Upgrading Skills

Primavera ACADEMY

Módulo 1

Programação Orientada a Objetos

Programming C# - Advanced

Plano de formação detalhado

1. Programação Orientada a Objetos
- *Classes e classes abstratas*
- *Classes e métodos estáticos*
- *Classes e métodos parciais*
- *Construtores de instância e estáticos*
- *Herança*
- *Encapsulamento*
- *Polimorfismo*
- *Interfaces*

2. Coleções
- Genéricos
- Interfaces IEnumerable, IEnumerator, ICollection, IList
- Arrays
- Listas, Filas e Pilhas
- Dicionários

3. Princípios SOLID e código limpo
- Single Responsibility Principle
- Open Close Principle
- Liskov Principle
- Interface Segregation Principle
- Dependency Inversion Principle (IoC Containers)
- Código limpo e Organização de Código
- Comentários

4. Introdução ao desenvolvimento orientado a testes
- Introdução aos testes unitários
- Padrão de escrita de testes AAA (Arrange, Act, Assert)
- Conceitos de desenvolvimento orientado a testes
- Validação de testes com xUnit

5. Language Integrated Query
- Sintaxe
- Operadores
- Expressões de consulta
- Execução diferida
- Subconsultas
- Estratégias de composição e projeção

6. Conteúdos avançados
- Delegates
- Events
- Expressões Lambda
- Tratamento de excepções
- Extension Methods
- Tipos nulos
- Programação assíncrona

7. *Funcionalidades recentes*
- Tuples
- Pattern matching
- Funções anónimas
- Funções locais
- Discards

Conceitos gerais

- A evolução tecnológica forçou o aparecimento do paradigma da Programação Orientada a Objetos (POO).
- Por um lado, havia a necessidade de diminuir a complexidade das aplicações, por outro, a produtividade dos programadores teria de aumentar exponencialmente.
- Assim, a POO veio ajudar a simplificar projetos, quer pelas suas técnicas e conceitos de programação, quer pela reutilização de código.
- Antes da POO, um programa era visto como um conjunto de instruções para desempenhar uma tarefa específica.
- Depois da POO (melhor! com a POO), um programa é um conjunto de instruções e objetos que interagem entre si, onde cada elemento preserva a sua individualidade.

Conceitos de POO (1/4)

- A POO permite-nos modelar o mundo real através de código.
- Ao contrário da programação procedimental, em que tudo é visto como um fluxograma, podemos criar objetos com atributos e associar-lhes comportamentos.
- Um carro, por exemplo, apresenta caracteristicas (**atributos**) tais como: cor, comprimento, altura, kgs e exibe diversos comportamentos (**métodos**) tais como: acelerar, travar, virar à esquerda etc.

Conceitos de POO (2/4)

Abstração

- A abstração significa usar coisas simples para reduzir a complexidade. Não necessitamos de entender como um candeeiro funciona para interagir com ele.
- Através de objetos, classes e atributos conseguimos representar código complexo que pode ser reutilizado múltiplas vezes.

Encapsulamento

- Define o conceito de esconder o acesso às variáveis e lógica interna de um objeto, prevenindo que sejam acedidas inadvertidamente por componentes não autorizados.
- Os membros da classe internos devem ser marcados como privados, enquanto os que desejamos expôr para fora, são públicos.

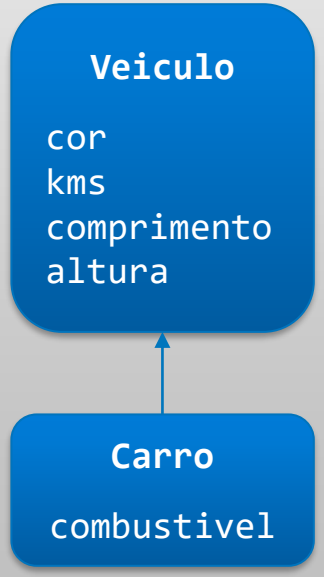
Conceitos de POO (3/4)

Herança / Especialização

- Através da herança, podemos criar novas classes, reaproveitando a lógica e atributos de outras classes.
- Um carro é um veículo. Isto implica que um carro pode herdar os seus atributos, não necessitando de os definir na sua classe.

- Como há veículos que podem não usar qualquer tipo de combustível, o atributo é apenas definido na classe especializada.

E se quisermos representar um carro elétrico, uma moto de água ou um avião?



Conceitos de POO (4/4)

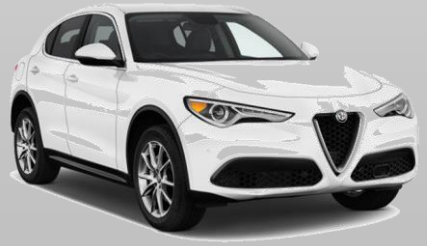
Polimorfismo

- O polimorfismo está intimamente ligado à liberdade que um objeto tem de definir o seu comportamento como quiser.
- Um avião e um carro são ambos veículos mas implementam a operação de acelerar de forma completamente diferente.
- O polimorfismo pode ser implementado de 2 formas:
    - **`Overriding` de métodos** - quando diferentes objetos que herdam de uma classe base, reimplementam os métodos de forma diferente.
    - **`Overloading` de métodos** – Quando o mesmo objeto define vários métodos com o mesmo nome para fazer uma operação semelhante (sendo a assinatura dos métodos diferente)

Objetos

- Um objeto pode ser uma abstracção de algo visível que existe no mundo real mas também a representação de algo que não seja tangível, e.g. um objeto que envie mensagens para outros sistemas.
- Um objeto é composto por atributos i.e. as características que o definem mas também por métodos (o seu comportamento)
- Atributos são as características do objeto. Um carro pode ter cor, comprimento, altura, etc.
- Os métodos são as operações que um determinado objeto pode realizar. Um carro pode avançar, parar, virar à direita e à esquerda etc.

```
Objeto = Atributos + métodos
```

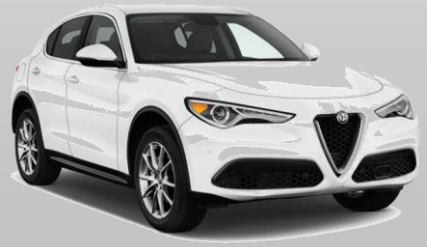


Classes (1)

- Uma classe pode ser comparada a um molde ou forma que possibilita a criação de vários objetos, com diferentes valores para as suas propriedades.
- Por exemplo, podemos ter uma classe que permite criar objetos do tipo Carro, Animal, Casa. Esta abstracção permite-nos visualizar e melhor entender como a informação está relacionada e como os diferentes objetos se inter-relacionam.
- As classes podem ser organizadas como quisermos mas, classes bem definidas são desenhadas de modo a representar conceitos.

Classes (2)

- As classes são a estrutura dos objetos, ou seja, representam as suas especificações.
- As classes são vistas como o molde para a criação de objetos.
- `class` é a palavra reservada usada para criar uma classe.
- Os atributos são definidos dentro da sua classe.

Classe carro	
Atributos (características): - Cor - Pneu - …	
Métodos (funções): - Conduzir - Travar	

Classes (3)

Sintaxe de Classes

```
<acesso> class <nome da classe>
{
   // membros da classe
}
```

```
public class Carro
{
   // membros da classe
}
```

Modificadores de acesso	Descrição
internal	• A classe só pode ser acedida a partir do assembly onde se encontra definida. • Nível de proteção definido por omissão.
public	• A classe pode ser acedida a partir de qualquer assembly.

Assembly *representa um ficheiro ou arquivo de código compilado. Existem 2 tipos de assemblies: biblioteca (.dll) ou executável (.exe).*

Classes (4)

Sintaxe de Atributos

[acesso] <tipo dados> <nome do atributo>

```
public class Carro
{
   // Atributos
    int numKms;
    private string matricula;
    private Cor cor;
}
```

Classes (5)

Sintaxe de Métodos

[acesso] <tipo retorno> <nome>([<lista parâmetros>])

```
public class Carro
{
   private int velocidade;

   private void Acelerar(int kmh)
   {
      velocidade += kmh;
   }

   private void Desacelerar(int kmh)
   {
      velocidade -= kmh;
   }
}
```

Modificadores de Acesso (1)

- O modificador de acesso define o nível de proteção para atributos, métodos e classes, representando o tipo de acesso a outras classes/métodos.
- Os *namespaces* não têm modificador de acesso, são considerados sempre públicos.

Modificadores de Acesso (2)

Modificadores de acesso	Descrição
`private`	• Só pode ser acedido a partir da classe onde foi declarado. • Nível de proteção definido, por omissão, no caso dos membros de uma classe (atributos, métodos, construtor, etc.)
`public`	• Pode ser acedido a partir de qualquer classe (inclusive fora do *assembly* onde foi definido).
`protected`	• Só pode ser acedido a partir da classe onde foi declarado e a partir de classes derivadas (inclusive fora do *assembly* onde foi definido).
`internal`	• Pode ser acedido a partir de qualquer classe dentro do *assembly* (excetuando fora do *assembly* onde foi definido).
`protected internal`	• Só pode ser acedido a partir da classe onde foi declarado e a partir de classes derivadas (excetuando fora do *assembly* onde foi definido).
`private protected`	• O acesso limitado à classe que o contém ou a classes derivadas dentro do mesmo *Assembly* (disponível a partir da versão de C# 7.2).

Métodos (1)

- Permitem introduzir comportamento. Podem receber múltiplos parâmetros de entrada e no máximo um tipo de retorno.
- Quando não devolve nada, marca-se o método com **`void`**.

Overloading de métodos

- Quando há multiplos métodos com o mesmo nome mas a assinatura do método é diferente.
- A assinatura de um método refere-se à combinação do nome do método e o tipo, ordem e número de parâmetros

O código seguinte é válido?

```
public class Macaco1
{
    public void SaltarArvore() { }
    public void SaltarArvore(string tipoArvore){}
    public void SaltarArvre(int numeroArvores, string tipoArvore){}
    public void SaltarArvre(string tipoArvore, int numeroArvores) {}
}
```

Métodos (2)

Overloading de métodos. Exemplos

```
public class Macaco
{
    // Métodos válidos.
    // Mesmo nome mas parâmetros diferentes
    public void SaltarArvore() { }
    public void SaltarArvore(string tipoArvore){}
    public void SaltarArvre(int numeroArvores){}
}
```

```
public class Macaco
{
    public int SaltarArvore() { }
    public string SaltarArvore(string tipoArvore) { }

    // Método não permitido.
    // O número de parâmetros, o tipo e a ordem são iguais
    public int SaltarArvore(string nomeArvore) { }
}
```

Métodos (3)

Métodos Expression Bodied (C# 6)

- Permitem a criação de métodos de forma compacta
- Têm uma única expressão
- O tipo de dados de retorno pode ser **`void`**.

Os métodos **`Soma`** e **`Somar`** são idênticos

```
public class Calculadora
{
    int Soma(int x, int y)
    {
        return x + y;
    }

    int Somar(int x, int y) => x + y;
}
```

Construtor de Instância (1)

- Permite a criação de objetos do tipo da classe
- Permite correr código de inicialização
- Chamado quando se usa o operador **`new`**
- Pode haver múltiplos construtores (com assinaturas diferentes)
- Método especial que tem o nome e o tipo de retorno da classe
- Uma classe tem sempre um construtor, mesmo quando não é criado explicitamente.

Construtor por omissão

```
public class Computador
{
    private int numTeclas;
}

// Usa o construtor por omissão, criado implicitamente
var computador = new Computador();
```

Construtor de instância (2)

- O construtor sem parâmetros é criado automaticamente apenas quando não há nenhum outro construtor definido.

```
public class Elefante
{
    private string _cor;

    public Elefante(string cor)
    {
        _cor = cor;
    }
}

// Válido.
var elefante = new Elefante("branco");

// Inválido.
// Quando há pelo menos um construtor criado explicitamente,
// então o construtor sem parâmetros já não é criado de forma implícita.
var elefante = new Elefante();
```

Construtor de instância (3)

Overloading de construtores

- Tal como nos métodos, podemos ter overloading de construtores, desde que a assinatura seja diferente
- Podemos chamar construtores a partir de outros construtores usando a palavra reservada **`this`**.

```
public class Elefante
{
    private string _nome;
    private int _idade;
    private string _cor;

    public Elefante(string cor)
    {
        _cor = cor;
    }

    public Elefante(int idade, string nome, string cor) : this(cor)
    {
        _idade = idade;
        _nome = nome;
    }

}


var elefante = new Elefante(10, "Brutamontes", "branco");
```

Construtores de instância (4)

Operador `this`

- **`this`** refere-se à própria instância do objeto.
- Além de permitir chamar outros construtores da classe, serve também para referenciar os membros da classe.
- A referência **`this`** permite desambiguar uma variável local ou parâmetro, por exemplo:

```
public class Jacare
{
    private string nome;
    private int idade;

    public Jacare(string nome)
    {
        this.nome = nome;
    }

    public Jacare(int idade, string nome) : this(nome) => this.idade = idade;
}
```

O que o 2.º construtor tem de especial?

Construtores de instância (5)

Construtores privados

- Não é necessário ter construtores públicos caso tenhamos uma forma de criar os objetos.

No exemplo, ambos os construtores são privados e por isso, o método `CriarJacare` tem a responsabilide de criar o objeto.

```
public class Jacare
{
    private string nome;
    private int idade;

    private Jacare(string nome)
    {
        this.nome = nome;
    }

    private Jacare(int idade, string nome) : this(nome) => this.idade = idade;

    public static Jacare CriarJacare(string nome, int idade)
    {
        return new Jacare(idade, nome);
    }
}
```

Construtores de instância (6)

Inicializadores de objeto

- Qualquer variável ou propriedade acessíveis de um objeto podem ser definidos via inicializadores de objeto.

```
public class Jaguar
{
    public string Nome;
    public int Idade;
    public string Pais;

}
```

```
var jaguar = new Jaguar()
{
    Nome = "Trincas",
    Idade = 2,
    Pais = "Índia"
};
```

Construtores de instância (7)

Parâmetros opcionais versus inicializadores de objeto

- Em vez dos incializadores de objeto, podemos criar um construtor que aceite parâmetros opcionais

```
public class Jaguar
{
    public string Nome;
    public int Idade;
    public string Pais;

    public Jaguar(string nome, int idade = 0, string pais = "none")
    {
        Nome = nome;
        Idade = idade;
        Pais = pais;
    }
}

var jaguar = new Jaguar()
{
    Nome = "Trincas",
    Idade = 2,
    Pais = "Índia"
};

// Chama construtor com parâmetros opcionais
var jaguarito = new Jaguar("Trinquitas");
```

Quais as vantagens e desvantagens de cada uma das abordagens?

Construtores de instância (8)

Modificadores de acesso

Modificadores de acesso	Descrição
`private`	• Só pode ser acedido a partir da classe onde foi declarado. • Nível de proteção definido, por omissão, no caso dos membros de uma classe (atributos, métodos, construtor, etc.)
`public`	• Pode ser acedido a partir de qualquer classe (inclusive fora do *assembly* onde foi definido).
`protected`	• Só pode ser acedido a partir da classe onde foi declarado e a partir de classes derivadas (inclusive fora do *assembly* onde foi definido).
`internal`	• Pode ser acedido a partir de qualquer classe dentro do *assembly* (excetuando fora do *assembly* onde foi definido).

Desconstrutores

- Método especial que serve para desconstruir um objeto
- Tem o nome especial `Deconstruct` e deve definir um ou mais parâmetros com o operador `out`.

```
public class Computer
{
    public int Storage { get; set; }

    public Computer(int storage)
    {
        Storage = storage;
    }

    public void Deconstruct(out int storage, out string name)
    {
        storage = Storage;
        name = "default";
    }
}

var computer = new Computer(10);

computer.Deconstruct(out var capacidade, out var name);
// ou
var (capacidade, nome) = computer;
// ou
(int capacidade, string nome) = computer;
```

Método destrutor (*finalizer*)

- Em C#, este método é o *finalizer*.
- Indica as instruções para o objeto executar antes de ser libertado da memória.
- Só existe 1 *finalizer* por objeto.
- O método deve iniciar com um til (~) e ter o mesmo nome da classe.

```
class Carro
{
    string cor, marca, modelo;
    int cilindrada, velocidade;
    ...

    ~Carro()
    {
       Console.WriteLine("finalizer em ação...");
    }
     ...
}
```

- Não pode ser chamado explicitamente.
- Utilizado para libertar recursos da memória que não sejam geridos pelo mecanismo de *garbage collection*. Como tal, deve ser usado em casos pontuais.
- Considere Implementar o interface IDisposable

Propriedades (1)

- Propriedades são métodos públicos que permitem alterar o valor dos atributos do objeto. Esses métodos são os *métodos* assessores (get e set) mas são apresentados como se se tratassem de um atributo.
- Por fora, assemelham-se a variáveis mas internamente podem conter lógica.
- Não é possível saber o que é uma variável ou propriedade a partir do código externo à classe onde a variável está definida.
- Uma propriedade é classicamente declarada como uma variável mas com blocos get/set

Propriedades (2)

Get e set são *assessores* de propriedade

- O **get** corre quando a propriedade é acedida e tem de devolver um valor do tipo da propriedade
- O **set** corre quando é atribuído um valor à propriedade
- Tem o parâmetro implícito “value” do tipo da propriedade que geralmente se associa a uma variável privada

```
class Moto
{
    private string _matricula;

    // Propriedade
    public string Matricula
    {
        get { return _matricula; }
        set { _matricula = value.Replace("-", ""); }
    }


    var moto = new Moto();
    moto.Matricula = "AA-99-BB";

    Console.WriteLine(moto.Matricula);  // Escreve AA99BB
}
```

Propriedades (3)

- A propriedade Matricula expõe publicamente o valor do atributo privado _matricula
- Permite controlar o set/get dos atributos privados. Neste caso, garantimos que _matricula nunca seja armazenado com hífen

```
class Moto
{
    private string _matricula;

    // Propriedade
    public string Matricula
    {
        get { return _matricula; }
        set { _matricula = value.Replace("-", ""); }
    }


    var moto = new Moto();
    moto.Matricula = "AA-99-BB";

    Console.WriteLine(moto.Matricula);  // Escreve AA99BB
}
```

Propriedades (4)

- Grande vantagem de não expôr os detalhes internos
- Pode também efetuar validações caso o valor esteja fora de um determinado limite, por exemplo, a idade não pode ser < 0
- As propriedade permitem os seguintes modificadores

Static Modifier	`Static`
Access modifiers	`public internal private protect`
Inheritance modifiers	`new virtual abstract override sealed`
Unmanaged code modifiers	`unsafe extern`

Propriedades (5)

- Podemos simplificar a declaração de propriedades quando não há lógica interna associada

Os exemplos seguintes são equivalentes, embora o segundo seja mais intuitivo e compacto

```
class Moto
{
    private string _matricula;

    public string Matricula
    {
        get { return _matricula; }
        set { _matricula = value; }
    }

}
```

```
class Moto
{
    public string Matricula { get; set; }

}
```

- O compilador gera automaticamente uma variável de suporte privada que não pode ser acedida. O acessor pode ser marcado como `private` ou `protected`.

Propriedades (6)

- Uma propriedade é `readonly` se apenas se definir o acessor get e é do tipo write-only quando se especifica unicamente o acessor set. Propriedade write-only são usadas muito raramente.
- Normalmente uma propriedade tem uma variável associada, mas também pode ser calculada a partir de outra

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

    public string NomeCompleto => $"{Nome} {Apelido}";
}
```

- A propriedade NomeCompleto é apenas de leitura e não dá para definir explicitamente

Propriedades (7)

Funcionalidades Recentes

- A partir de c# 6 é possível inicializar automaticamente as propriedades.

```
public decimal CurrentPrice { get; set; } = 123;
```

- A partir do c# 7 podemos estender ainda mais o conceito de propriedades “expression-bodied”

```
class Titulo
{
    private int numTitulos = 100;
    private decimal precoAtual;

    public decimal Valor
    {
        get => numTitulos * precoAtual;
        set => numTitulos = (int)(value / precoAtual);
    }
}
```

Construtores estáticos (1)

- Um construtor estático é executado apenas uma única vez por tipo de objeto, em vez de um por instância
- Não pode ter quaisquer parâmetros e o nome tem de ser definido da seguinte forma

```
class Test
{
  static Test() { Console.WriteLine ("Type Initialized"); }
}
```

- O *runtime* invoca automaticamente o construtor estático mesmo antes do tipo ser usado, nomeadamente:
    - No momento da instanciação do objeto
    - Acessando um membro estático da classe

Construtores estáticos (2)

Ordem de inicialização dos membros estáticos

- Atenção: A ordem de inicialização é importante

```
public class Program
{
    public static void Main()
    {
        Console.WriteLine(Calculadora.PI);
    }
}

public class Calculadora
{
    public static Calculadora StaticCalculadora = new Calculadora();
    public static double PI = 3.14;

    Calculadora()
    {
        Console.WriteLine(PI);
    }
}

// Output:
// 0
// 3.14
```

Classes estáticas

Uma classe pode ser marcada como estática e neste caso indica que:

- Só pode conter membros estáticos
- Não pode ser derivada
- Não pode ter quaisquer construtores
- Exemplos: System.Math e System.Console

```
public static class Calculadora
{
    public static int Somar(params int[]  parms) => parms.Sum();
    public static int Media(params int[] parms) => Somar(parms)/parms.Length;

    // Erro! Não pode ter métodos que não sejam estáticos
    public string Versao() => "2.1";
}

// OK
Calculadora.Somar(1, 2, 3, 4);

// Erro! Classes estáticas não podem ser instanciadas
Calculadora = new Calculadora();
```

Herança (1)

- É um dos principais conceitos de POO. A sua utilização permite a reutilização de código.
- Esta técnica de programação permite criar classes a partir de outras que já existem.
- Funciona como uma hierarquia:
  - Nível superior: classe-base
  - Nível inferior: classe-derivada
- A classe-derivada consegue herdar todos os atributos e métodos da classe original, logo não há duplicidade de código. Também é possível definir novos atributos e métodos.
- Sintaxe

```
class classe-derivada : classe-base
{
    //Aqui devem estar as instruções que definem a classe-derivada
}
```

Herança (2)

CLASSE-DERIVADA

```
 class CarroNovo : Carro
    {
        public bool descapotavel; //Novo atributo
        public void Arrancar() //Novo método
        {
            velocidade = velocidade + 1;
        }
        public CarroNovo(bool descapotavelOrigem) //Construtor
        {
            descapotavel = descapotavelOrigem;
        }
}
```

Herança (3)

CLASSE-BASE

```
class Carro
    {
        string cor, marca, modelo;
        public int cilindrada, velocidade;

        //2º construtor criado (sem parâmetros)
        public Carro()
        {
        }

        //Construtor
        public Carro(string corOrigem, int cilindradaOrigem)
        {
            //Código associado à construção do objeto:
            cor = corOrigem;
            cilindrada = cilindradaOrigem;
            velocidade = 0;
            Console.WriteLine("O objeto foi construído");
            Console.WriteLine("Pressione ENTER para destruí-lo...");
            Console.Read();
        }
          ...
}
```

Agora é simples instanciar um novo objeto (da classe CarroNovo), com base no objeto original (Classe Carro).

Herança (4)

```
class Start
{
   static void Main(string[] args)
   {
      CarroNovo carro = new CarroNovo(true); //2ª nova instância
      carro.velocidade = 0; //Atributo herdado da classe-mãe
      carro.Arrancar(); //Método da classe-filha
   }
}
```

Herança (5)

```
public class Vehicle
{
    public int MadeIn { get; set; }
    public string Brand { get; set; }
}

public class WheeledVehicle : Vehicle
{
    public int NumOfWheels { get; set; }
}

public class FlyingVehicle : Vehicle
{
    public int WingLength { get; set; }
}

public static class Foo
{
    public static void Boo()
    {
        var car = new WheeledVehicle(){MadeIn = 2000, NumOfWheels = 4};
        var plane = new FlyingVehicle(){Brand = "Avionic", MadeIn = 2012, WingLength = 500};
    }
}
```

Herança (6)

Chamada implícita ao construtor da classe base

- Se o construtor da subclasse omitir a keyword base, o construtor por omissão da classe base é chamado implicitamente.
- Se a classe base não tiver um construtor sem parâmetros acessível, as subclasses são forçadas a usar a keyword base nos seus construtores

```
public class BaseClass
{
  public int X;
  public BaseClass() { X = 1; }
}

public class Subclass : BaseClass
{
  public Subclass() { Console.WriteLine (X); }  // 1
}
```

Herança (7)

Keyword base

Similar ao **`this`** e serve 2 propósitos:

- Aceder a um membro de função *overriden* a partir de uma subclasse
- Chamar o construtor da classe base

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

    public Veiculo(string cor)
    {
        Cor = cor;
    }
}

class Moto : Veiculo
{
    public Moto(string cor) : base(cor)
    {
    }
}
```

Construtores e Herança (1)

- Uma subclasse deve declarar os seus próprios construtores. Os construtores da classe base são acessíveis (a menos que sejam privados) à classe derivada mas nunca são automaticamente herdados.
- A subclasse deve redefinir sempre os construtores que pretende expôr, podendo chamar os construtores da classe base com o operador `base`.
- Os construtores da classe base são sempre executados primeiro, assegurando que a inicialização da base ocorre antes da inicialização especializada.

Construtores e Herança (2)

Ordem de inicialização dos campos

- Da subclasse para a classe base
  - 1. Inicialização de variáveis
  - 2. Argumentos para o construtor da classe base são avaliados
  - 5. Execução do construtor

- Da classe base para a subclasse
  - 3. Inicialização de variáveis
  - 4. Execução do construtor

```
class A
{
    int x = 1;       // 3.º
    public A(int x)
    {
        x = 2;       // 4.º
    }
}

class B:A
{
    int y = 20;       // 1.º
    public B(int x)
        :base(x+1)    // 2.º
    {
        y++;          // 5.º
    }
}
```

Resolução de overloading

- Quando um método *overloaded* é chamado, a precedência vai para o método mais específico.

```
var moto = new Moto();
var veiculo = new Veiculo();

var dashboard = new Dashboard();
dashboard.VelocidadeAtual(moto);    // chama VelocidadeAtual(Moto moto)
dashboard.VelocidadeAtual(veiculo); // chama VelocidadeAtual(Veiculo veiculo)
```

Funções virtuais

- Um membro marcado como **`virtual`** pode ser *overriden* por uma subclasse que pretenda providenciar uma implementação especializada.
- Métodos, propriedades, indexers e eventos podem ser declarados como `virtual`.

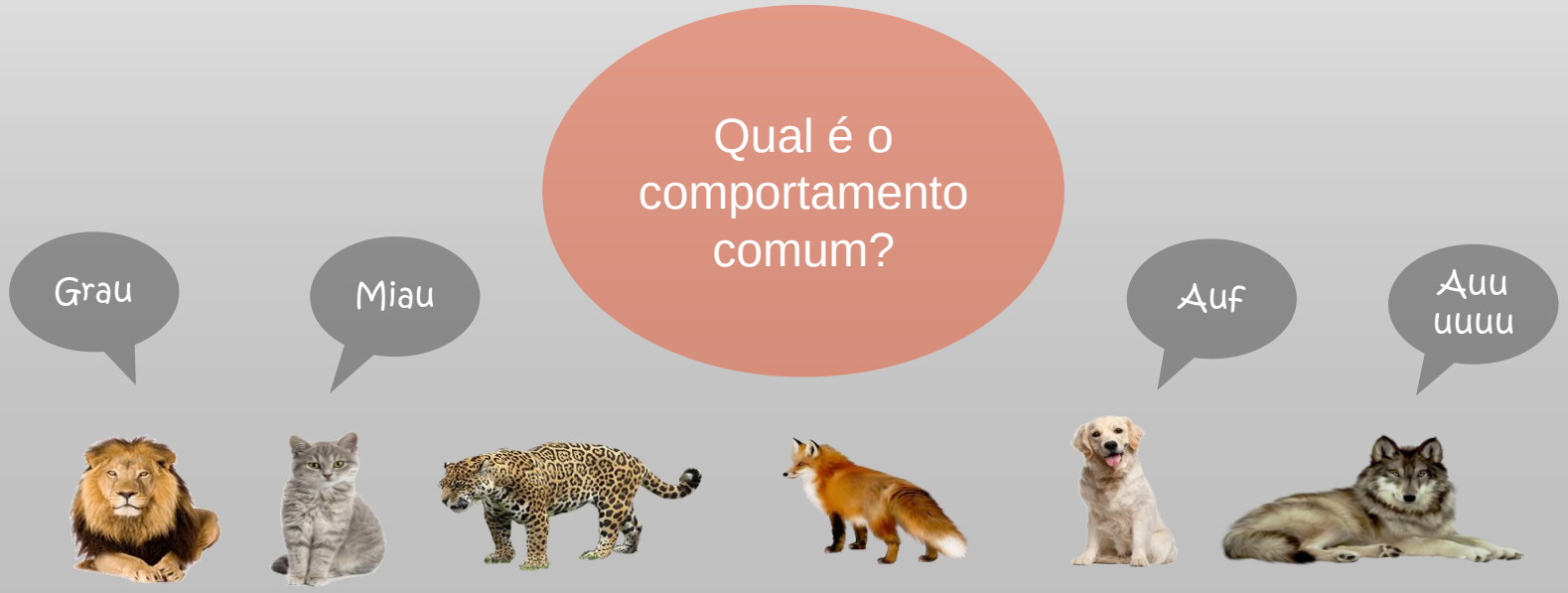
```
class Ativo
{
    public decimal InvestimentoInicial => 0;
    public decimal RetornoTotal => 0;

    public virtual decimal Lucro => RetornoTotal - InvestimentoInicial;
}

class InvestimentoBolsa : Ativo
{
    public decimal Comissoes { get; set; }
    public override decimal Lucro => base.Lucro - Comissoes;
}
```

Polimorfismo (1)

- **Poli (**várias**) + Morfos (**formas**).** Um objeto pode assumir diferentes formas.
- O polimorfismo está associado à herança.
- O conceito de polimorfismo distingue a possibilidade de 2 ou + objetos executarem a mesma ação.



Polimorfismo (2)

- As referências são polimórficas, o que significa que um objeto do tipo x pode apontar para um objeto derivado de x.

```
public static void Display(Vehicle vehicle)
{
    Console.WriteLine(value:$"MadeIn: {vehicle.MadeIn} Brand: {vehicle.Brand}");
}
```

- O método display() pode mostrar tanto carros como aviões, uma vez que ambos são veículos.

```
var car = new WheeledVehicle(){MadeIn = 2000, NumOfWheels = 4};
var plane = new FlyingVehicle(){Brand = "Avionic", MadeIn = 2012, WingLength = 500};

Vehicle.Display(car);
Vehicle.Display(plane);
```

- O polimorfismo assenta na premissa que as subclasses têm todas as características da classe base (Vehicle). No entanto, o oposto não é verdadeiro. Se a classe fosse modificada para aceitar um WheeledVehicle, então não poderíamos passar um avião

Conversões de referências: Casting

Uma referência de objeto pode ser:

- Implicitamente convertida para uma referência da classe base (*upcast*)
- Explicitamente convertida para uma referência da subclasse (*downcast*)
- Um *upcast* é sempre realizado com sucesso enquanto um downcast é apenas bem sucedido se o tipo de objeto for correto.

Upcasting

- Um upcasting cria uma referência da classe base a partir de uma referência de subclasse.

```
Veiculo veiculo;
var carro = new Carro();
veiculo = carro;              // upcasting: OK
```

- Após o upcasting, tanto veiculo como carro referenciam o mesmo carro. O objeto referenciado não é alterado ou convertido.

```
if (veiculo == carro)  // true
```

- Apesar de tanto o veiculo como o carro referenciaram o mesmo objeto, o veículo tem uma visão mais restrita do objeto pois não pode aceder às propriedades definidas na classe `Carro`.

Downcasting

- Uma operação do tipo downcasting cria uma referência de subclasse a partir da referência da base classe.

```
Veiculo veiculo;
var carro = new Carro();
veiculo = carro;              // upcasting: OK
var carroOne = (Carro)veiculo; // Downcasting: OK
```

- Tal como acontece no upcasting, tanto veiculo como o carroOne referenciam o mesmo objeto. O objeto referenciado não é alterado ou convertido.
- carroOne tem acesso a todos os membros do Carro, não estando restrito aos membros da classe base Veiculo
- O downcasting pode falhar quando o tipo não é compatível.

```
var veiculo = new Bicicleta();
var moto = (Moto)veiculo; // Downcasting: ERRO (Veiculo não é uma Moto)
```

Casting: Exemplo

```
public class Program
{
    public static void Main()
    {
        Veiculo veiculo;
        var carro = new Carro();
        veiculo = carro;                // upcasting: OK
        var carroOne = (Carro)veiculo; // Downcasting: OK

        veiculo = new Bicicleta();
        var MotoOne = (Moto)veiculo; // Downcasting: Fail (Veiculo não é uma Moto)
    }
}

public class Veiculo { }

public class Carro : Veiculo { }

public class Moto : Veiculo { }

public class Bicicleta : Veiculo { }
```

Operador is

- O operador `is` testa se uma conversão terá sucesso; em outras palavras, quando um objeto deriva de uma classe específica (ou implementa um interface). É geralmente usada antes de uma operação de downcasting.

```
if (veiculo is Moto)
    moto = (Moto)veiculo;
else
    throw new InvalidCastException(
          $"Não pode converter um objeto do tipo {veiculo.GetType()} para uma Moto");
```

O operador is e pattern variables (C# 7)

- A partir do C# 7, podemos inserir a variável na expressão e esta fica com o tipo correto se a expressão for verdadeira

```
if (veiculo is Carro carro)
{
    Console.WriteLine($"Cor: {carro.Cor}");
}
```

é equivalente a

```
if (veiculo is Carro)
{
    carro = (Carro)veiculo;
    Console.WriteLine($"Cor: {carro.Cor}");
}
```

- A variável inserida fica disponível para utilização imediata e continua disponível mesmo fora do contexto da expresão `is`

```
if (veiculo is Carro carro)
    Console.WriteLine($"Cor: {carro.Cor}");
else
    carro = new Carro();       // carro está disponível

Console.WriteLine(carro.Cor);  // carro continua disponível
```

O operador as

- O operador **as** faz o *downcast* quando o tipo é correto mas atribui null à variável em vez de lançar uma exceção se este falhar.

```
var novoCarro = veiculo as Carro; // novoCarro = null.
```

- O *casting* explícito “deve” ser usado quando temos a certeza que estamos a converter para o tipo correto

Ocultação de membros herdados

- Uma classe base e uma subclass podem definir membros idênticos. Por exemplo

```
public class A      { public int Counter = 1; }
public class B : A  { public int Counter = 2; }
```

- O campo `counter` da class B esconde o campo counter da classe A. Acontece usualmente por acidente
- O compilador gera um aviso e resolve a ambiguidade da seguinte forma:

- Referência a A => A.Counter
- Referência a B => B.Counter

Se o objectivo for mesmo o de esconder o membro deliberadamente, pode usar-se o operador “`new`”, que não faz nada mais do que suprimir o aviso do compilador.

New versus override

- O modificador **new** indica a itenção ao compilador e a outros programadores, que o membro duplicado não é acidental.

```
public class Animal
{
    public virtual void Passear() { Console.WriteLine("Animal: Passeando"); }
}

public class Coelho : Animal
{
    public new void Passear() { Console.WriteLine("Coelho: Fugindo da raposa"); }
}

public class Raposa : Animal
{
    public override void Passear() { Console.WriteLine("Raposa: Caçando coelhos"); }
}
```

```
var coelho = new Coelho();
Animal animal = coelho;
coelho.Passear(); // Coelho: Fugindo da raposa
animal.Passear(); // Animal: Passeando

var raposa = new Raposa();
animal = raposa;
raposa.Passear(); // Raposa: Caçando coelhos
animal.Passear(); // Raposa: Caçando coelhos
```

Sealing funções e classes

- Only overrided members can me marked as sealed
- Também se pode aplicar sealed a uma classe, “sealing” implicitamente todas as funções virtuais.
- Aplicar `sealed` a uma classe é mais comum do que a um membro de função.

```
class Veiculo
{
    public virtual string Cor { get; set; }
}

class Moto : Veiculo
{
    public sealed override  string Cor { get; set; }
}
class MotoVoadora : Moto
{
   // ERRO: Propriedade Cor foi selada na classe Moto
    public override string Cor { get; set; }
}
```

Classes abstratas (1)

- Em POO as classes abstratas são utilizadas para atribuir funcionalidades iguais a objetos diferentes.
- Uma classe abstrata não pode ser instanciada, só as classes que derivam dela.
- Quando uma classe tem um método abstrato, deve ser declarada como abstrata. Mas, é possível ter uma classe abstrata que não tenha métodos abstratos.
- Quando existe um método abstrato que não possui corpo, obriga a que nas classes derivadas o mesmo tenha de ser implementado, caso contrário, serão também elas classes abstratas.
- Sintaxe

```
abstract class <nome_classe_abstrata>
{
  <bloco de instruções>;
}
```

Classes abstratas (2)

- No exemplo ***a***, a classe não possui nenhum método abstrato, logo as classes que derivem desta não precisam de implementar, obrigatoriamente, um método.
- No exemplo ***b***, a classe Carro (que deriva da classe Veiculo), terá de implementar o método *DevolveNumeroRodas()*, obrigatoriamente.

O modificador de acesso *override* é utilizado para implementar um método que deriva de uma classe base.

```
//EXEMPLO a

        abstract class Veiculo
        {
            ...
        }
```

```
// EXEMPLO b

        abstract class Veiculo
        {
            ...
            public abstract int DevolveNumeroRodas();
            ...
        }
        class Carro : Veiculo
        {
            int numeroRodas;
             ...
             public override int DevolveNumeroRodas()
             {
                return numeroRodas;
             }
        ...
        }
```

Interfaces (1)

- Uma interface representa um contrato que uma classe tem que cumprir.
- Uma classe quando implementa uma determinada interface compromete-se a implementar todos os métodos identificados nessa interface.
- O nome deve começar por I, apesar de não ser obrigatório e.g. `IValidator`, `IEnumerator`, etc.

- Sintaxe:

```
interface <nome>
{
  …
}
```

Interfaces (2)

- Similar a uma classe, providenciando uma especificação em vez de uma implementação para os seus membros. Tem as seguintes características:
    - Todos os seus membros são implicitamente abstratos. Em contraste, uma classe pode providenciar membros abstratos e membros concretos com implementação
    - Uma classe pode implementar múltiplos interfaces mas herdar apenas de uma única classe

Exemplo do interface `IEnumerator`

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

Interfaces (3)

```
interface IAutentica
{
   bool Validar(string username, string password);
   void Sair();
}
```

```
class Utilizador : IAutentica
{
   public bool Validar(string user, string password)
   {
      if (user == "admin" && password == “never store a pwd like this”)
      {
         return true;
      }
      else
      {
         return false;
      }
   }
   public void Sair()
   {
      Console.WriteLine("Até à próxima.");
   }
}
```

Interfaces (4)

- Os membros de uma interface são sempre públicos por omissão e não podem declarar modificadores de acesso.

```
public interface IVeiculo
{
    int AnoFabrico { get; set; }
    void Arrancar();
    void Desacelarar();
}

public class Submarino : IVeiculo
{
    public int AnoFabrico { get; set; }

    public void Arrancar()
    {
        throw new NotImplementedException();
    }

    public void Desacelarar()
    {
        throw new NotImplementedException();
    }
 }


IVeiculo veiculo = new Submarino();

// Podemos aceder aos membros da classe Submarino através da interface IVeiculo
veiculo.AnoFabrico = 2002;
veiculo.Arrancar();
veiculo.Desacelarar();
```

Interfaces (5)

Extensão de interfaces

- Uma interface pode derivar de outras interfaces

```
public interface IVeiculo
{
    int AnoFabrico { get; set; }
    void Arrancar();
    void Desacelarar();
}

public interface IVeiculoAquatico : IVeiculo
{
    void Mergulhar();
}

public class Submarino : IVeiculoAquatico
{
    public int AnoFabrico { get; set; }

    public void Arrancar()      // Implementação para a interface IVeiculo
    {
        throw new NotImplementedException();
    }

    public void Desacelarar()   // Implementação para a interface IVeiculo
    {
        throw new NotImplementedException();
    }

    public void Mergulhar()     // Implementação para a interface IVeiculoAquatico
    {
        throw new NotImplementedException();
    }
}
```

Implementação explícita de interfaces

- A implementação de interfaces múltiplos pode originar colisão entre as assinaturas dos membros. Estas podem ser resolvidas ao associar explicitamente cada método em conflito à respetiva interface.

```
public interface IVoar
{
    void DefinirDestino(string destino);
}

public interface IAndar
{
    void DefinirDestino(string destino);
}

public class Gaivota : IVoar, IAndar
{
    void IVoar.DefinirDestino(string destino)
    {
        Console.WriteLine($"Voando livremente até {destino}");
    }

    void IAndar.DefinirDestino(string destino)
    {
        Console.WriteLine($"Caminhando lentamente até {destino}");
    }
}
```

Implementação explícita de interfaces

- A única forma de chamar um membro definido explicitamente é fazendo um *cast* para a interface.

```
var gaivota = new Gaivota();

((IAndar)gaivota).DefinirDestino("Lisboa"); // Caminhando lentamente até Lisboa
((IVoar)gaivota).DefinirDestino("Paris");   // Voando livremente até Paris
```

```
public class Gaivota : IVoar, IAndar
{
    void IVoar.DefinirDestino(string destino)
    {
        Console.WriteLine($"Voando livremente até {destino}");
    }

    void IAndar.DefinirDestino(string destino)
    {
        Console.WriteLine($"Caminhando lentamente até {destino}");
    }
}
```

Implementação de membros de interface virtuais

- Um membro de interface implícito é sealed por omissão. Deve ser marcado como **`virtual`** ou **`abstract`** na classe base para que possa ser overriden.

```
interface IAndar
{
    void Mover();
}

class Animal : IAndar
{
    public virtual void Mover() => Console.WriteLine("Anda Animal");
}

class Hiena : Animal
{
    public override void Mover() => Console.WriteLine("Anda Hiena");
}

var hiena = new Hiena();
hiena.Mover();              // Anda Hiena
((IAndar)hiena).Mover();    // Anda Hiena
((Animal)hiena).Mover();    // Anda Hiena
```

Reimplementação de interfaces em subclasses

- Uma subclasse pode reimplementar qualquer membro de interface implementado na classe base.
- A reimplementação “sequestra” a implementação do membro (quando chamada via interface) e funciona quer o membro seja ou não virtual na classe base. Também funciona quer o membro seja implememententado de forma implícita ou explícita, apesar de funcionar melhor no último caso.

```
interface IAndar
{
    void Mover();
}

class Animal : IAndar
{
    public void Mover() => Console.WriteLine("Anda Animal");
}

class Hiena : Animal, IAndar
{
    public void Mover() => Console.WriteLine("Anda Hiena");
}

var hiena = new Hiena();
hiena.Mover();              // Anda Hiena
((IAndar)hiena).Mover();    // Anda Hiena
((Animal)hiena).Mover();    // Anda Animal
```

Boas práticas: Classes versus interfaces

Como directriz:

- Use classes e subclasses para tipos que naturalmente partilham uma implementação
- Use interfaces para tipos que têm implementações independentes