



# Programação e Sistemas de Informação

CURSO PROFISSIONAL TÉCNICO DE  
GESTÃO E PROGRAMAÇÃO DE SISTEMAS INFORMÁTICOS

## Programação Orientada a Objetos

### MÓDULO 10

Professor: João Martiniano

# Conteúdos abordados neste módulo

- Herança
- *Sealed class*
- Polimorfismo
- Classes abstratas
- Associação de classes:
  - Agregação
  - Composição

# Introdução

- Os 3 conceitos fundamentais da programação orientada a objetos são:
  - encapsulamento: esconder os detalhes da implementação de um objeto
  - herança: reutilização das classes
  - polimorfismo: tratar objetos relacionados da mesma forma

| HERANÇA

# Herança

- A herança é um conceito fundamental da programação orientada a objetos
- Consiste em definir uma classe que herda as funcionalidades de outra classe
- A classe que herda pode utilizar ou redefinir as propriedades e métodos da classe herdada
- Ao contrário de outras linguagens de programação (C++, por exemplo) em C# uma classe apenas pode herdar de uma única classe

## Vantagens da herança

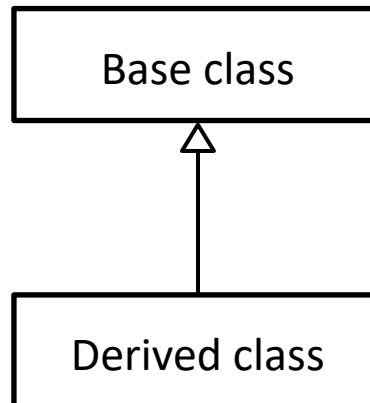
- Promove a reutilização do código o que permite poupar tempo e esforço

## Desvantagens da herança

- O código entre as classes fica muito interdependente
- Aumenta a complexidade do código
- Dificulta certo tipo de mudanças no código
- etc.

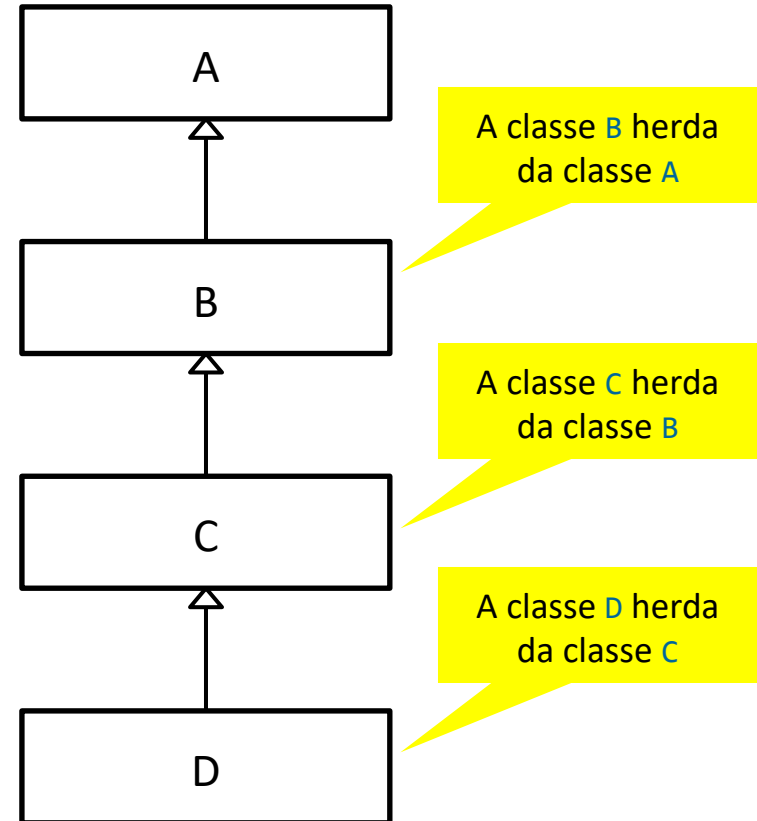
# Herança

- A classe que herda é a **derived class** (classe derivada)
- A classe a partir de onde se herda, é a **base class** (classe base)
- Num diagrama de classe a herança é representada da seguinte forma:



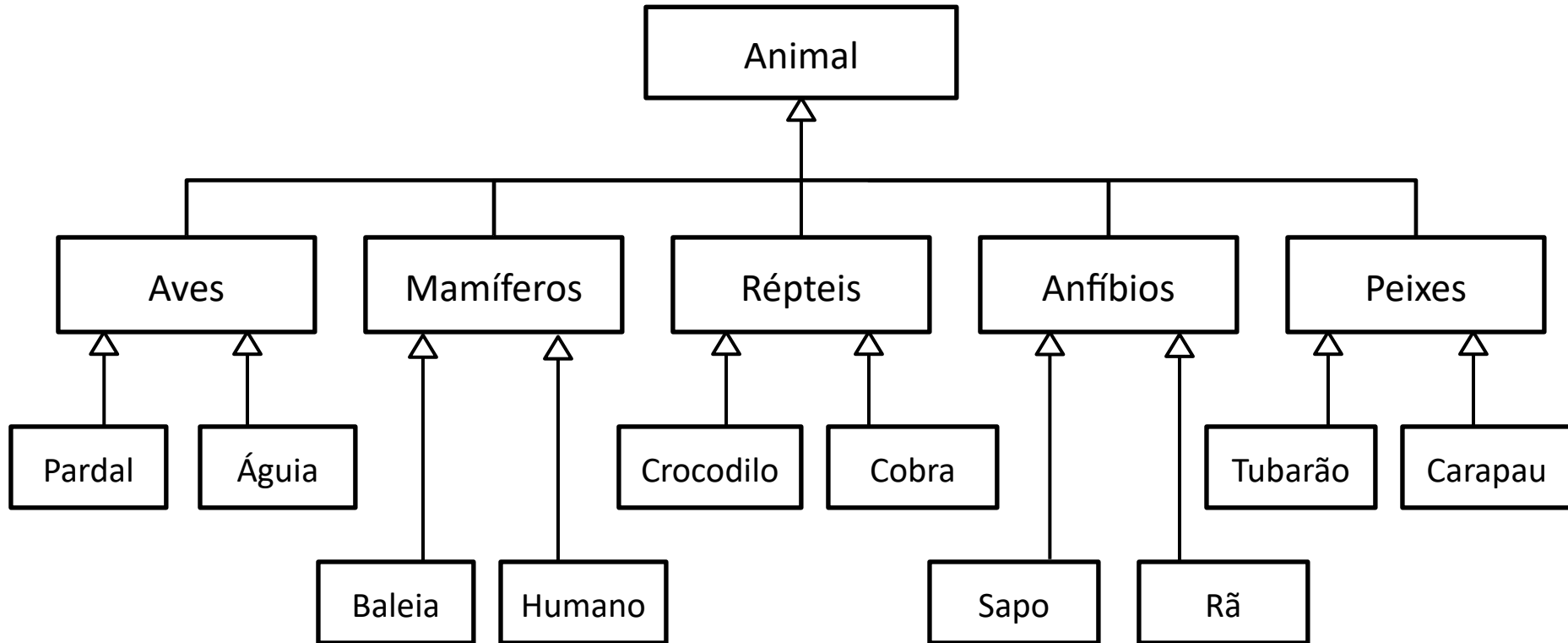
# Herança

- Podem ser definidos vários níveis de herança
- Por exemplo:
  - a classe **B** herda da classe **A**
  - a classe **C** herda da classe **B**
  - a classe **D** herda da classe **C**
  - etc.



# Herança

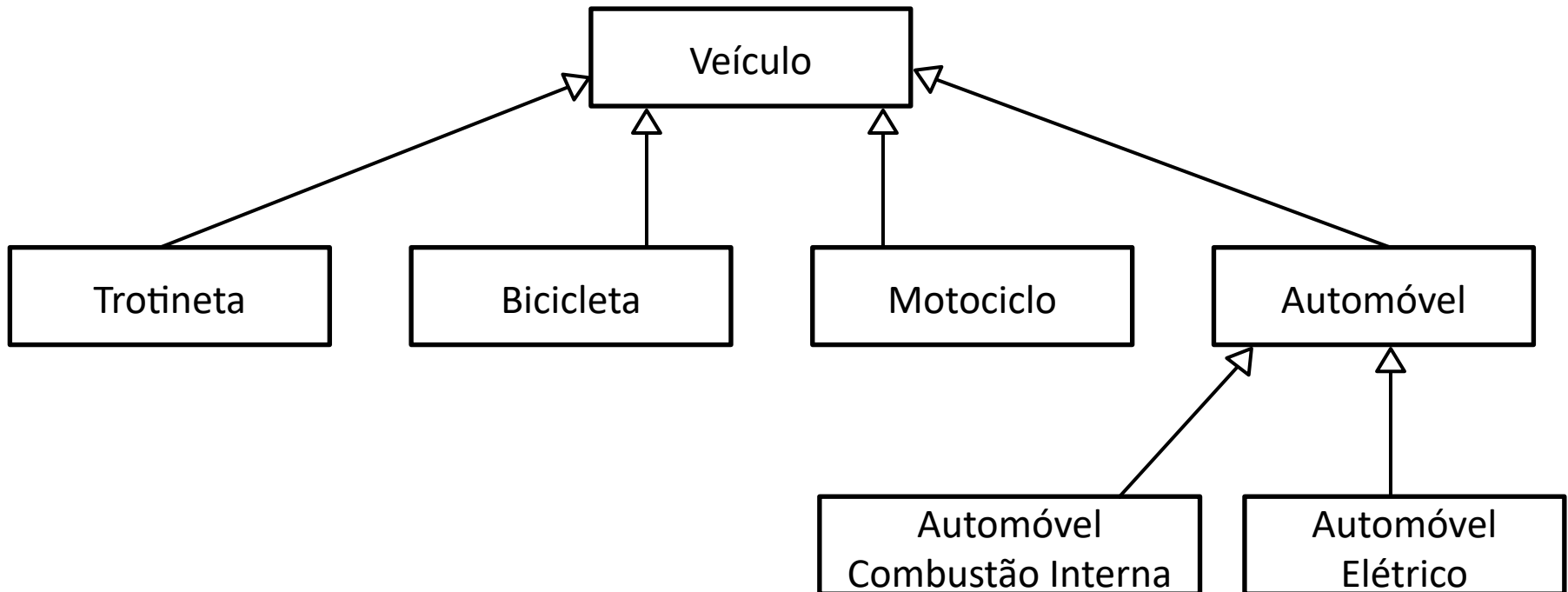
- Uma classe derivada representa **uma especialização de uma classe base**
- No mundo real encontramos várias formas de herança
- Por exemplo (utilizando um diagrama de classe):





# Herança

- Exemplo:

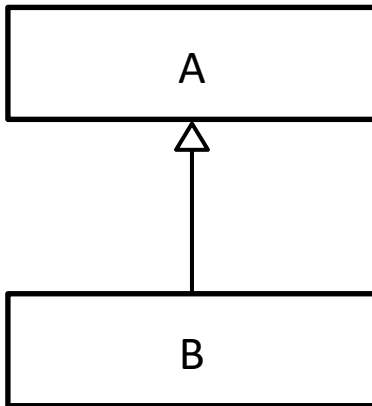


# Herança

- Sintaxe:

```
[access_modifier] class nome [: base_class]
{
    ...
}
```

- Exemplo:



```
class A
{
}

class B : A
{
}
```

A classe B herda a classe A

# Herança

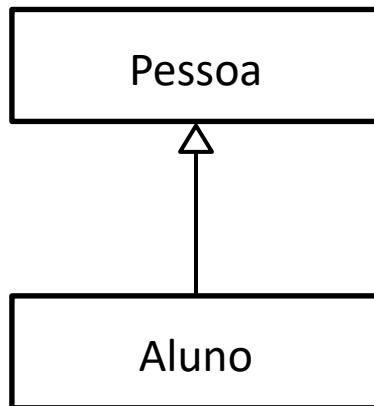
## Exercício

- Crie uma classe chamada **Aluno**, a qual é derivada da classe **Pessoa**
- Desenhe o diagrama de classe e escreva o código C#

# Herança

## Exercício: Resolução

- Crie uma classe chamada **Aluno**, a qual é derivada da classe **Pessoa**
- Desenhe o diagrama de classe e escreva o código C#

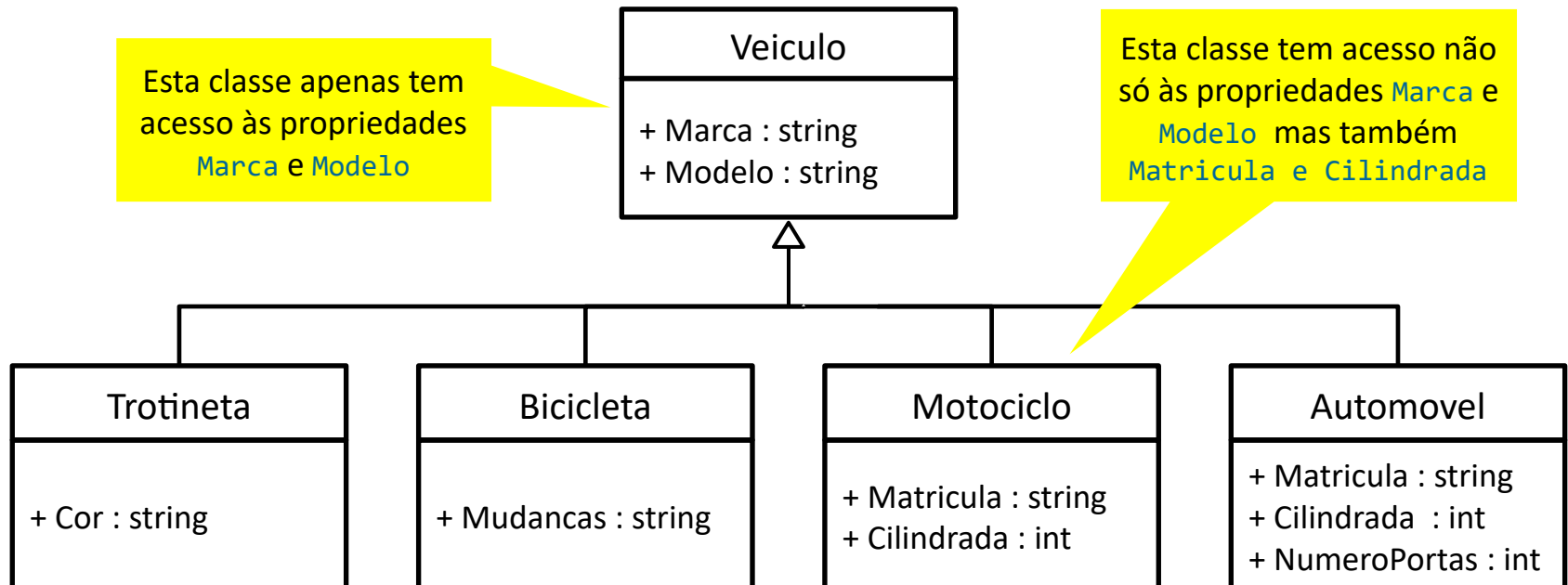


```
class Pessoa
{
}

class Aluno : Pessoa
{
}
```

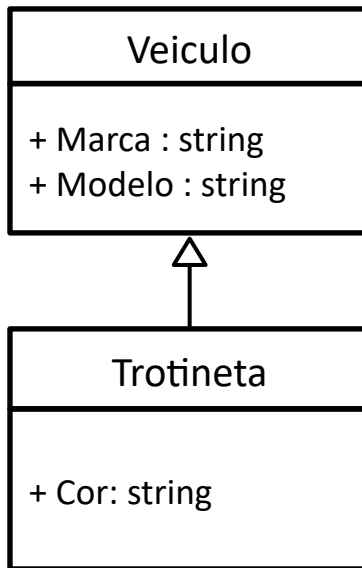
# Herança

- Uma classe derivada tem acesso aos membros da classe base
- Uma classe derivada pode definir os seus próprios membros e a classe base não tem acesso aos mesmos
- Por exemplo:



# Herança

- Exemplo: a classe `Trotineta` herda a classe `Veiculo`

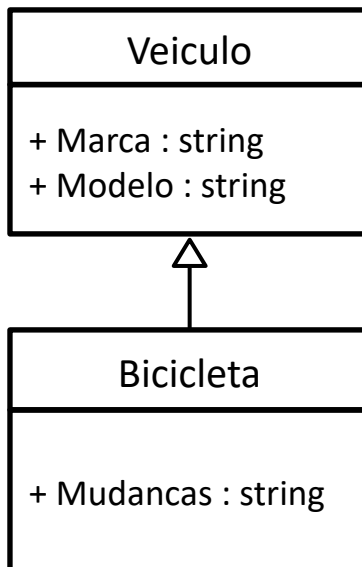


```
public class Veiculo
{
    public string Marca { get; set; }
    public string Modelo { get; set; }
}

public class Trotineta : Veiculo
{
    public string Cor { get; set; }
}
```

# Herança

- Exemplo: a classe `Bicicleta` herda a classe `Veiculo`

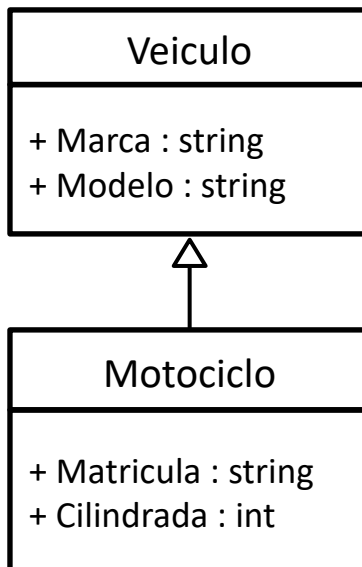


```
public class Veiculo
{
    public string Marca { get; set; }
    public string Modelo { get; set; }
}

public class Bicicleta : Veiculo
{
    public string Mudancas { get; set; }
}
```

# Herança

- Exemplo: a classe `Motociclo` herda a classe `Veiculo`



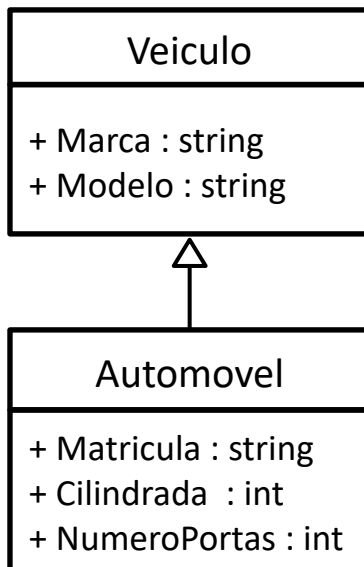
```
public class Veiculo
{
    public string Marca { get; set; }
    public string Modelo { get; set; }
}

public class Motociclo : Veiculo
{
    public string Matricula { get; set; }
    public int Cilindrada { get; set; }
}
```



# Herança

- Exemplo: a classe `Automovel` herda a classe `Veiculo`

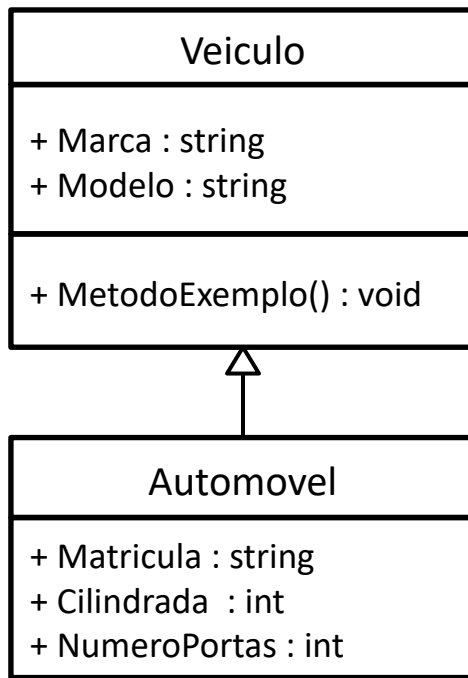


```
public class Veiculo
{
    public string Marca { get; set; }
    public string Modelo { get; set; }
}

public class Automovel : Veiculo
{
    public string Matricula { get; set; }
    public int Cilindrada { get; set; }
    public int NumeroPortas { get; set; }
}
```

# Herança: Acesso aos membros herdados

- Os membros que uma classe derivada herda, são acedidos normalmente como se tivessem sido declarados na própria classe
- Tomando como exemplo as classes **Veiculo** e **Automovel** e acrescentando um método à classe **Veiculo**:



```

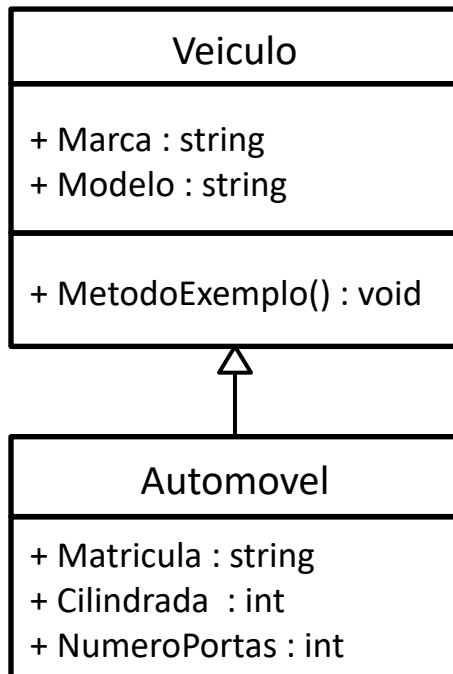
public class Veiculo
{
    public string Marca { get; set; }
    public string Modelo { get; set; }

    public void MetodoExemplo()
    { }
}

public class Automovel : Veiculo
{
    public string Matricula { get; set; }
    public int Cilindrada { get; set; }
    public int NumeroPortas { get; set; }
}
  
```

# Herança: Acesso aos membros herdados

- O exemplo seguinte demonstra o acesso aos vários membros da classe `Automovel`, incluindo o método herdado de `Veiculo`:



```

Automovel auto1 = new Automovel();

// Membros definidos na classe base
auto1.Marca = "Volvo";
auto1.Modelo = "XC-60 2.0 Dynamic";

// Membros definidos na classe Automovel
auto1.Matricula = "99-AA-00";
auto1.Cilindrada = 1969;
auto1.NumeroPortas = 5;

// Executar o método definido na classe base
auto1.MetodoExemplo();
    
```

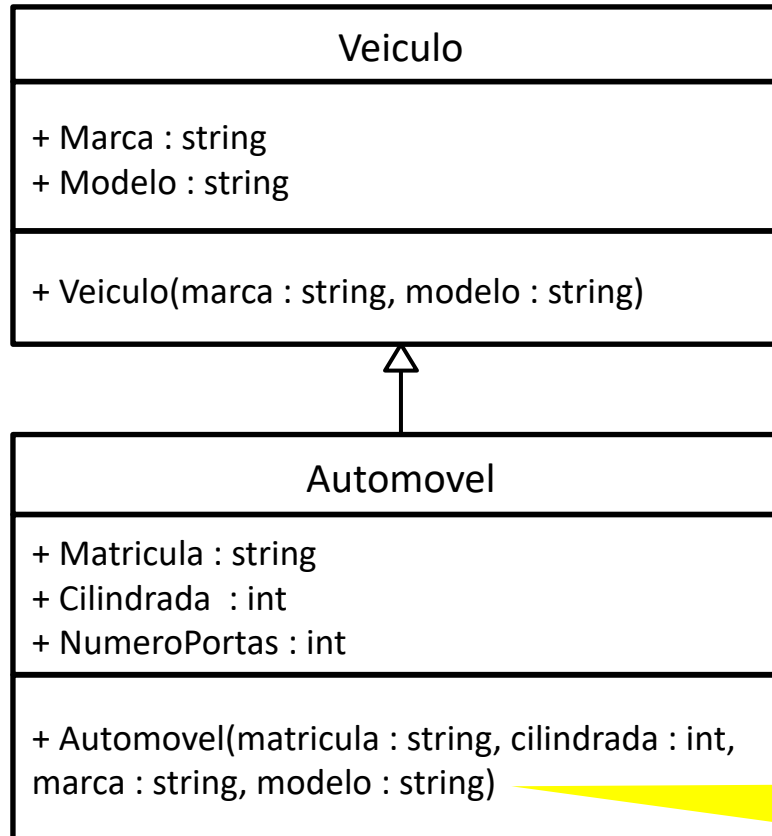
O método `MetodoExemplo()` é  
definido em `Veiculo`

## Herança: Invocar o construtor da *base class*

- É frequente, no construtor de uma classe derivada, invocar o construtor da *base class*
- Deste modo são inicializados os membros herdados

# Herança: Invocar o construtor da *base class*

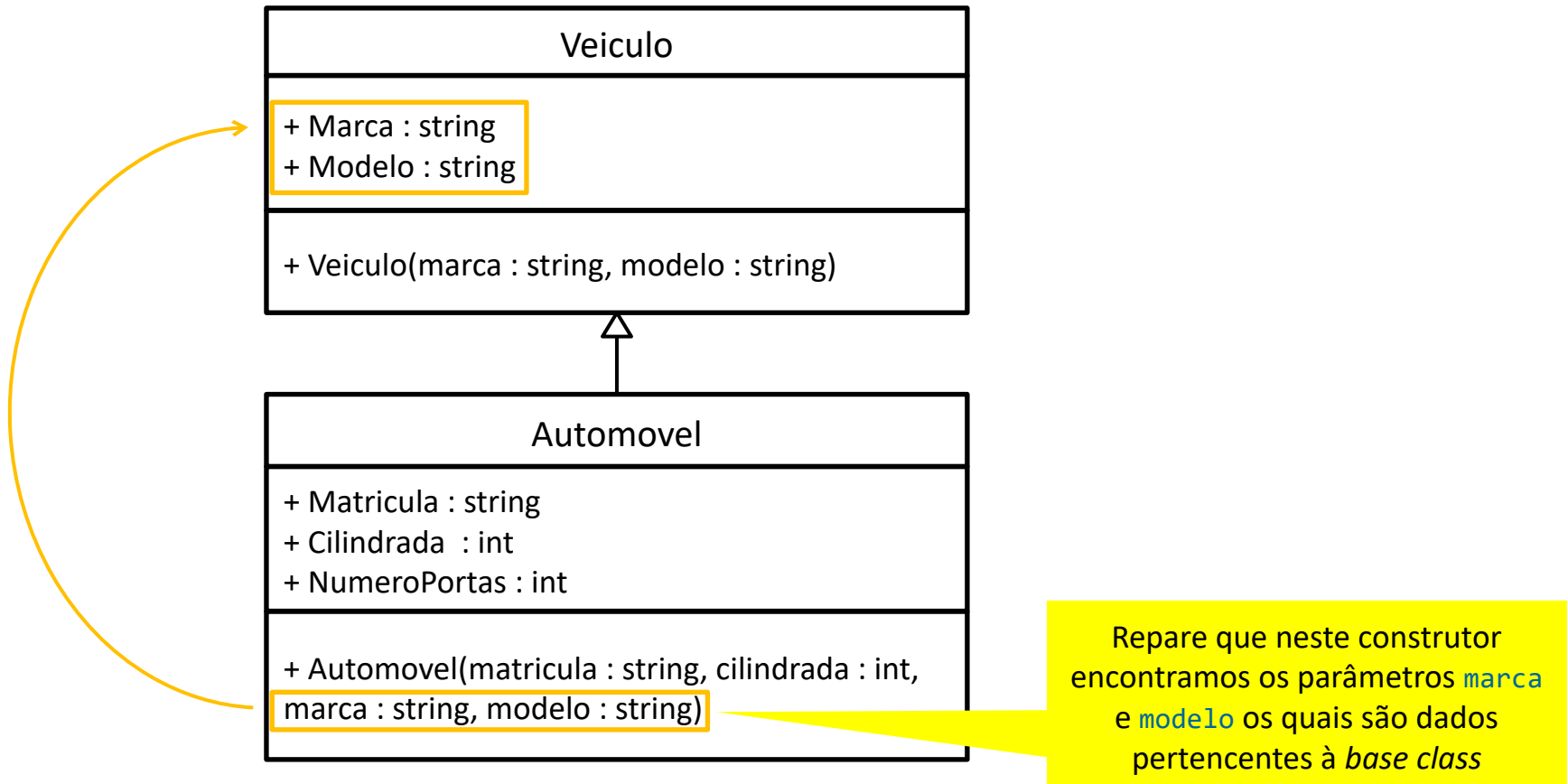
- Analisemos o seguinte exemplo:



Repare que neste construtor encontramos os parâmetros **marca** e **modelo** os quais são dados pertencentes à *base class*

# Herança: Invocar o construtor da *base class*

- Analisemos o seguinte exemplo:



# Herança: Invocar o construtor da *base class*

- Neste exemplo, o construtor da classe `Automovel` invoca o construtor da *base class* `Veiculo`:

```
public class Veiculo
{
    public string Marca { get; set; }
    public string Modelo { get; set; }

    public Veiculo(string marca, string modelo)
    {
        Marca = marca;
        Modelo = modelo;
    }
}

public class Automovel : Veiculo
{
    public string Matricula { get; set; }
    public int Cilindrada { get; set; }
    public int NumeroPortas { get; set; }

    public Automovel(string matricula, int cilindrada, int numeroPortas, string marca, string modelo)
        : base(marca, modelo)
    {
        Matricula = matricula;
        Cilindrada = cilindrada;
        NumeroPortas = numeroPortas;
    }
}
```

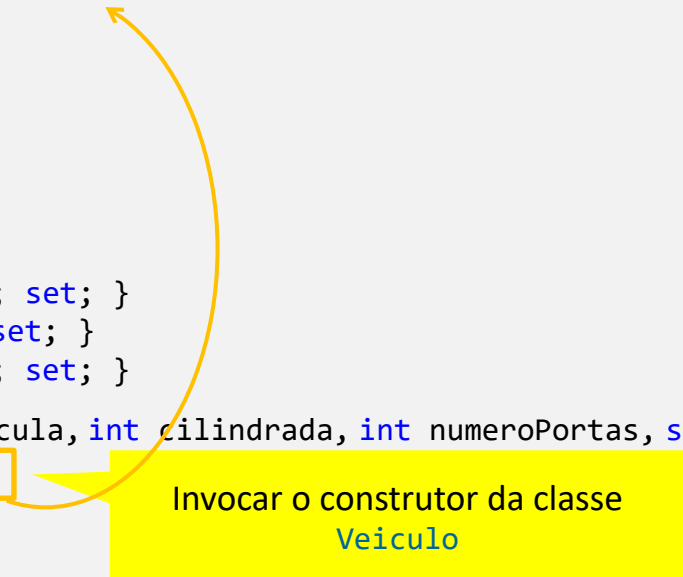
Invocar o construtor da classe `Veiculo`

# Herança: Invocar o construtor da *base class*

- Neste exemplo, o construtor da classe `Automovel` invoca o construtor da *base class* `Veiculo`:

```
public class Veiculo
{
    public string Marca { get; set; }
    public string Modelo { get; set; }
    public Veiculo(string marca, string modelo)
    {
        Marca = marca;
        Modelo = modelo;
    }
}

public class Automovel : Veiculo
{
    public string Matricula { get; set; }
    public int Cilindrada { get; set; }
    public int NumeroPortas { get; set; }
    public Automovel(string matricula, int cilindrada, int numeroPortas, string marca, string modelo)
        : base(marca, modelo)
    {
        Matricula = matricula;
        Cilindrada = cilindrada;
        NumeroPortas = numeroPortas;
    }
}
```

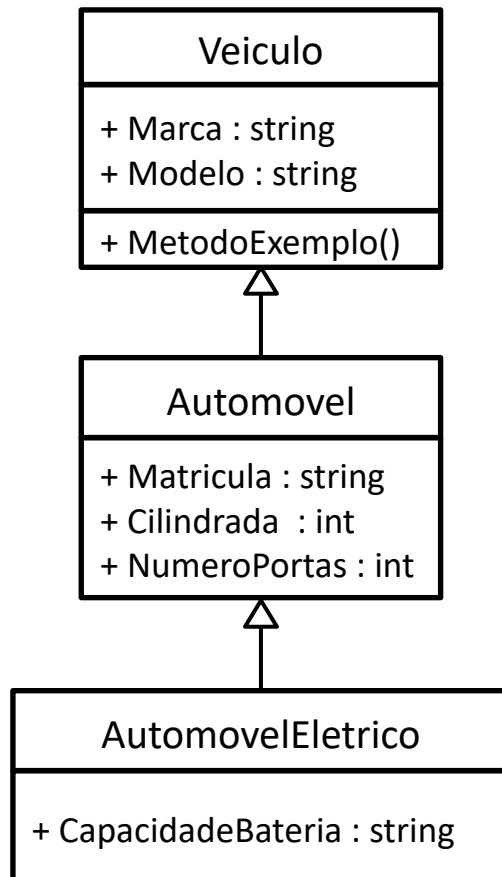


Invocar o construtor da classe `Veiculo`



# Herança: Níveis de herança

- Não existe limite quanto ao número de níveis de herança
- Exemplo com 3 níveis de herança:



```

public class Veiculo
{
    public string Marca { get; set; }
    public string Modelo { get; set; }

    public void MetodoExemplo()
    { }
}

public class Automovel : Veiculo
{
    public string Matricula { get; set; }
    public int Cilindrada { get; set; }
    public int NumeroPortas { get; set; }
}

public class AutomovelEletrico : Automovel
{
    public string CapacidadeBateria { get; set; }
}
    
```

# Herança: Níveis de herança

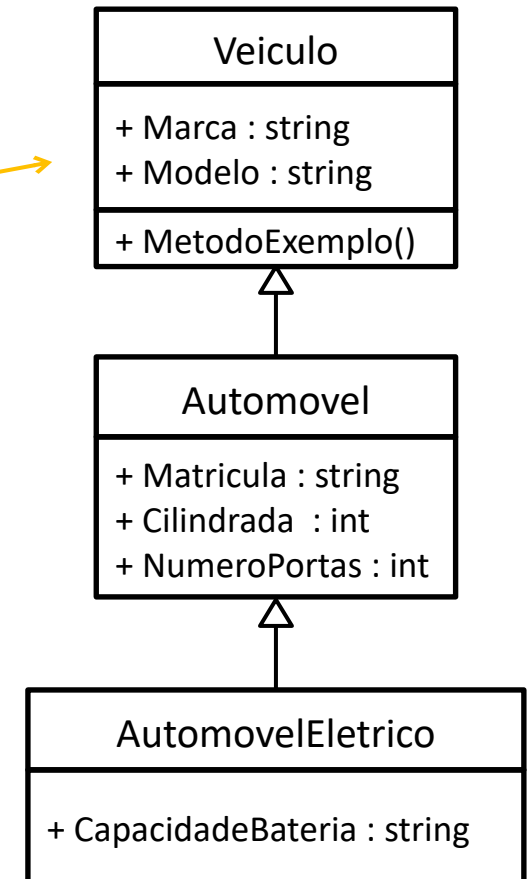
- Não existe limite quanto ao número de níveis de herança
- Exemplo com 3 níveis de herança (continuação):

```
AutomovelEletrico auto2 = new AutomovelEletrico();

auto2.Marca = "Renault";
auto2.Modelo = "ZOE Zen Z.E. 40 R110";

auto2.Matricula = "00-BB-99";
auto2.NumeroPortas = 5;

auto2.CapacidadeBateria = "41 kWh";
```



# Herança: Níveis de herança

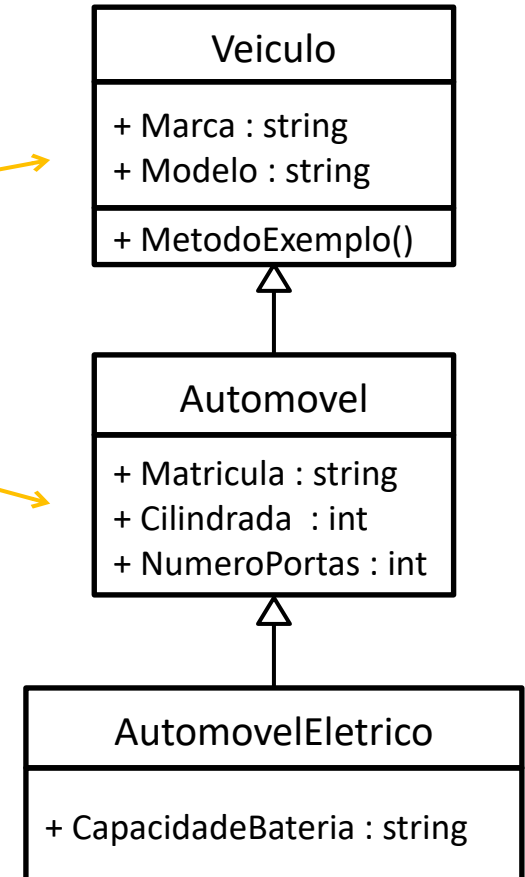
- Não existe limite quanto ao número de níveis de herança
- Exemplo com 3 níveis de herança (continuação):

```
AutomovelEletrico auto2 = new AutomovelEletrico();
```

```
auto2.Marca = "Renault";  
auto2.Modelo = "ZOE Zen Z.E. 40 R110";
```

```
auto2.Matricula = "00-BB-99";  
auto2.NumeroPortas = 5;
```

```
auto2.CapacidadeBateria = "41 kWh";
```



# Herança: Níveis de herança

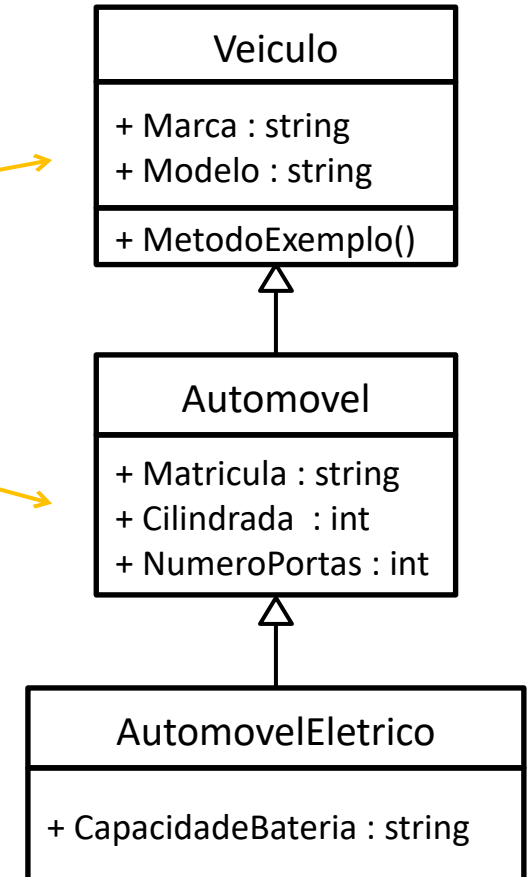
- Não existe limite quanto ao número de níveis de herança
- Exemplo com 3 níveis de herança (continuação):

```
AutomovelEletrico auto2 = new AutomovelEletrico();
```

```
auto2.Marca = "Renault";  
auto2.Modelo = "ZOE Zen Z.E. 40 R110";
```

```
auto2.Matricula = "00-BB-99";  
auto2.NumeroPortas = 5;
```

```
auto2.CapacidadeBateria = "41 kWh";
```



# Acesso aos membros herdados

- Uma classe derivada herda todos os membros de uma classe base
- Exceto os membros que tenham sido definidos como `private`
- Membros de uma classe que tenham sido definidos como `private` apenas podem ser acedidos pelos membros dessa classe
- Ou seja, não podem ser acedidos pelas classes derivadas

# Acesso aos membros herdados

- O seguinte exemplo demonstra uma classe derivada (**B**) a tentar ter acesso aos membros da *base class* (**A**)

```
public class A
{
    private int X { get; set; }
    public int Y { get; set; }

    public void Metodo1()
    {
        X = 5;
    }

    private void Metodo2()
    {
        // Faz qualquer coisa
    }
}
```

Atenção: propriedade privada

Atenção: método privado

```
public class B : A
{
    public void Metodo()
    {
        // A seguinte instrução não é válida
        X = 10;

        // A seguinte instrução é válida
        Y = 20;

        // A seguinte instrução não é válida
        Metodo2();
    }
}
```

# Acesso aos membros herdados

- O seguinte exemplo demonstra uma classe derivada (**B**) a tentar ter acesso aos membros da *base class* (**A**)

```
public class A
{
    private int X { get; set; }
    public int Y { get; set; }

    public void Metodo1()
    {
        X = 5;
    }

    private void Metodo2()
    {
        // Faz qualquer coisa
    }
}
```

Atenção: propriedade privada

Erro: tentativa de acesso a propriedade privada

Atenção: método privado

```
public class B : A
{
    public void Metodo()
    {
        // A seguinte instrução não é válida
        X = 10;

        // A seguinte instrução é válida
        Y = 20;

        // A seguinte instrução não é válida
        Metodo2();
    }
}
```

Erro: tentativa de acesso a método privado

## O *access modifier* `protected`

- O *access modifier* `protected` permite:
  - que um membro seja privado fora da classe (ou seja, está inacessível fora da classe)
  - mas que as classes derivadas possam ter acesso a esse membro

```
public class A
{
    protected int X { get; set; }
    public int Y { get; set; }

    public void Metodo1()
    {
        X = 5;
    }

    protected void Metodo2()
    {
        // Faz qualquer coisa
    }
}
```

Já é possível aceder à propriedade

```
public class B : A
{
    public void Metodo()
    {
        // A seguinte instrução é válida
        X = 10;

        // A seguinte instrução é válida
        Y = 20;

        // A seguinte instrução é válida
        Metodo2();
    }
}
```

Já é possível aceder ao método



# | *SEALED CLASS*

## Sealed class

- Uma *sealed class* é uma classe que não pode ser herdada
- É um mecanismo que sinaliza que a classe fornece todas as funcionalidades necessárias e não é necessário acrescentar mais funcionalidades através de herança
- É necessário ter em atenção que uma classe abstrata nunca pode ser *sealed*
- Sintaxe:

```
[access_modifier] sealed class nome  
{  
    ...  
}
```

- Exemplo:

```
public sealed class A  
{  
    // A classe pode conter os membros normais de uma classe normal  
}
```

## Sealed class

- No seguinte exemplo, definimos a classe `Trotineta` como sendo *sealed*

```
public class Veiculo
{
    public string Marca { get; set; }
    public string Modelo { get; set; }
}

public sealed class Trotineta : Veiculo
{
    public string Cor { get; set; }
}
```

*Sealed class*: nenhuma classe  
pode herdar da classe `Trotineta`

# | POLIMORFISMO

# Polimorfismo

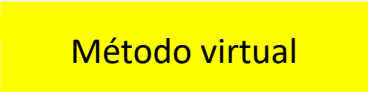
- A palavra **polimorfismo** significa algo que pode tomar muitas formas
- O polimorfismo é um dos pilares da programação orientada a objetos
- De uma forma simplificada podemos dizer que:
  - é uma forma de diferentes classes poderem ser utilizadas da mesma forma
  - ou seja, embora implementem as mesmas operações, estas são implementadas de formas diferentes
- Em C# o polimorfismo é implementado utilizando o mecanismo de *overriding*:
  - *property overriding*: uma classe derivada implementa a sua própria versão de uma propriedade definida pela classe base
  - *method overriding*: uma classe derivada implementa a sua própria versão de um método definido pela classe base

O ponto principal a reter é que o mecanismo de *overriding* é um **mecanismo de extensão de um membro herdado**

# Polimorfismo: Métodos virtuais

- Quando uma classe base define um método que pode vir a ser *overriden* (redefinido) por uma classe derivada, deverá marcar esse método com a *keyword* **virtual**:

```
public class A
{
    public virtual void X()
    {
        // Faz qualquer coisa...
    }
}
```

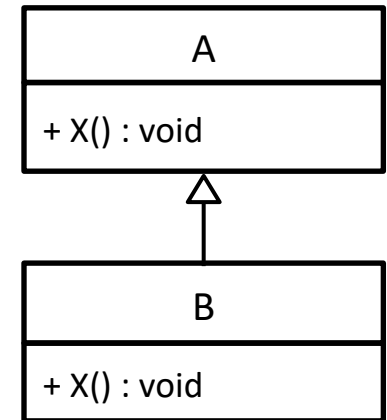


- Estes métodos são chamados **métodos virtuais** (*virtual methods*)

# Polimorfismo: Métodos virtuais

- Quando uma classe cria a sua própria implementação de um método, utiliza a *keyword* `override`:

```
public class B : A
{
    public override void X()
    {
        // Também faz qualquer coisa...
    }
}
```



# Polimorfismo: Métodos virtuais

- Se por qualquer motivo o novo método necessitar de aceder ao método da *base class*, pode fazê-lo utilizando a *keyword* `base`:

```
public class B : A
{
    public override void X()
    {
        // Também faz qualquer coisa...

        base.X();
    }
}
```

Executar o método `X()` da *base class*

```
public class A
{
    public virtual void X()
    {
        // Faz qualquer coisa...
    }
}
```

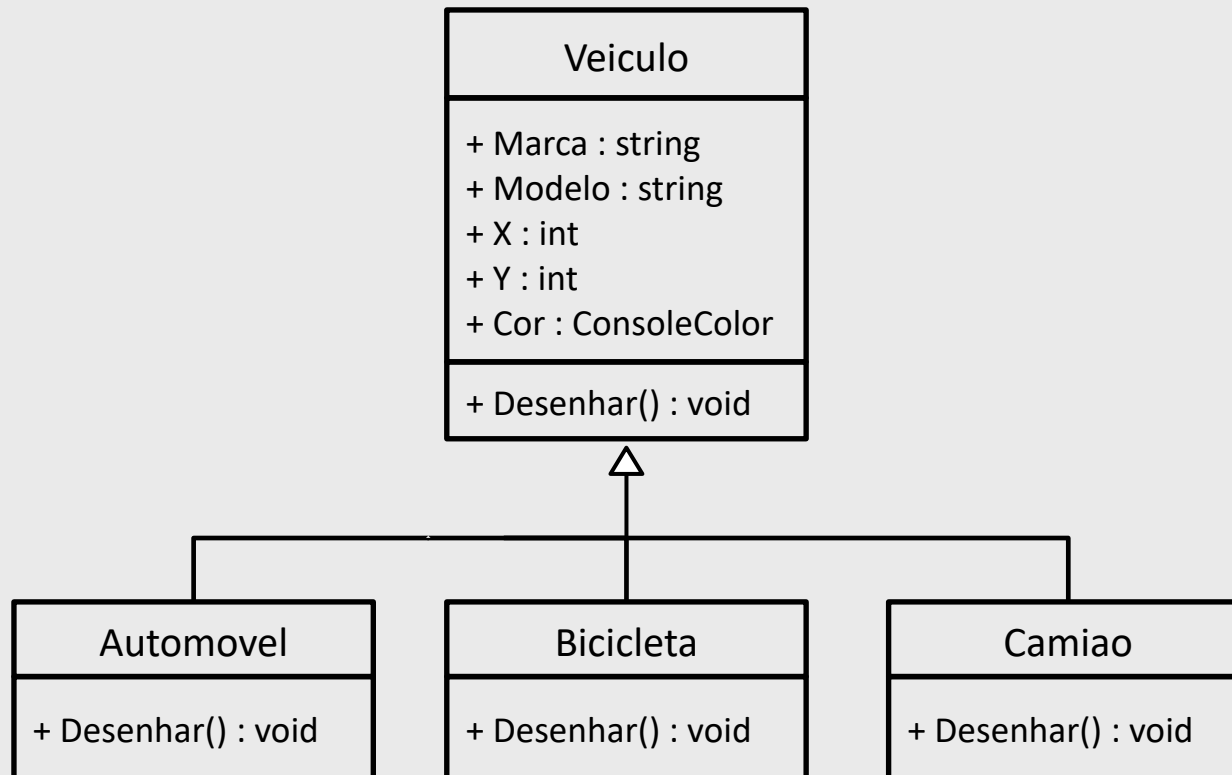


# Polimorfismo: Exemplo prático

- Vamos agora ver um exemplo de polimorfismo
- Faça download no Moodle, do ficheiro [PolimorfismoExemplo.zip](#)
- Neste projeto encontra:
  - a classe base [Veiculo](#)
  - as classes derivadas [Automovel](#), [Bicicleta](#) e [Camiao](#)

# Polimorfismo: Exemplo prático

- Na classe base é definido o método virtual `Desenhar()`
- As classes derivadas efetuam *override* do método `Desenhar()` criando as suas próprias versões do mesmo



# Polimorfismo: Exemplo prático

- No ficheiro `Program.cs`:
  - é declarada uma lista do tipo `Veiculo` e são armazenados nesta, instâncias das classes `Automovel`, `Camiao` e `Bicicleta`
  - tal só é possível porque todas são instâncias de classes derivadas da classe `Veiculo`

```
// Criar uma lista de objetos do tipo Veiculo
List<Veiculo> veiculos = new List<Veiculo>();

// Adicionar veículos de diferentes tipos à lista
veiculos.Add(new Automovel("Toyota", "Corolla", 2, 3, ConsoleColor.Yellow));
veiculos.Add(new Camiao("Toyta", "Hiace", 2, 8, ConsoleColor.Red));
veiculos.Add(new Bicicleta("Specialized", "S-Works Epic", 2, 13, ConsoleColor.Black));
veiculos.Add(new Automovel("Ford", "Focus", 18, 3, ConsoleColor.Blue));
veiculos.Add(new Camiao("Toyta", "Hiace", 18, 8, ConsoleColor.White));
veiculos.Add(new Bicicleta("DX3", "M14", 18, 13, ConsoleColor.Black));
```

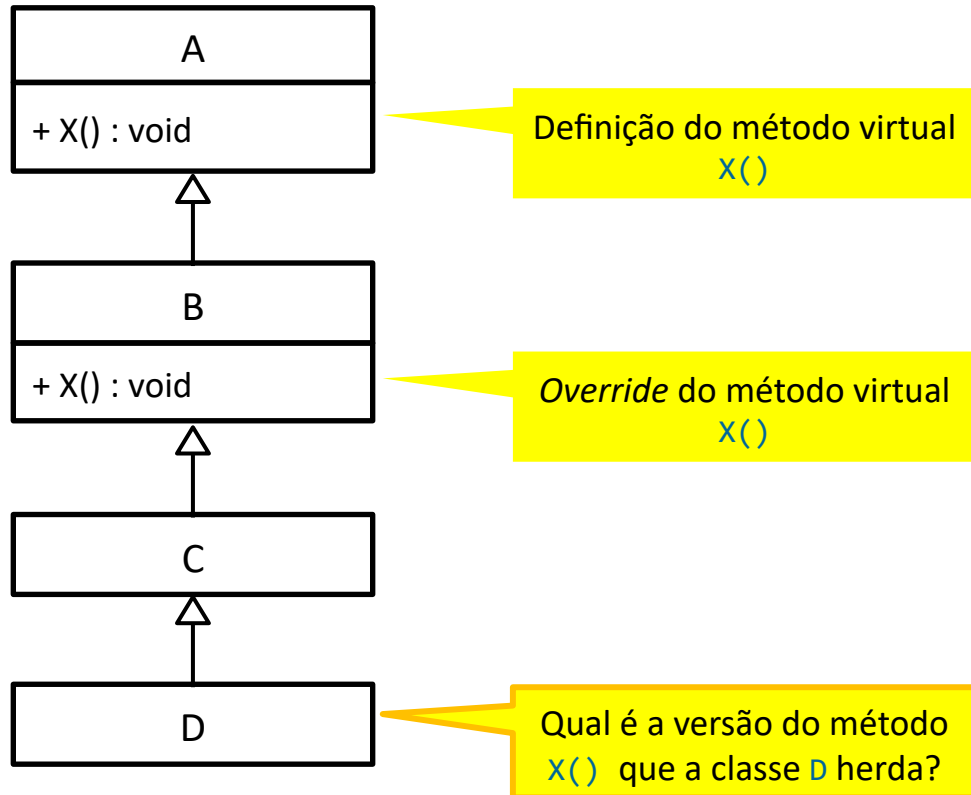
## Polimorfismo: Exemplo prático

- O poder do polimorfismo fica patente no seguinte fragmento de código
- Apesar de serem objetos diferentes, a lista de veículos é percorrida sem dificuldade e é chamado o método `Desenhar()` de cada objeto

```
// Percorrer a lista de veículos e desenhar cada um  
// (só é possível através do mecanismo de polimorfismo)  
foreach (Veiculo veiculo in veiculos)  
{  
    veiculo.Desenhar();  
}
```

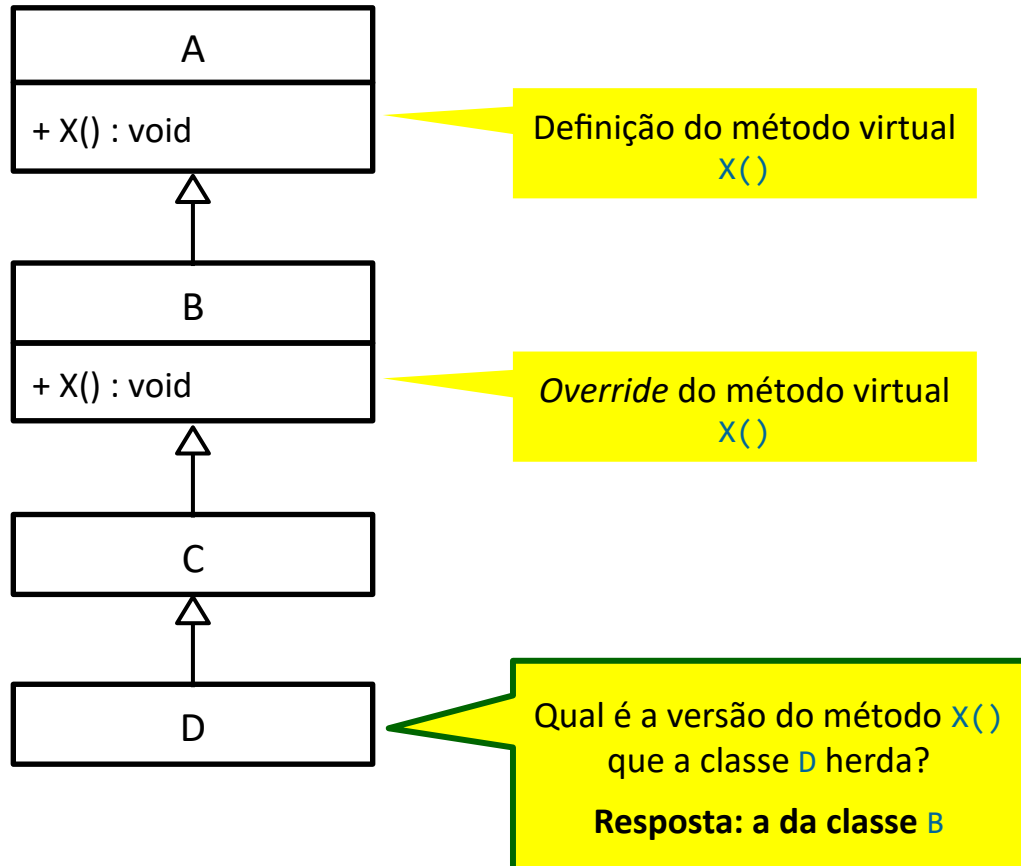
# Polimorfismo

- Os membros virtuais propagam-se através da hierarquia de classes:



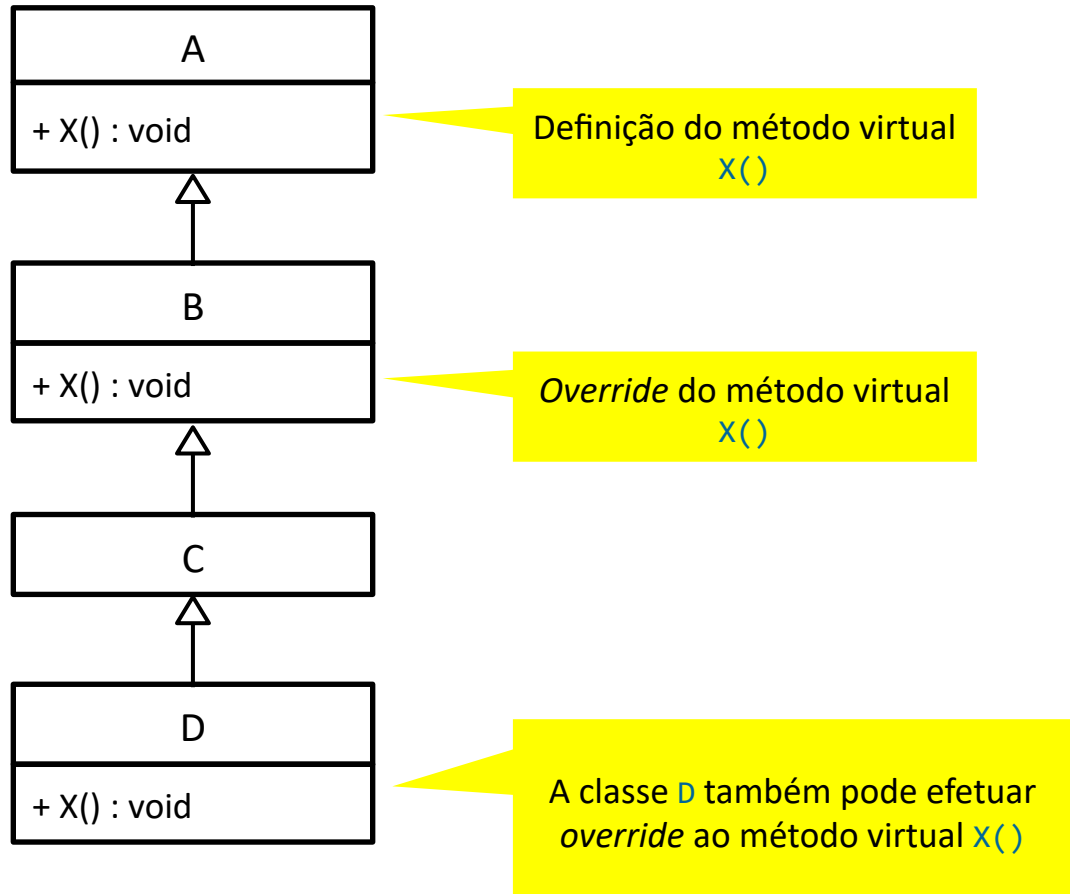
# Polimorfismo

- Os membros virtuais propagam-se através da hierarquia de classes:



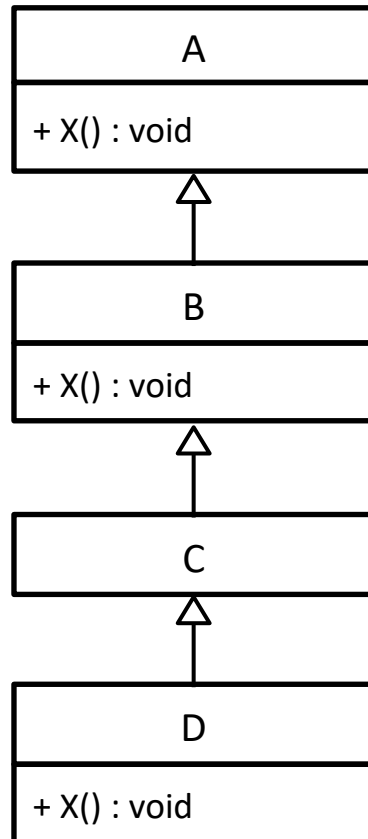
# Polimorfismo

- Os membros virtuais propagam-se através da hierarquia de classes:



# Polimorfismo

- Os membros virtuais propagam-se através da hierarquia de classes:



```
public class A
{
    public virtual void X()
    {
        Console.WriteLine("Método X() da classe A");
    }
}
```

```
public class B : A
{
    public override void X()
    {
        Console.WriteLine("Método X() da classe B");
    }
}
```

```
public class C : B
{ }
```

```
public class D : C
{
    public override void X()
    {
        Console.WriteLine("Método X() da classe D");
    }
}
```



# Polimorfismo: Impedir o *override*

- Podemos impedir o *override* de um membro virtual utilizando a *keyword* `sealed`
- Isto é, a partir de um certo momento, as classes derivadas deixam de poder efetuar *override* de uma propriedade ou de um método:

```
public class A
{
    public virtual void X()
    {
        // Faz qualquer coisa...
    }
}
```

```
public class B : A
{
    public sealed override void X()
    {
        // Também faz qualquer coisa...
    }
}
```

A partir daqui, as classes que herdam de B não podem fazer *override* do método `X()`

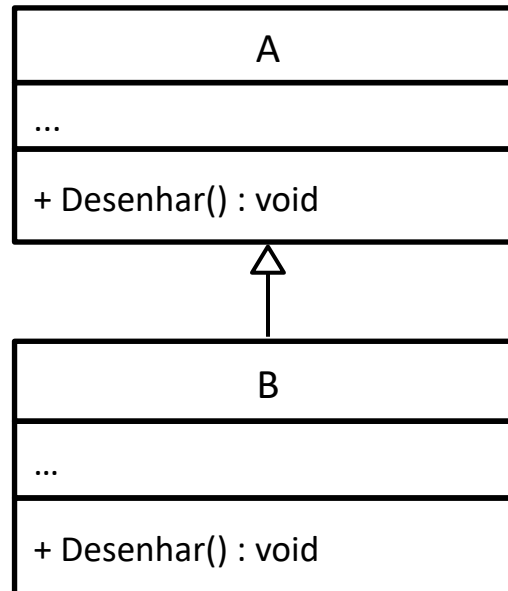
OCULTAR  
MEMBROS  
HERDADOS

# Ocultar membros herdados

- Um membro virtual quando é redefinido através de *override* é extendido: ou seja, procura-se aproveitar o membro definido inicialmente pela *base class* em conjunto com uma nova versão, definida pela classe derivada
- Por vezes existe a necessidade inversa: definir uma nova versão, radicalmente diferente, e esconder a versão da *base class*
- Para tal utiliza-se a keyword `new`

# Acesso aos membros herdados: Ocultar membros

- Imaginemos o seguinte cenário:
  - a classe **B** herda da classe **A** o método **Desenhar()**
  - no entanto a forma de desenhar a classe **B** é diferente
- Neste caso a classe **B** irá criar a sua própria versão do método **Desenhar()**



# Acesso aos membros herdados: Ocultar membros

- Neste caso, diz-se que o método `Desenhar()` de `B` oculta o método `Desenhar()` de `A`
- Devemos utilizar a *keyword* `new` para indicar explicitamente ao compilador que se trata de uma nova versão do método

```
public class A
{
    public void Desenhar()
    {
        // Desenhar o objeto
    }
}

public class B : A
{
    public new void Desenhar()
    {
        // Desenhar o objeto
    }
}
```

# Acesso aos membros herdados: Ocultar membros

- Para aceder aos membros que estão ocultos, na *base class*, utiliza-se uma *base access expression*
- Consiste em `base.` seguido do nome do membro que se pretende aceder
- Por exemplo, se na classe `B` se pretendesse aceder ao método `Desenhar()` da classe `A`:

```
public class B : A
{
    new public void Desenhar()
    {
        // Desenhar o objeto
    }

    public void MetodoExemplo()
    {
        Desenhar();

        base.Desenhar();
    }
}
```

Chamar o método `Desenhar()` da classe `B`

Chamar o método `Desenhar()` da classe `A`

# CLASSES ABSTRACTAS

## ***Abstract class***

- Uma *abstract class* é uma classe que não pode ser instanciada
- Ou seja, é feita propositadamente para servir como *base class*
- Uma *abstract class* é definida utilizando a keyword **abstract**

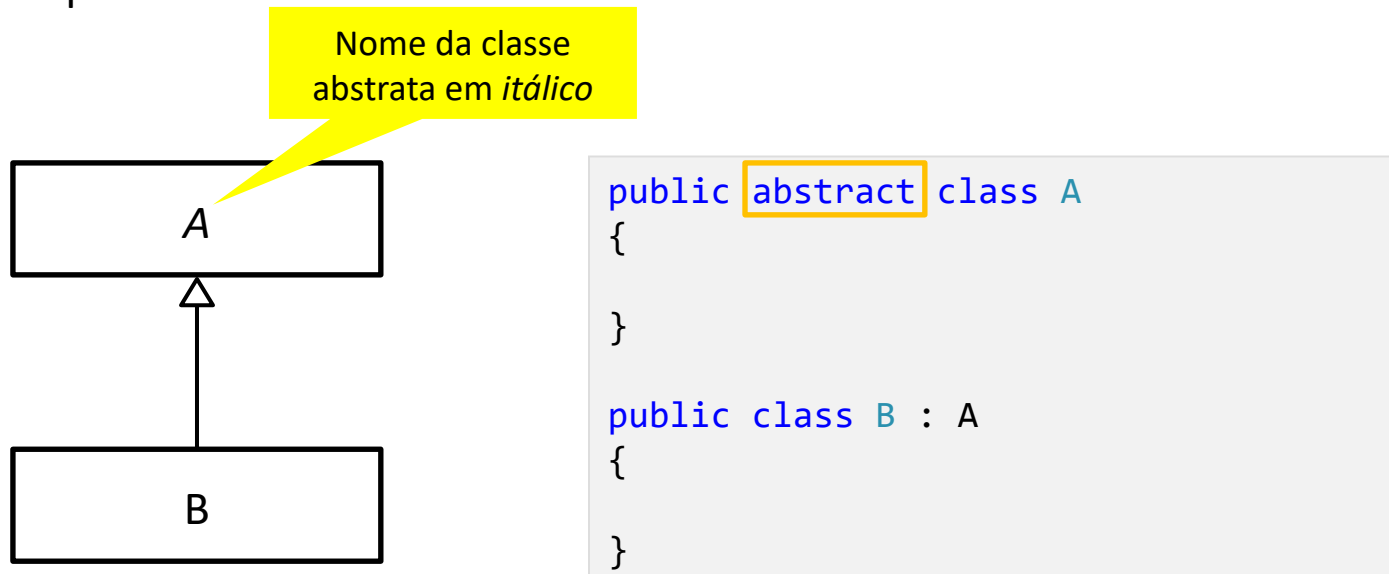


# Abstract class

- Sintaxe:

```
[access_modifier] abstract class nome  
{  
    ...  
}
```

- Exemplo:



## Abstract class

- Um exemplo prático consiste na classe `Veiculo`
- Se pensarmos bem, o conceito de **veículo** é um conceito **abstrato**
- Apenas um tipo de veículo concreto (como uma trotineta, um automóvel, etc.) é que não é uma ideia abstrata
- Assim sendo, podemos definir a classe `Veiculo` como sendo abstrata:

```
public abstract class Veiculo
{
    public string Marca { get; set; }
    public string Modelo { get; set; }
}

public class Trotineta : Veiculo
{
    public string Cor { get; set; }
}
```

# Membros abstratos

- Uma classe abstrata pode definir membros abstratos
- Ou seja, membros que são criados para serem implementados de forma concreta nas classes derivadas
- **Porquê declarar membros abstratos?**
- Porque são membros cuja implementação concreta depende da classe onde são utilizados
- Ou seja, não vale a pena implementá-los na classe abstrata porque só faz sentido fazê-lo nas classes herdeiras
- Ao definirmos um membro como abstrato estamos a dizer que qualquer classe derivada deve obrigatoriamente conter esse membro e terá de implementá-lo

# Membros abstratos

- Regras:
  - membros abstratos apenas podem ser declarados em classes abstratas
  - são declarados utilizando a keyword `abstract`
  - não podem conter código (utilizar apenas `;`)
- Apenas podem ser declarados como abstratos os seguintes tipos de membros:
  - propriedades (campos e constantes não podem ser abstratos)
  - métodos
  - eventos
  - *indexers*
- Os membros abstratos são definidos com a *keyword* `abstract`
- Nas classes derivadas, os membros que implementam os membros abstratos devem ser definidos com a *keyword* `override`

# Membros abstratos

- Exemplo:

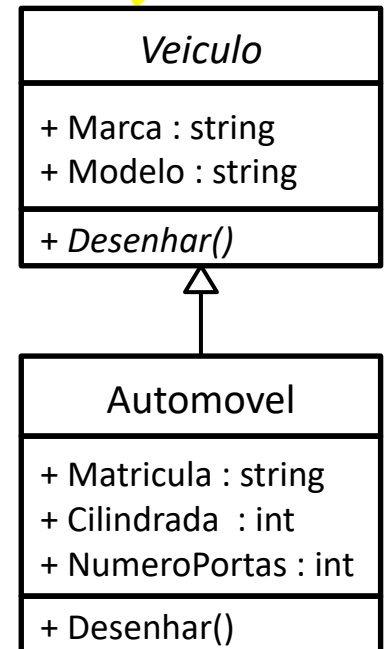
```
public abstract class Veiculo
{
    public string Marca { get; set; }
    public string Modelo { get; set; }

    public abstract void Desenhar();
}
```

```
public class Automovel : Veiculo
{
    public string Matricula { get; set; }
    public int Cilindrada { get; set; }
    public int NumeroPortas { get; set; }

    public override void Desenhar()
    {
        ...
    }
}
```

Membros abstratos deverão  
estar em *itálico*



Implementar o método `Desenhar()`  
na classe `Automovel`

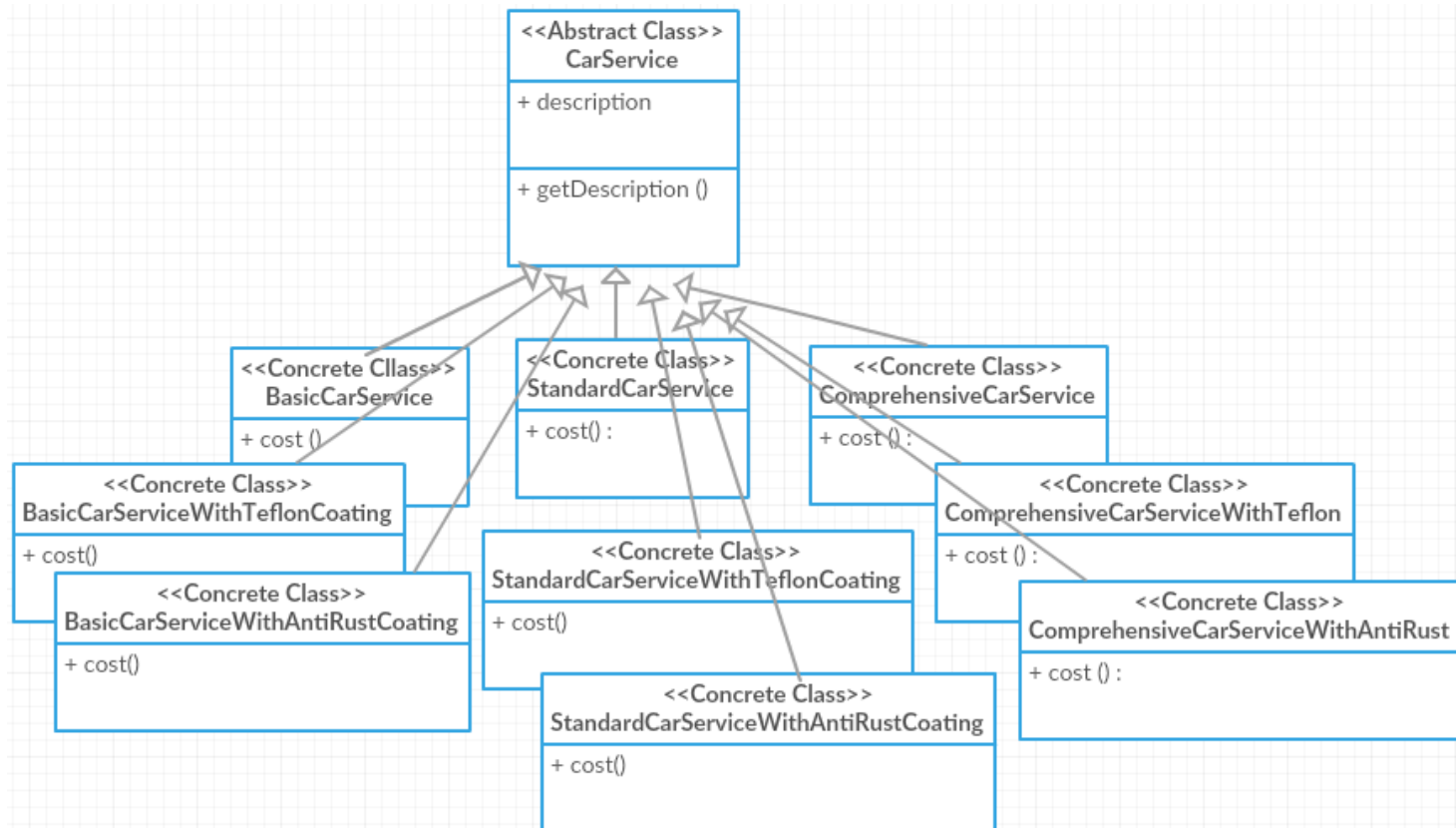
# COMPOSIÇÃO E AGREGAÇÃO

# Introdução

- Por muito lógico e conveniente que o mecanismo de herança possa parecer, à primeira vista, este possui, no entanto, desvantagens
- Facilmente se criam hierarquias longas e complexas: ***class explosion problem***
- O que torna o código:
  - complexo
  - difícil de alterar
  - difícil de manter

# Introdução

- Exemplo: classes para o sistema de uma empresa de lavagem e limpeza de automóveis



<https://medium.com/@vinodkumardwivedi/decorator-design-pattern-4e80f915a0f3>



# Introdução

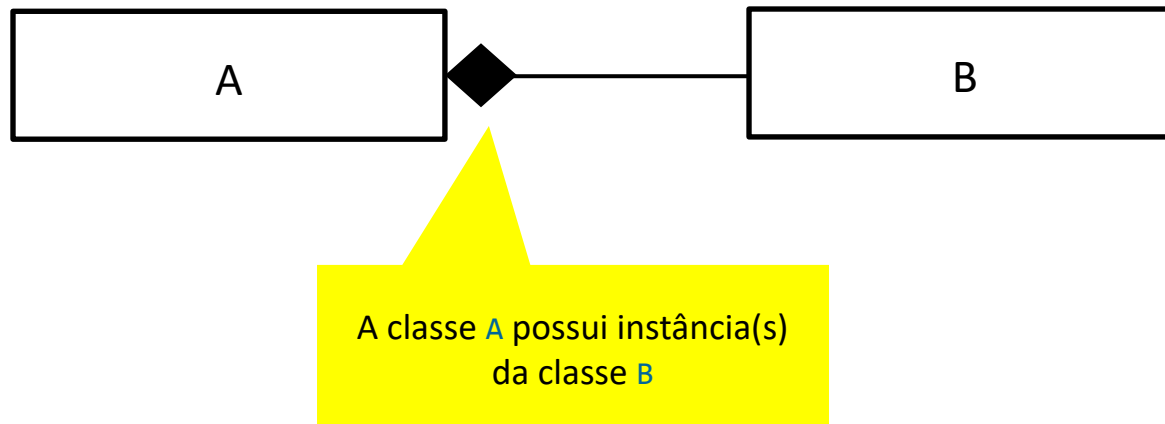
- Atualmente é recomendada a utilização de associação de classes em vez da herança
- Existem dois tipos de associação:
  - composição
  - agregação
- **A composição e a agregação permitem combinar classes, formando classes mais complexas**
- É importante notar que a herança deve continuar a ser utilizada
- Mas apenas nos casos em que tal se justifica realmente

# Introdução

- Na herança, existe um relacionamento "*is-a*" (**é**):
  - um automóvel **é** um veículo
  - um triângulo **é** uma forma
  - um quadrado **é** um retângulo
  - etc.
- Na composição/agregação existe um relacionamento "*has-a*" (**tem**, ou possui):
  - um automóvel **tem** um motor
  - um aluno **tem** várias disciplinas
  - uma pessoa **tem** contas bancárias
  - etc.

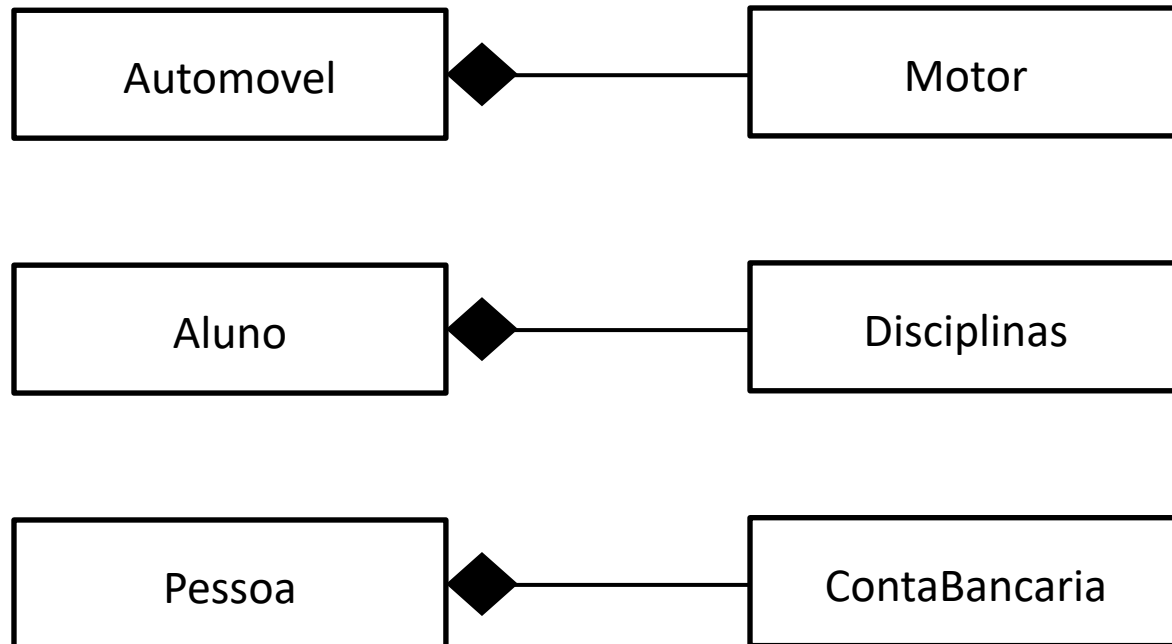
# Composição

- Permite combinar classes, formando classes mais complexas
- Uma instância de uma classe possui uma ou várias instâncias de outras classes
- Representação em diagrama de classes:



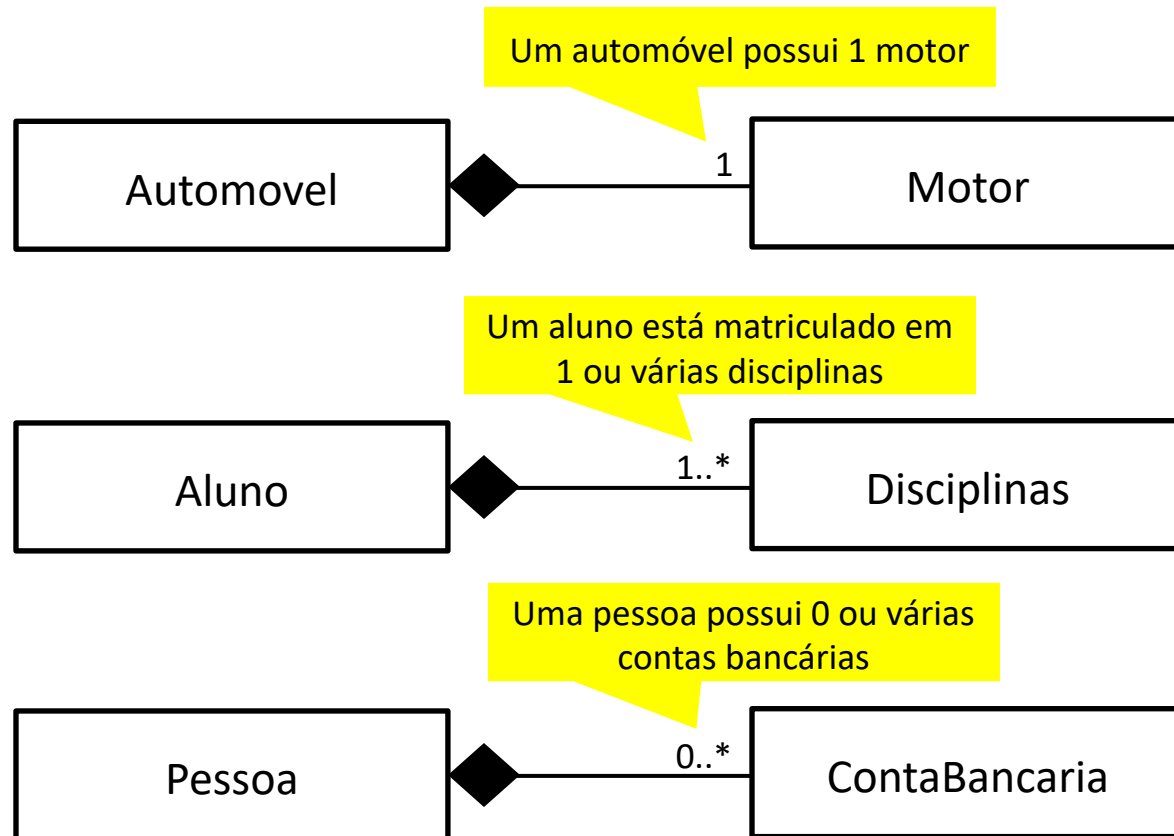
# Composição

- Exemplo:



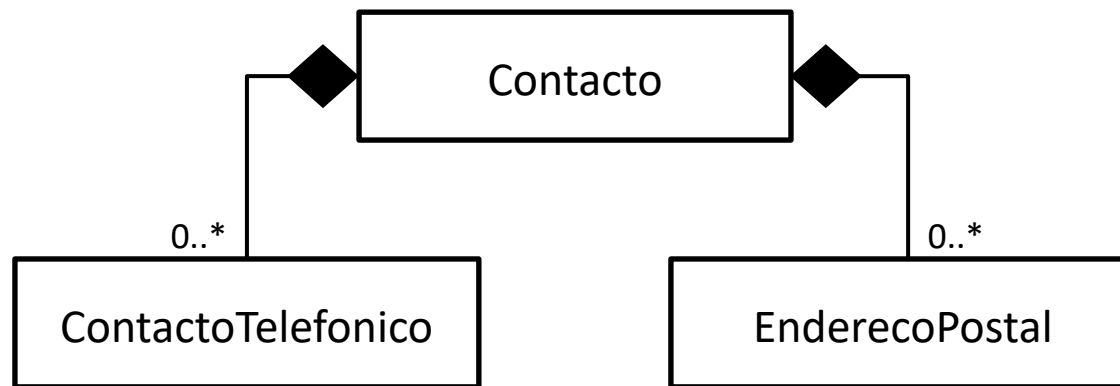
# Composição

- Indica-se no diagrama de classes a multiplicidade da relação
- Ou seja, o número mínimo e máximo de instâncias que uma classe possui



# Composição

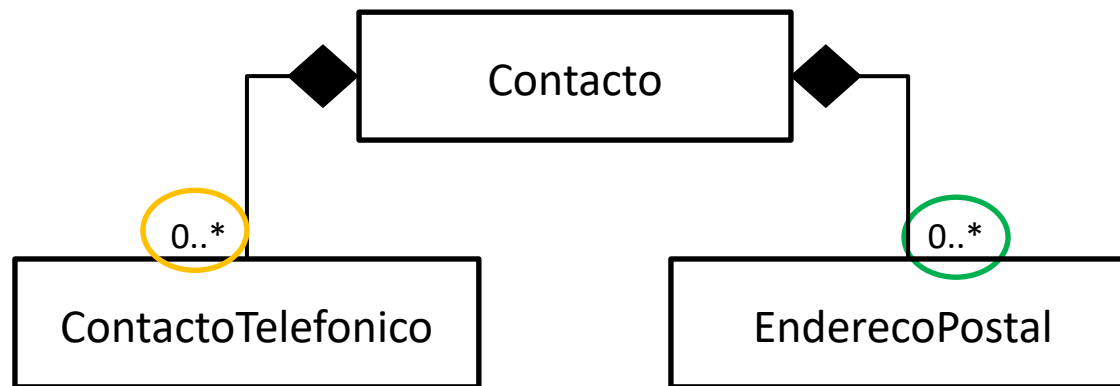
- Imaginemos como exemplo, uma aplicação para gestão contactos num telemóvel:
  - cada contacto pode conter zero ou mais números de telefone
  - cada contacto pode conter zero ou mais endereços



- um contacto telefónico pertence apenas a um contacto
- um endereço postal pertence apenas a um contacto

# Composição

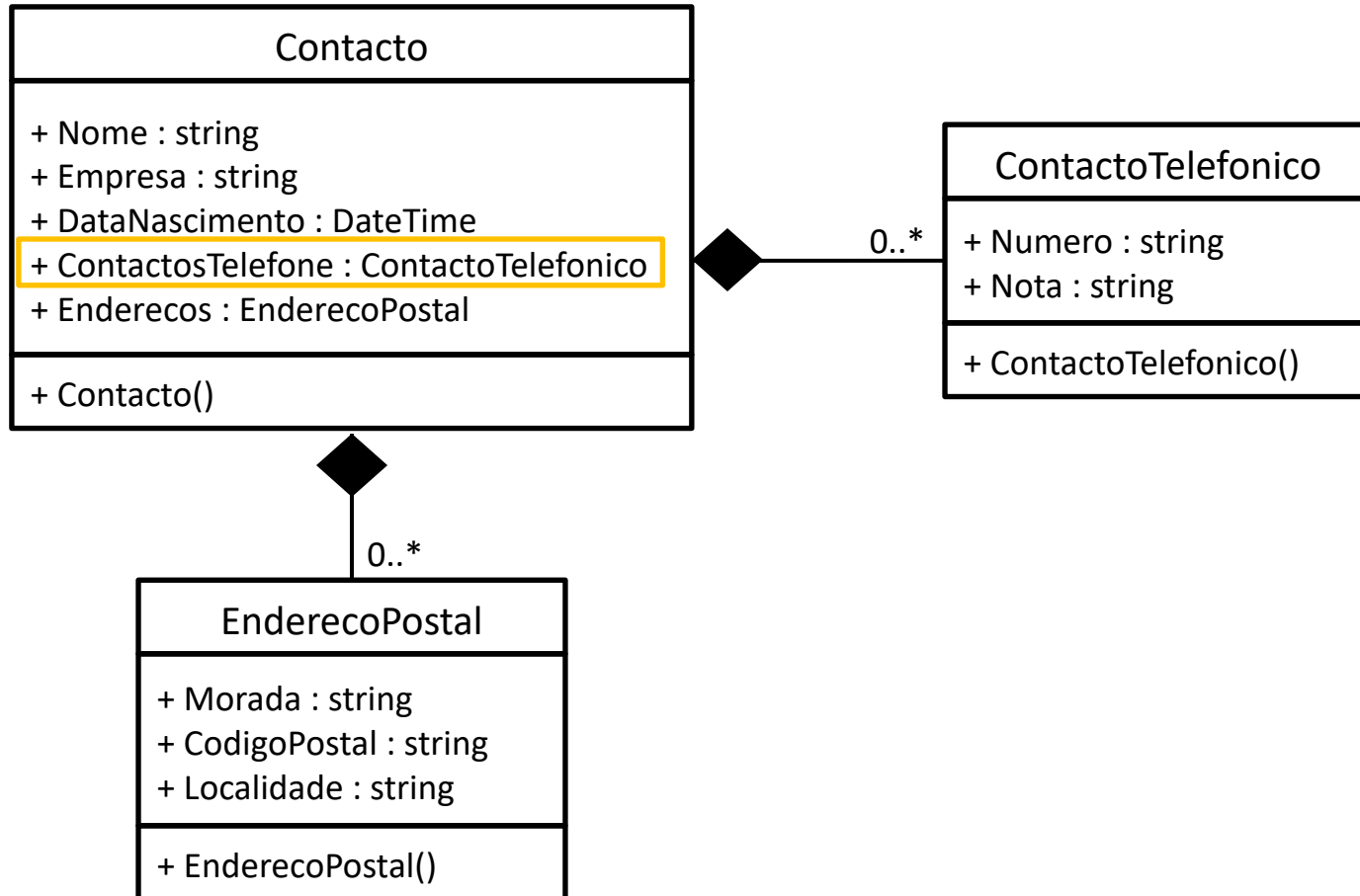
- Imaginemos como exemplo, uma aplicação para gestão contactos num telemóvel:
  - cada contacto pode conter zero ou mais números de telefone
  - cada contacto pode conter zero ou mais endereços



- um contacto telefónico pertence apenas a um contacto
- um endereço postal pertence apenas a um contacto

# Composição

- Diagrama de classes detalhado:





# Composição

- O código para a classe `Contacto`:

```
class Contacto
{
    public string Nome { get; set; }
    public string Empresa { get; set; }
    public DateTime DataNascimento { get; set; }

    public List<ContactoTelefonico> ContactosTelefone { get; set; } = new
List<ContactoTelefonico>();
    public List<EnderecoPostal> Enderecos { get; set; } = new
List<EnderecoPostal>();

    public Contacto(string nome, string empresa, DateTime dataNascimento)
    {
        Nome = nome;
        Empresa = empresa;
        DataNascimento = dataNascimento;
    }
}
```

# Composição

- O código para a classe `ContactoTelefonico`:

```
class ContactoTelefonico
{
    public string Numero;
    public string Nota;

    public ContactoTelefonico(string numero, string nota)
    {
        Numero = numero;
        Nota = nota;
    }
}
```

# Composição

- O código para a classe `EnderecoPostal`:

```
class EnderecoPostal
{
    public string Morada { get; set; }
    public string CodigoPostal { get; set; }
    public string Localidade { get; set; }

    public EnderecoPostal(string morada, string codigoPostal, string
localidade)
    {
        Morada = morada;
        CodigoPostal = codigoPostal;
        Localidade = localidade;
    }
}
```

# Composição

- No seguinte código:
  - é criada uma nova instância de `Contacto` chamada `contacto1`
  - dentro de `contacto1` são criados dois contactos telefónicos (duas instâncias de `ContactoTelefonico`)
  - dentro de `contacto1` é criado um endereço postal (uma instância de `EnderecoPostal`)

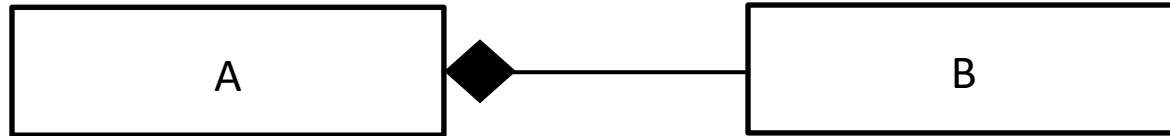
```
// Criar instância de Contacto
Contacto contacto1 = new Contacto("Miguel Santos", "Transportes Veloz",
new DateTime(1986, 10, 14));

// Criar dois contactos telefónicos
contacto1.ContactosTelefone.Add(new ContactoTelefonico("239123456",
"Empresa"));
contacto1.ContactosTelefone.Add(new ContactoTelefonico("911122334",
"Pessoal"));

// Criar um endereço para o contacto
contacto1.Enderecos.Add(new EnderecoPostal("Rua Xyz", "3000-001",
"Coimbra"));
```

# Composição

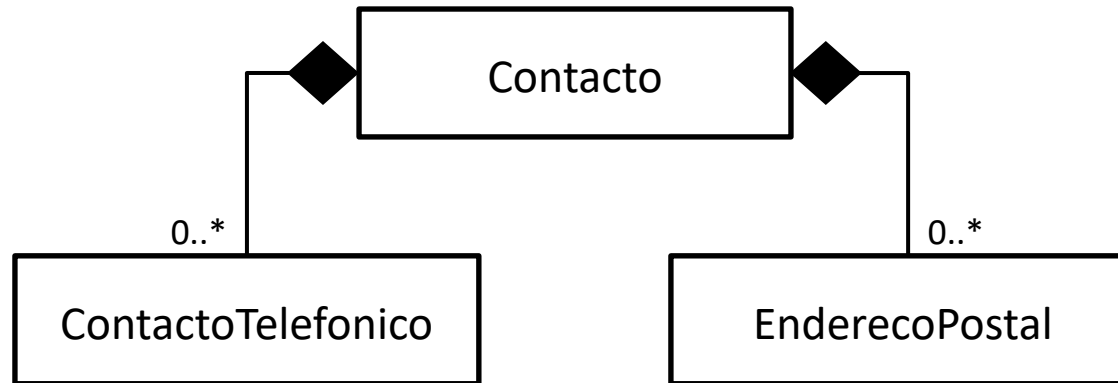
- Na composição, o relacionamento entre as classes é um relacionamento **forte**
- Ou seja, as instâncias que estão dentro de uma classe apenas existem dentro dessa classe



- Quando uma instância da classe **A** deixa de existir, todas as instâncias da classe **B**, que **A** possui, deixam de existir

# Composição

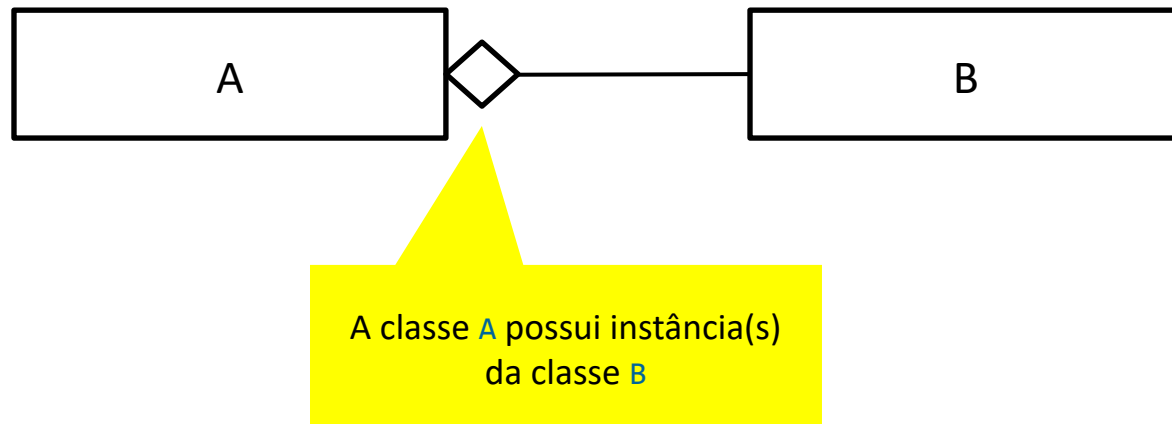
- No exemplo analisado anteriormente:



- Quando a instância **contacto1** deixa de existir, automaticamente deixam de existir também os 2 contactos telefónicos e o endereço postal

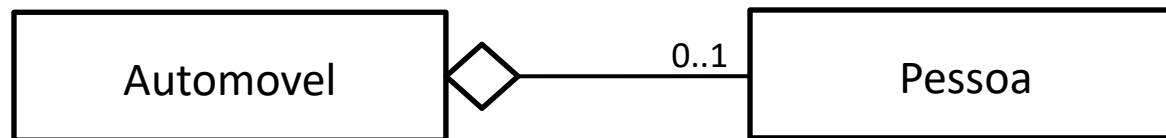
# Agregação

- Na agregação, a relação entre os objetos é uma relação **fraca**
- Ou seja, os objetos existem de forma independente
- Ao contrário da composição, em que alguns objetos estão subordinados a outros



# Agregação

- Imaginemos o seguinte exemplo:
  - uma empresa possui uma frota de automóveis
  - a cada automóvel é atribuído um condutor (uma instância da classe **Pessoa**)

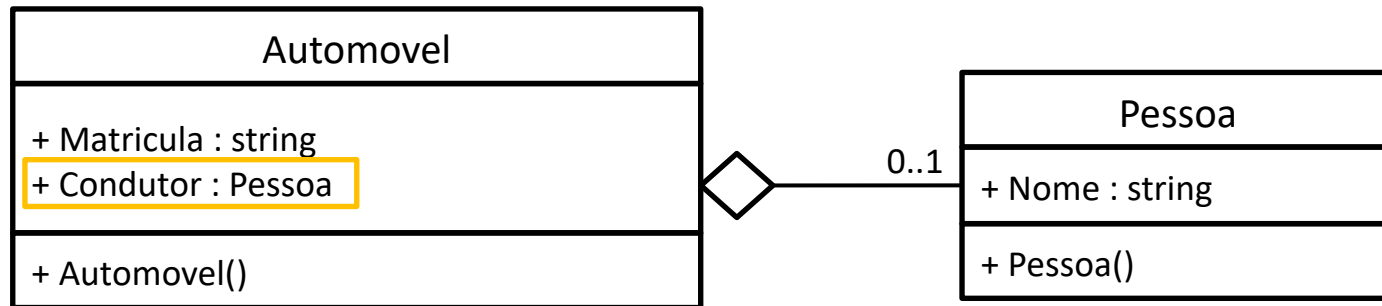


- A agregação difere da composição no sentido em que:
  - se um automóvel deixar de existir, o condutor (classe **Pessoa**) continua a existir



# Agregação

- Exemplo:



# Agregação

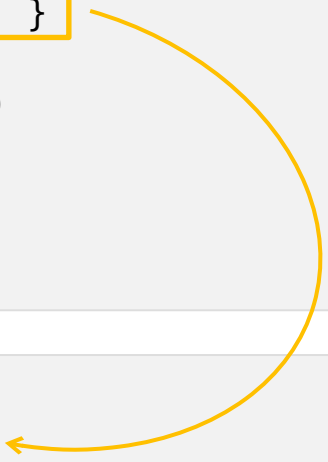
- O código para as classes `Automovel` e `Pessoa`:

```
class Automovel
{
    public string Matricula { get; set; }
    public Pessoa Condutor { get; set; }

    public Automovel(string matricula)
    {
        Matricula = matricula;
    }
}
```

```
class Pessoa
{
    public string Nome { get; set; }

    public Pessoa(string nome)
    {
        Nome = nome;
    }
}
```



# Agregação

- No seguinte código:
  - é criada uma nova instância de `Automovel` chamada `automovel1`
  - é criada uma nova instância de `Pessoa` chamada `peessoa1`
  - é feita a associação entre as duas classes, através da propriedade `Condutor`

```
// Criar um novo automóvel
Automovel automovel1 = new Automovel("99-HU-11");
// Criar uma nova pessoa
Pessoa peessoa1 = new Pessoa("Cláudia Martins");
```

```
// Atribuir um condutor a um automóvel
automovel1.Condutor = peessoa1;
```

Associar a classe `Pessoa` à  
classe `Automovel`

```
// Aceder ao nome da pessoa, através da classe Automovel
Console.WriteLine("O condutor do automóvel {0} é {1}",
    automovel1.Matricula, automovel1.Condutor.Nome);
```