

IMD0030

LINGUAGEM DE PROGRAMAÇÃO I

Aula 16 – Herança e métodos abstratos
(material baseado nas notas de aula do Prof. Silvio Sampaio)

Aulas anteriores

- Vimos que o **paradigma de Programação Orientada a Objetos (POO)** surgiu com o objetivo principal de facilitar o desenvolvimento de programas
 - agregando novos conceitos para a representação de elementos do mundo real de forma mais intuitiva
 - procurando melhorar a produtividade e a qualidade no desenvolvimento de *software*
 - Vimos que a solução de problemas utilizando POO é essencialmente baseada na **abstração**
 - das **entidades** do mundo real a serem representadas no programa
 - dos **dados e características** associados a tais entidades
 - das **ações** que podem ser realizadas por tais entidades
-

Aulas anteriores

Vimos que, em POO,

- **classes** representam entidades do mundo real
- **objetos** são instâncias de classes e representam indivíduos de uma entidade
- dados e características associados a um objeto são representados como **atributos** de classes
- as ações que podem ser realizadas por um objeto são implementadas como **métodos** de classes

Conceito	Elemento do mundo real
Entidade	Carro
Indivíduos	Carro X, carro Y, Carro Z, etc.
Dados e características	Cor, modelo, ano, placa, proprietário
Ações	Ligar, andar, parar

```
class Carro {                                int main {
    string cor;                               Carro x;
    string modelo;                           Carro y;
    string ano;                              Carro z;
    string placa;
    Pessoa proprietario;                     x.ligar();
    void ligar();                             x.andar();
    void andar();                             x.parar();
    void parar();                             };
};
```

Mas e se...

...quiséssemos criar classes para representar um caminhão e uma moto, de forma similar a um carro?

```
class Caminhao {  
    string cor;  
    string modelo;  
    string ano;  
    string placa;  
    Pessoa proprietario;  
    double capacidadeKg;  
    int qtdeEixos;  
    void ligar();  
    void andar();  
    void parar();  
};
```

```
class Moto {  
    string cor;  
    string modelo;  
    string ano;  
    string placa;  
    Pessoa proprietario;  
    double cilindradas;  
    void ligar();  
    void andar();  
    void parar();  
};
```

Qual o problema dessa abordagem?

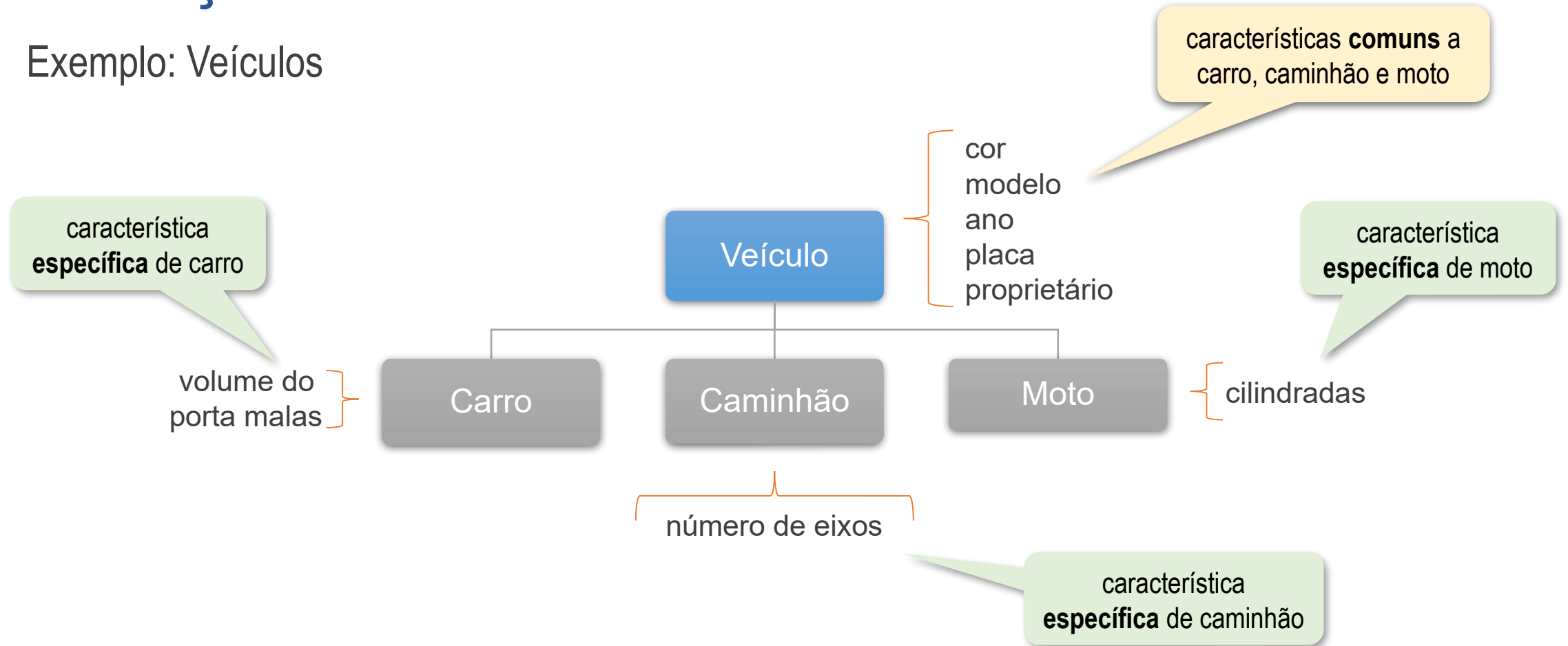
As classes `Carro`, `Caminhao` e `Moto` possuem **membros idênticos**, havendo **repetição de código**

Herança

- Mecanismo existente em POO que permite que uma classe **herde** membros (atributos e/ou métodos) de outra classe
 - Objetivos: aumentar **reuso**, produtividade e simplicidade na programação
 - Na herança
 - membros comuns a diferentes classes são reunidos em uma única classe, conhecida como **classe base** ou **superclasse**
 - a partir da classe base, outras classes (chamadas **classes derivadas** ou **subclasses**) podem ser definidas possuindo os mesmos membros especificados na classe base
 - as classes derivadas podem conter membros que sejam **particulares** a elas, ou seja, não são compartilhados com as outras classes derivadas
-

Herança

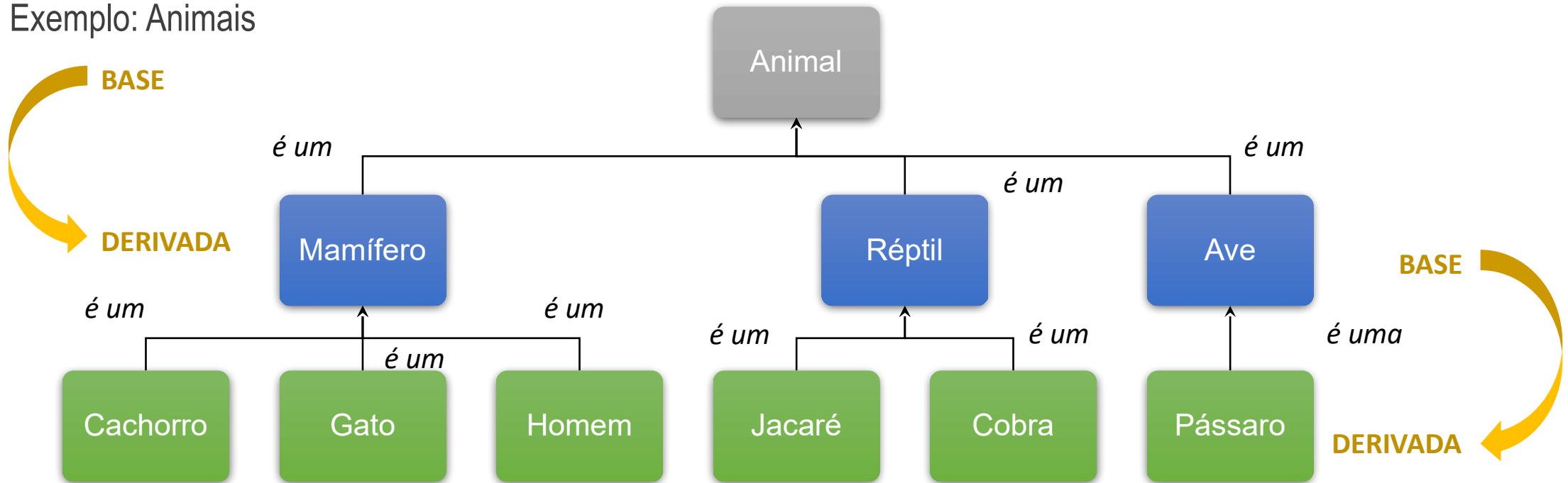
Exemplo: Veículos



Herança

O relacionamento entre objetos de classes base e de classes derivadas é tipicamente chamado de **é-um**

Exemplo: Animais



Como implementar herança entre classes em C++

- A classe base é **implementada como uma classe comum**
 - Uma classe derivada pode também ser uma classe base
- Além dos modificadores de acesso já apresentados (*public* e *private*), existe um outro especificador relacionado estritamente ao conceito de herança: ***protected***
 - Membros ***protected*** são visíveis às classes derivadas, enquanto que membros privados (*private*) não
 - Se os membros forem definidos como privados (*private*), **apenas os métodos da classe base** terão acesso a eles

```
#include <string>
using std::string;

#include "pessoa.h"

class Veiculo {
    protected:
        string cor;
        string modelo;
        string ano;
        string placa;
        Pessoa proprietario;
    public:
        void ligar();
        void andar();
        void parar();
};
```

Como implementar herança entre classes em C++

- A criação de classes derivadas que herdam de uma classe base é feita acrescentando-se
 - um modificador de acesso (tipicamente *public*)
 - o operador : (dois-pontos)
- As classes derivadas podem ter **novos atributos e métodos** além dos já existentes na classe base e que foram herdados
 - Os membros públicos e protegidos definidos na classe base são herdados pela classe derivada

```
#include <iostream>
#include <string>

using namespace std;

class Carro : public Veiculo {
    private:
        double volumePortaMalas;
    public:
        string getVolumePortaMalas();
        void setVolumePortaMalas(double v);
};
```

Como implementar herança entre classes em C++

- As classes derivadas podem **sobrescrever** métodos definidos na classe base
 - A classe derivada **redefine a implementação** do método
- O método tem de ter **exatamente a mesma assinatura**
 - Se um objeto da classe base invoca o método, é executada a versão da classe base
 - Se um objeto da classe derivada invoca o método, é executada a versão da classe derivada

```
#include <iostream>
#include <string>

using namespace std;

class Carro : public Veiculo {
    private:
        double volumePortaMalas;
    public:
        string getVolumePortaMalas();
        void setVolumePortaMalas(double v);
        void ligar();
        void andar();
        void parar();
};

void Carro::ligar() {
    cout << "Carro foi ligado" << endl;
}
```

Como implementar herança entre classes em C++

- Resumo dos modificadores de acesso aplicados aos membros de uma classe

Tipo de acesso	<i>public</i>	<i>private</i>	<i>protected</i>
Membros da mesma classe	SIM	SIM	SIM
Membros de classes derivadas	SIM	NÃO	SIM
Não membros	SIM	NÃO	NÃO

- Modificadores de acesso também são aplicados à classe base

```
class classeDerivada : <tipo_acesso> classeBase { ... };
```

- ***public*** : não altera a visibilidade dos membros da classe
 - ***private***: herda os membros públicos (***public***) e protegidos (***protected***) como privados
 - ***protected***: herda os membros públicos (***public***) e protegidos (***protected***) como protegidos
-

Como implementar herança entre classes em C++

- A instanciação de objetos de classes derivadas é feita normalmente
 - Acesso a atributos e métodos da própria classe
 - Acesso a atributos e métodos da classe base com visibilidade protegida

```
int main() {  
    Carro c;  
    c.setModelo("Toyota Corolla");  
    c.setPlaca("ABC-1234");  
    c.setVolumePortaMalas(22.50);  
    c.ligar();  
  
    return 0;  
}
```

membros definidos na
classe base Veiculo

membro específico da
classe derivada Carro

método da classe base Veiculo
sobrescrito pela classe derivada Carro

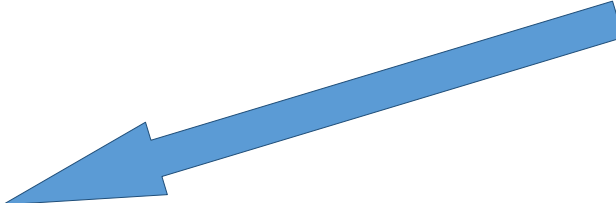
Upcasting e downcasting

- C++ permite que um ponteiro (ou referência) para uma classe derivada seja tratado como um ponteiro para a classe base
 - Isso é upcasting
- Downcasting é o processo oposto, no qual um ponteiro (ou referência) para a classe base é convertido para um ponteiro para a classe derivada

Upcasting

- Quando é feito um upcasting, os membros de um objeto não são modificados
 - Assim, quando um upcast é realizado, **será somente possível acessar membros que estão definidos na classe base**

```
int main() {  
    Veiculo* v = new Carro;  
    v->setModelo("Toyota Corolla");  
    v->setPlaca("ABC-1234");  
    v->setVolumePortaMalas (22.50); // ERRO: Pois upcasting foi utilizado  
    v->ligar();  
  
    return 0;  
}
```



Upcasting permite manipular a classe derivada como se fosse a sua classe base

Downcasting

- Downcasting não é seguro como um upcasting
 - No upcasting há a garantia de que um objeto da classe derivada pode sempre ser tratado como um objeto da classe base, uma vez que todas as classes derivadas herdam os mesmos membros da classe base
 - Entretanto, no caso oposto (downcasting) não se pode dizer o mesmo
 - No exemplo aqui usado: Todo Carro é um Veículo, mas nem todo Veículo é um Carro

```
Veiculo* v = new Veiculo;  
  
Carro* c1 = (Carro*) (v);  
c1->setModelo("Toyota Corolla");  
c1->setPlaca("ABC-1234");  
c1->setVolumePortaMalas(22.50);  
c1->ligar();
```

V é um veículo
(superclasse),
mas estamos
fazendo casting
dele para Carro
(subclasse)

Como implementar herança entre classes em C++

Nem tudo é herdado quando se declara uma classe derivada:

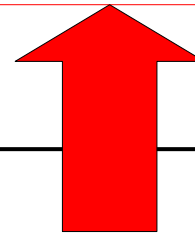
- Construtores
- Destrutores
- Relacionamentos *friend*
- Atributos com visibilidade privada (*private*)

Métodos construtores e destrutores em herança

- Se a classe for derivada de alguma outra, o método construtor da classe base é invocado **antes** do método construtor da classe derivada
 - Se a classe base também é derivada de outra, o processo é repetido recursivamente até que uma classe base não derivada seja alcançada
 - Se uma classe base não possui um método construtor padrão, a classe derivada deve **obrigatoriamente** definir um método construtor padrão, ainda que vazio
 - No caso dos métodos destrutores, a ordem de chamada é invertida: invoca-se primeiro o método destrutor da classe derivada e depois o método destrutor da classe base
-

Exemplo

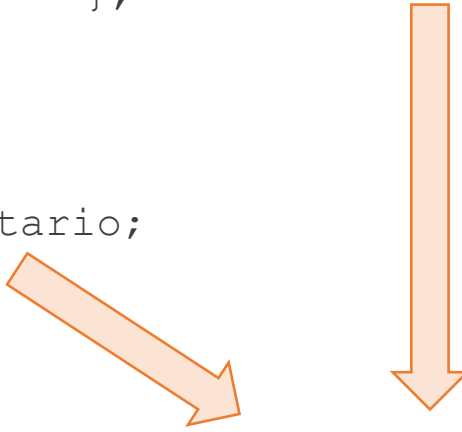
```
class Carro : public Veiculo{  
public:  
    Carro(int param, int rodas);  
    };  
  
    Carro::Carro(int outroParam, int rodas) : Veiculo(rodas){  
        //código  
    }
```



Herança múltipla

- Os tipos de herança vistos até então são chamados de **herança simples**: a classe derivada herda de apenas uma classe base
- A linguagem C++ permite realizar **herança múltipla**: uma mesma classe derivada pode herdar de mais de uma classe ao mesmo tempo
 - Após o operador : (dois-pontos), segue lista com nomes das classes base das quais a classe derivada irá herdar todos os atributos e métodos públicos ou protegidos

```
class BemMove1 {  
    protected:  
        float preco;  
        string codReceita;  
};  
  
class Veiculo {  
    protected:  
        string cor;  
        string modelo;  
        string ano;  
        string placa;  
        Pessoa proprietario;  
};  
  
class Carro : public Veiculo, BemMove1 {  
    protected:  
        string classificacao;  
    public:  
        // metodos da classe  
};
```



Métodos virtuais

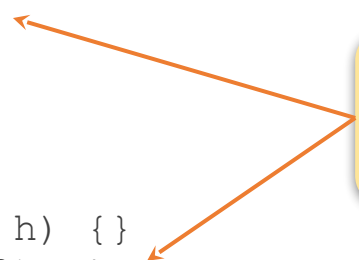
- Um **método virtual** é um método de uma classe base que pode ser **redefinido** pelas suas classes derivadas. Ao se referir a um objeto da classe derivada utilizando-se um ponteiro para a classe base, pode-se invocar a função virtual para aquele objeto e executar a versão da classe derivada para aquela função.
 - A definição de métodos virtuais é feita utilizando-se a palavra-chave `virtual` **antes** do tipo de retorno do método
 - Não é necessário adicionar a palavra-chave à assinatura dos métodos das classes derivadas
-

Métodos virtuais

```
class Poligono {  
    protected:  
        double largura;  
        double altura;  
    public:  
        Poligono(double a, double h) : largura(a), altura(h) {}  
        virtual double area() { return 0; }  
};
```

```
class Retangulo : public Poligono {  
    public:  
        Retangulo(double a, double h) : Poligono(a, h) {}  
        double area() { return largura * altura; }  
};
```

```
class Triangulo : public Poligono {  
    public:  
        Triangulo(double a, double h) : Poligono(a, h) {}  
        double area() { return (largura * altura / 2); }  
};
```



Redefinição do método `area` definido na classe base `Poligono` pelas classes derivadas `Retangulo` e `Triangulo`

Métodos virtuais

```
#include <iostream>
using std::cout;
using std::endl;

int main() {
    Poligono* r = new Retangulo(1, 2);
    Poligono* t = new Triangulo(3, 4);
    Poligono* p = new Poligono(2, 1);

    cout << "Area do retangulo: " << r->area() << endl;
    cout << "Area do triangulo: " << t->area() << endl;
    cout << "Area do poligono: " << p->area() << endl;

    return 0;
}
```

Resultado da execução:

```
$ ./poligono
Area do retangulo: 2
Area do triangulo: 6
Area do polígono: 0
```

Exercício

DIFICULDADE - FÁCIL

Crie uma classe chamada **Animal** que tem atributos privados **altura** e **peso**, e o método público virtual **andar**.

Construa a classe **Cachorro** que herda **Animal**, possui o atributo privado **nome** e define o método virtual **andar**.

Construa uma classe **Canil** que tem como atributo privado um vetor de **10 Animais**. Para facilitar, este vetor já é instanciado com 10 **Animais** no próprio construtor de **Canil**

Crie um método público chamado **ordenar** que ordena o vetor baseado no **nome**. **Teste exhaustivamente**.

DIFICULDADE - MÉDIO

Modifique **Canil** para que sempre seja inicializado com **0 Animais**, ie, um vetor vazio.

Crie o método público **adicionar** que adiciona um **Cachorro** (*passado como parâmetro*) no **Canil**. Este método sempre retorna **True**.

Crie um método público **remover**, que remove o **Cachorro** baseado em seu nome (parâmetro de entrada). Este método retorna **False** caso não exista o **Cachorro**, e **True** caso contrário. Sempre remova a primeira ocorrência do nome.

O tamanho do vetor deve ser sempre exatamente a quantidade de animais.
