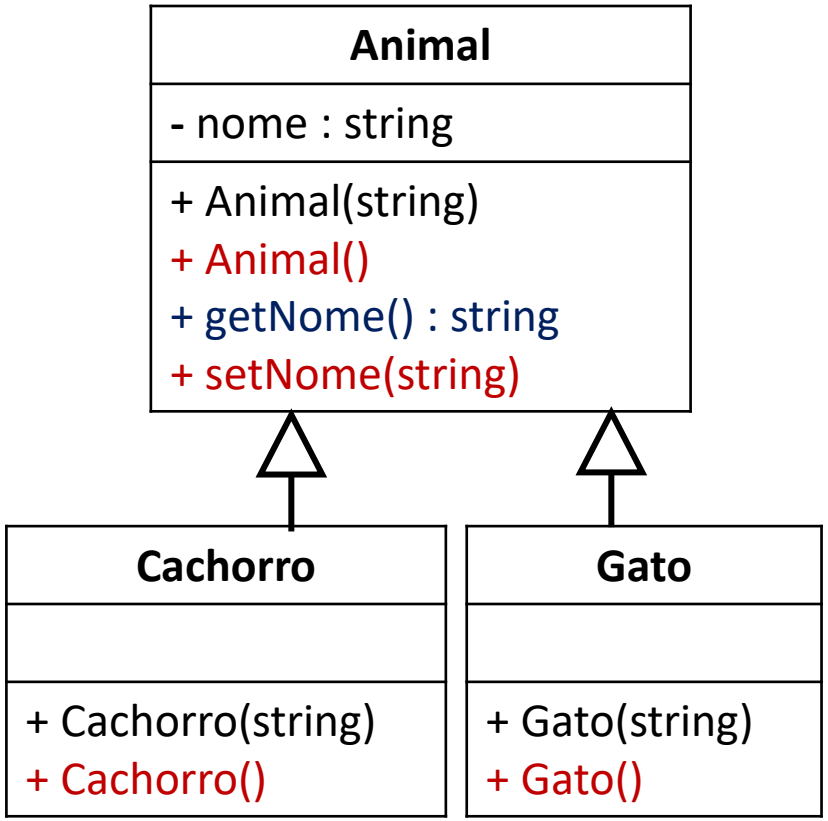

```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
           setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {   }
     Cachorro() {  }
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
      Gato() { }
};
```

```
...
Cachorro cachorro[3];
leCachorros(cachorro);
...
Gato gato[3];
leGatos(gato);
...
listaAnimais(cachorro);
listaAnimais(gato);


void listaAnimais(Animal animal[]){
  for(int i=0; i<3; i++){
     cout << endl << animal[i].getNome();
  }
}
```



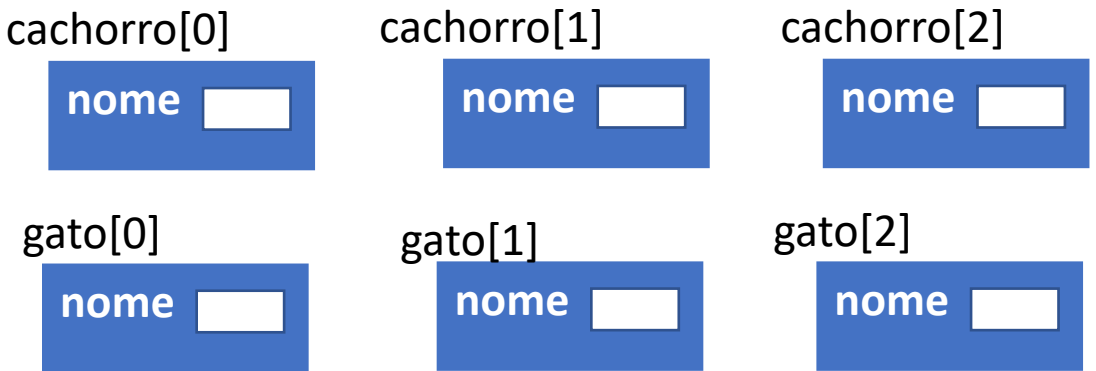



```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
          setNome(nome);
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
};
```

```
class Cachorro : public Animal {
   public :
     Cachorro(string nome) : Animal(nome) {   }
     Cachorro() {  }
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
      Gato() { }
};
```

```
...
Cachorro cachorro[3];
Gato gato[3];
...

listaAnimais(cachorro);
listaAnimais(gato);

// Função polimórfica
void listaAnimais(Animal animal[]){
  for(int i=0; i<3; i++){
     cout << endl << animal[i].getNome();
   }
}
```



Polimorfismo Forte
Determinado em tempo de execução

Uma única maneira de evocar um serviço,

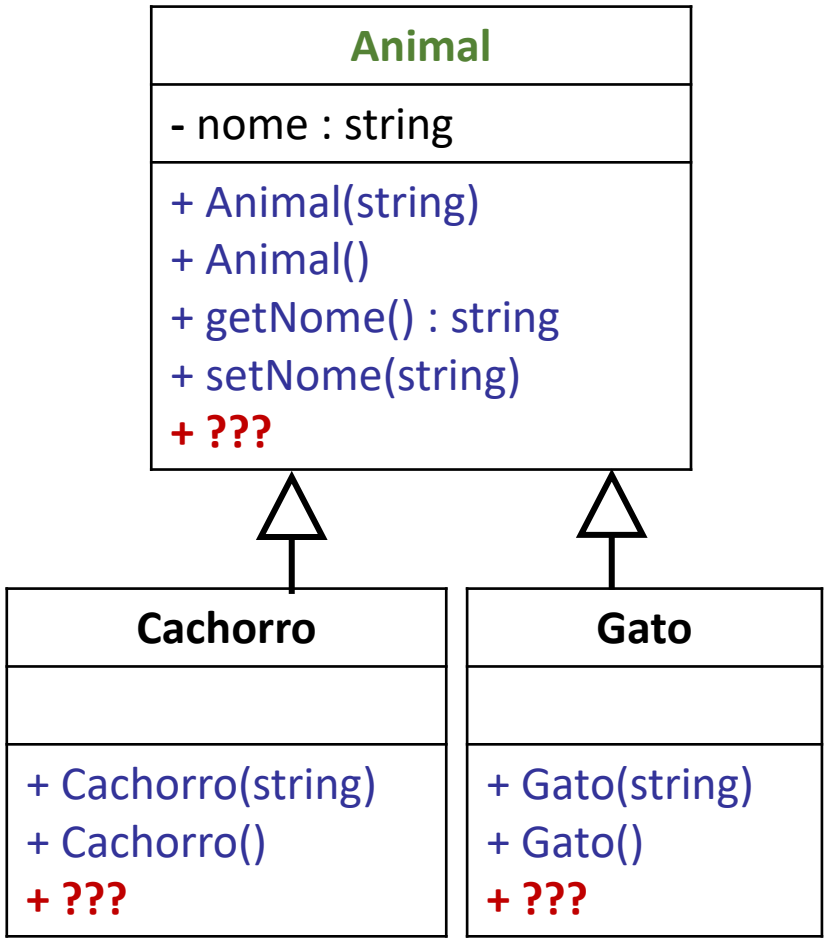
diferentes formas de ele ser prestado.

O método a ser executado é definido pelo tipo concreto do objeto corrente, e não do tipo da variável que o referencia.

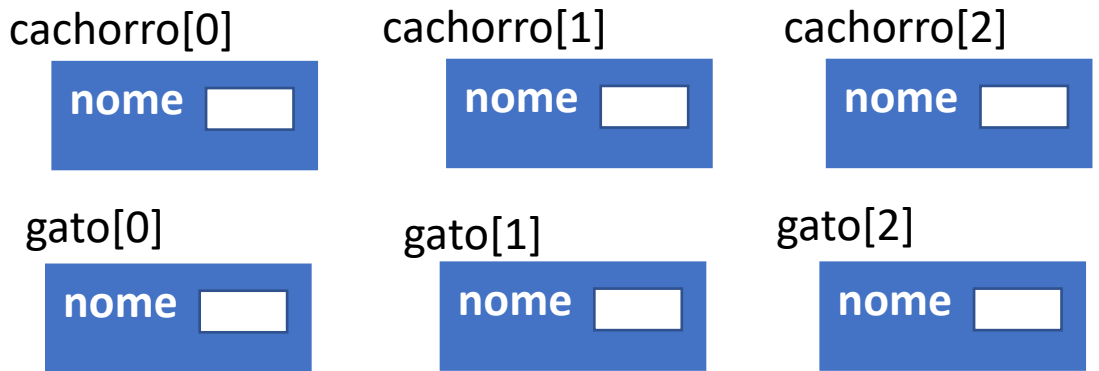
Desafio:

Prover método para cachorro e gato falarem

Planeje onde dever ser implementada a operação para fazer o animal falar




```
...
Cachorro cachorro[3];
leCachorros(cachorro);
...
Gato gato[3];
leGatos(gato);
...
falaAnimais(cachorro);
falaAnimais(gato);


void falaAnimais(Animal animal[]){
  for(int i=0; i<3; i++){
     cout << endl << animal[i].fala();
  }
}
```



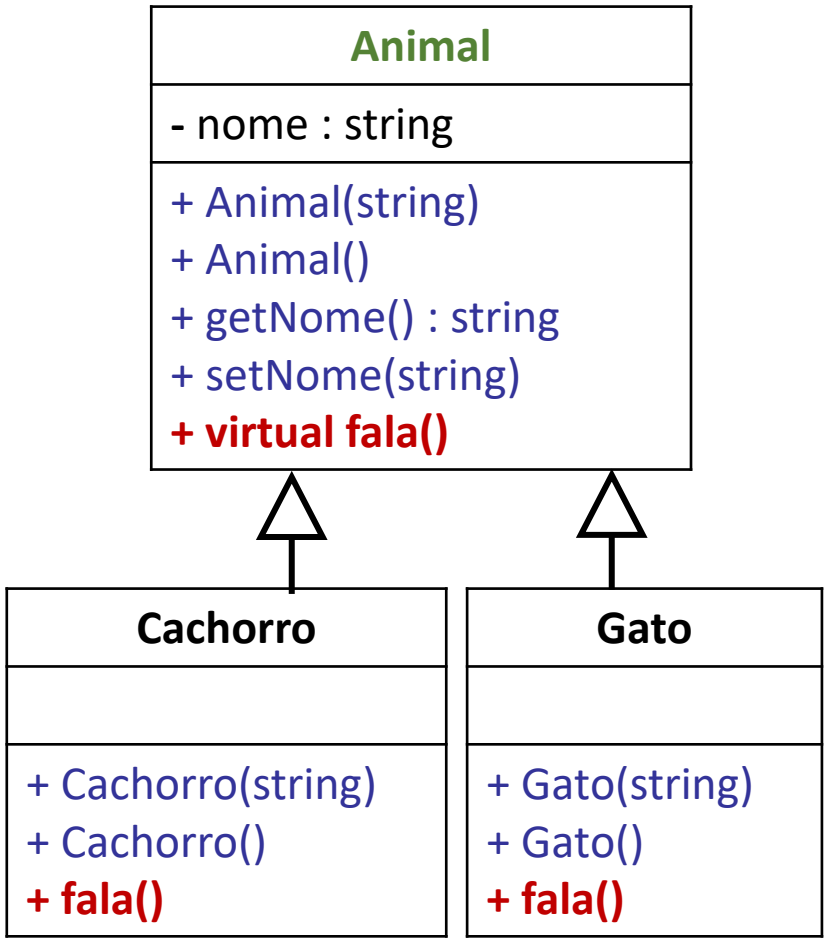
Função Virtual

Considere que uma classe derivada tenha uma função de igual assinatura da classe base.

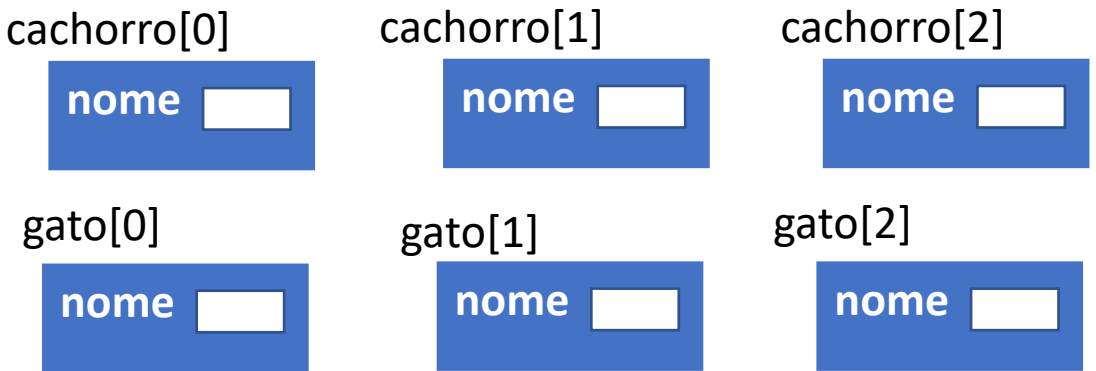
Em caso de ela ser evocada sobre uma instância da classe derivada, qual das duas será executada?

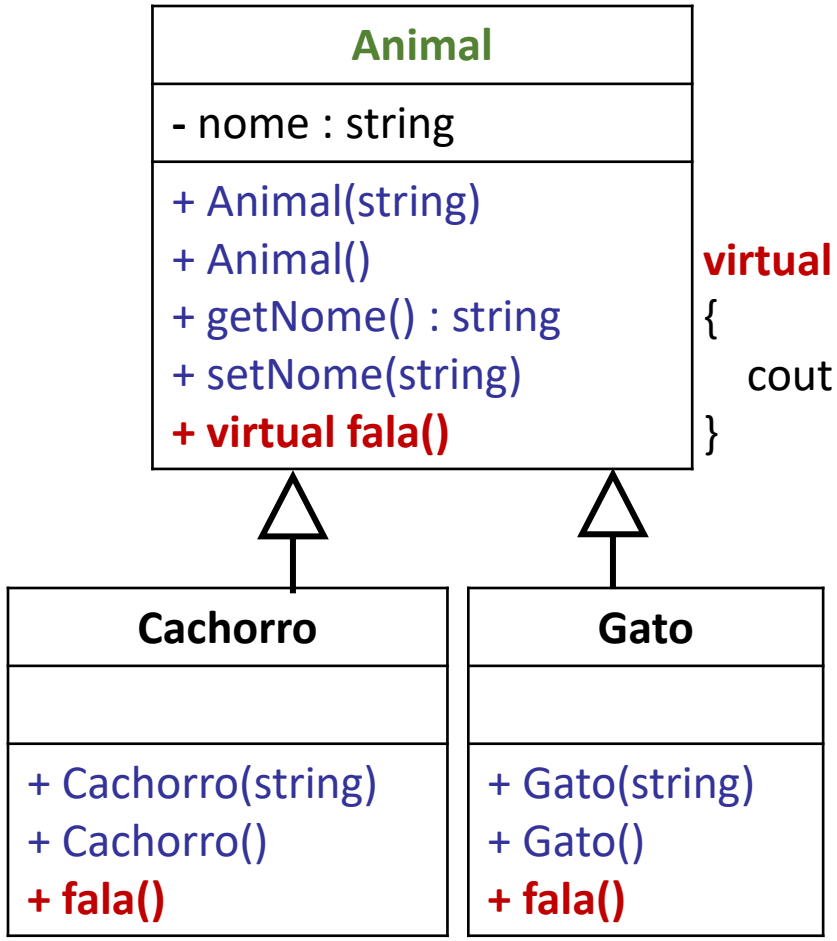
* Uma Função Virtual é aquela que autoriza ser ela especializada na classe derivada. Será ela executada ainda que o objeto seja do tipo de sua classe base: essência do Polimorfismo

Planeje onde dever ser implementada a operação para fazer o animal falar




```
...
Cachorro cachorro[3];
leCachorros(cachorro);
...
Gato gato[3];
leGatos(gato);
...
falaAnimais(cachorro);
falaAnimais(gato);


void falaAnimais(Animal animal[]){
  for(int i=0; i<3; i++){
     cout << endl << animal[i].fala();
  }
}
```





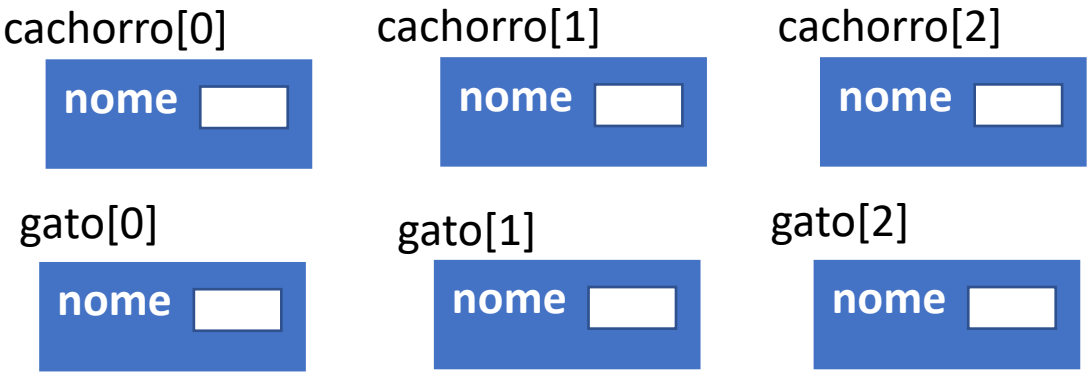

```
virtual void fala()
{
    cout << endl << " ~ ";
}
```

```
void fala() {
   cout << endl << " Au Au ";
}
```

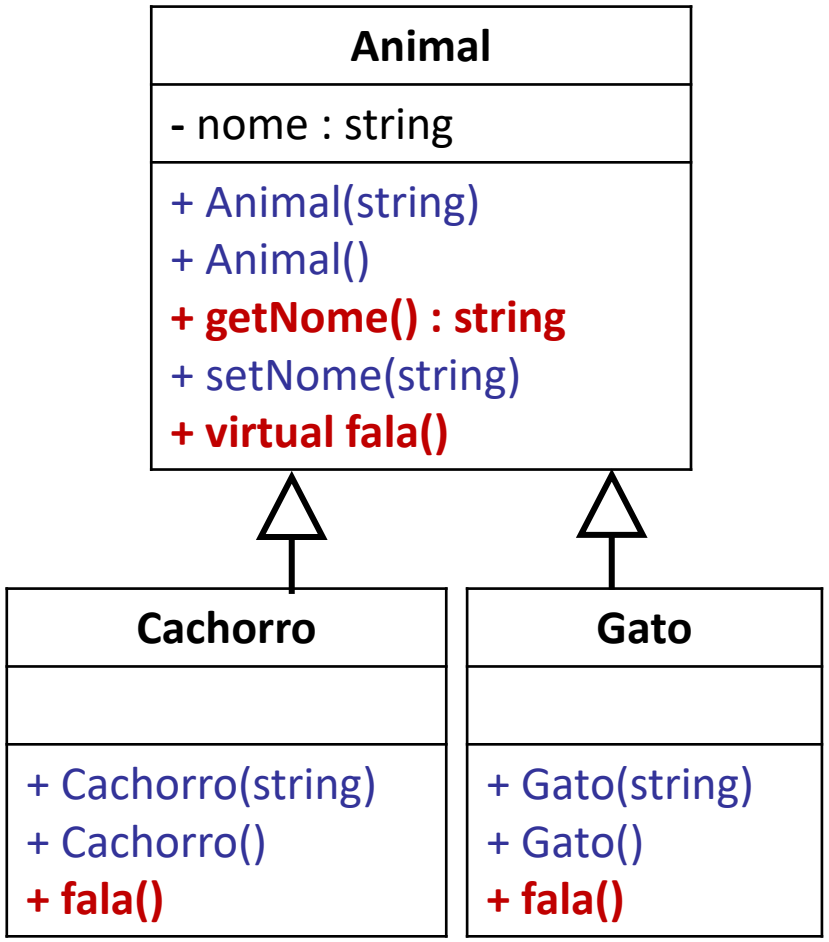
```
void fala() {
    cout << endl << " Miau Miau";
}
```

```
...
Cachorro cachorro[3];
leCachorros(cachorro);
...
Gato gato[3];
leGatos(gato);
...
falaAnimais(cachorro);
falaAnimais(gato);
```

```
void falaAnimais(Animal animal[]){
  for(int i=0; i<3; i++){
     cout << endl << animal[i].fala();
   }
}
```

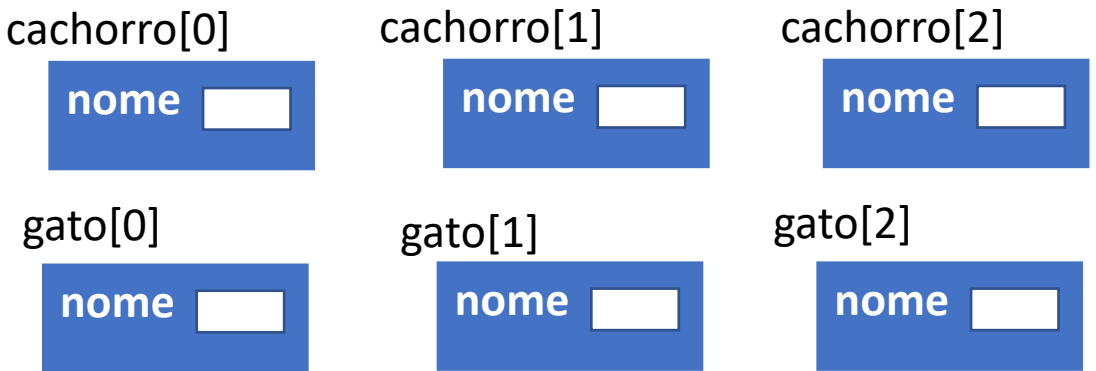


Analise as funções ao lado sob o ponto de vista do polimorfismo




```
void falar(Animal* animal)
{
    cout << endl << animal->fala();
}
```

```
void nomear(Animal* animal)
{
    cout << endl << animal->getNome();
}
```



Função Virtual Pura
Função Abstrata

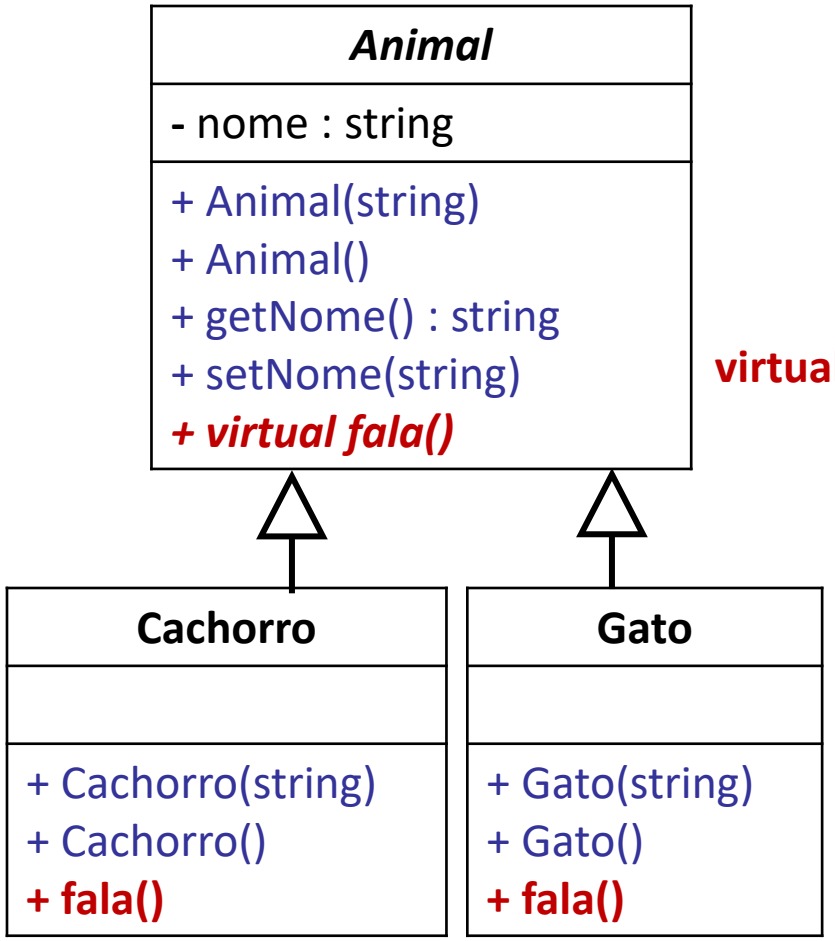
Função que adia a implementação do serviço (operação) para as classes derivadas (subclasses).

Em C++: sua assinatura é igualada a zero.

UML: apresentada em itálico

Uma classe é dita abstrata em contraposição à classe concreta. Ela não autoriza ser instanciada. Logo, é útil para prover polimorfismo ou para prover interface.

UML: classe descrita em itálico

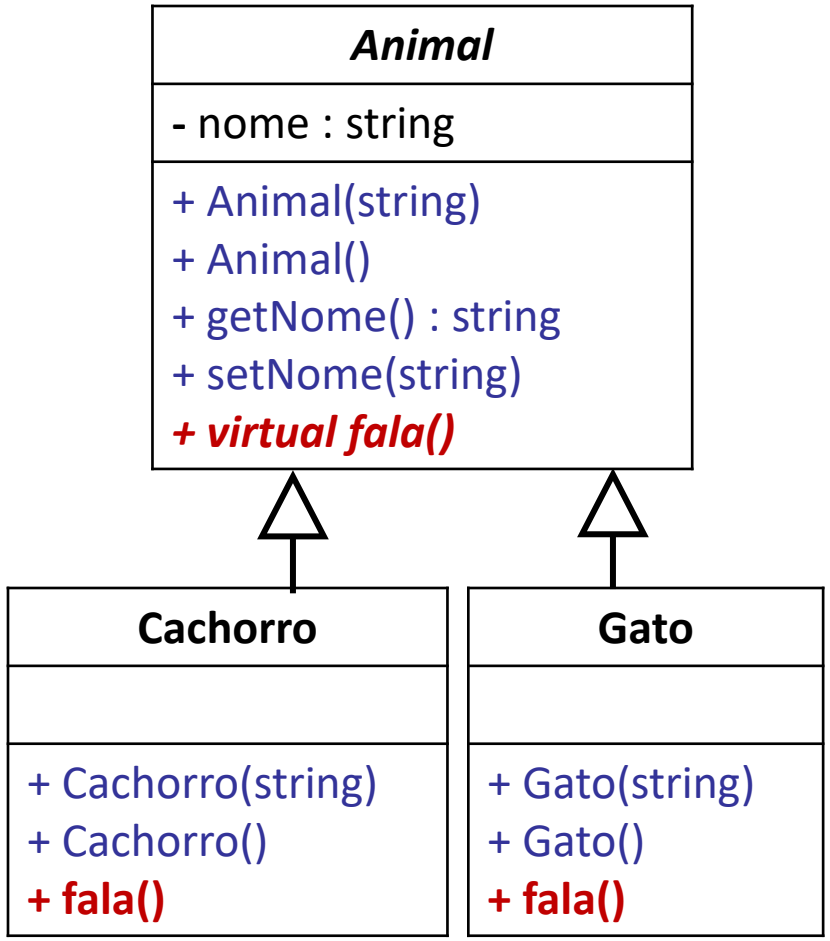


virtual void fala() = 0;

```
class Cachorro : public Animal {
  public :
    Cachorro(string nome) : Animal(nome) {   }
    Cachorro() {  }
    void fala() {
       cout << endl << " Au Au ";
    }
};
```

```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
          this->nome= nome;
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
     virtual void fala() = 0;
};
```

```
class Gato : public Animal {
   public :
      Gato(string nome) : Animal(nome) {   }
      Gato() { }
       void fala() {
          cout << endl << " Miau ";
       }
};
```




```
class Animal {
  private :
     string nome;
  public :
     Animal(string nome) {
          this->nome= nome;
     }
     Animal() { }
     string getNome() {
         return nome;
     }
     void setNome(string nome){
       this->nome= nome;
     }
     virtual void fala() = 0;
};
```

Analise e avalie cada instrução abaixo:

```
Animal animal;
Animal* animal;

Animal* animal= new Animal;
Animal* animal= new Cachorro;
Animal* animal= new Gato;

animal.fala();
animal->fala();

void falar(Animal* animal){
     animal->fala();
}
```

```
class Cachorro : public Animal {
  public :
     Cachorro(string nome) : Animal(nome) { }
     Cachorro() {  }
     void fala() {
        cout << endl << " Au Au ";
     }
};
```

```
class Gato : public Animal {
    public :
       Gato(string nome) : Animal(nome) { }
       Gato() { }
        void fala() {
           cout << endl << " Miau ";
        }
 };
```

Questões acerca de Funções Abstratas, Classes Abstratas, Interface e Herança Múltipla

Uma classe com pelo menos uma função abstrata, pode ser uma classe concreta?

Uma classe abstrata pode implementar uma interface?

Interface e Herança Múltipla

Próximos passos

Trabalho Prático Final

Implementar polimorfismo

A funcionalidade *Aniversariantes do Mês* deverá listar todos os aniversariantes: ambos, alunos e professores.

Para isto, um único *array* do tipo Pessoa deverá reunir todos os alunos e todos os professores.

O submenu deverá oferecer a seguinte opção:

1 - Cadastrar uma pessoa
    1.1 - Cadastrar Professor
    1.2 - Cadastrar Aluno
2 - Listar todas as pessoas
    2.1 – Listar Professores
    2.2 – Listar Alunos
3 – Pesquisar por nome
    3.1 – Pesquisar Professores por nome
    3.2 – Pesquisar Alunos por nome
4 – Pesquisar por CPF
    4.1 – Pesquisar Professores por CPF
    4.2 – Pesquisar Alunos por CPF
5 – Excluir pessoa
    5.1 – Excluir Professor (pelo CPF)
    5.2 – Excluir Aluno (pelo CPF)
6 - Apagar todas as pessoas cadastradas
    6.1 – Excluir todos os Professores
    6.2 – Excluir todos os Alunos
7 – Aniversariantes do mês
    7.1 – Informar o mês a ser pesquisado
    7.2 – Listar os aniversariantes do mês

Próximas aulas

Quarta, dia 4/dez

Dicas diversas de C e C++

Quinta e sexta, dias 5 e 6/dez, laboratório

Tratamento de Exceção

Segunda, dia 9/dez

Exercícios diversos

Quarta, dia 11/dez

Prova III (25 pontos)

Quinta e sexta, dias 12 e 13/dez, laboratório

Apresentação do Trabalho Prático

Implementação: 2 pontos

Três perguntas sorteadas: 3 pontos

Segunda, dia 16/dez

Correção e entrega da Prova III

Quarta, dia 18/dez

Reavaliação (apenas para quem não obteve aprovação)

25 pontos – Substitui a menor nota

Quinta, dia 19 e 20/dez, Laboratório

Atendimento a todos: entrega das provas e solução de pendências