
PRÁTICAS DE C#
PROGRAMAÇÃO ORIENTADA POR OBJETOS


PROGRAMAR
programação orientada a objectos em C#

Programação Estruturada vs Programação Orientada a Objetos

Hoje em dia, praticamente todo o software está a ser escrito em linguagens de programação orientadas a objetos (POO).

Por exemplo, num programa em programação estruturada todas as operações têm acesso a todos os dados, e uma mudança num dos módulos pode ter implicações em todo o programa. A programação orientada a objectos alivia alguns desses problemas, **encapsulando** cada uma das **estruturas de dados** num objeto. O objeto também possui funções, chamadas em POO de **métodos**, que permitem interagir com o objeto.

POO – Características

A POO assenta em três conceitos base:

- **Encapsulamento de informação;**
- **Herança;**
- **Polimorfismo.**

- Encapsulamento de informação

Um dos pontos fulcrais do POO é esconder as estruturas de dados em objetos, aos quais são associados métodos que manipulam essas estruturas. Para este processo o programador tem de definir classes, que são um tipo abstracto de dados ou uma família de dados. Cada classe costuma ter **as *variáveis, método(s) construtor(es), métodos que modificam ou devolvem o valor das variáveis***. Antes de entrar em detalhes sobre o que é cada um, é melhor tomar em atenção o seguinte código:

```
class Aluno
{
        private int Numero;             //Número do aluno
        private string Nome;            //Nome do aluno
        private string NomeEscola;      //Nome da escola

        public Aluno(int numeroAluno,string nomeAluno, string nomeEscola)
        {
                Numero = numeroAluno;
                Nome = nomeAluno;
                NomeEscola = nomeEscola;
        }

        public void MostraAluno()
        {
                Console.WriteLine("O aluno número {0} chama-se {1} e estuda na
                escola {2}.", Numero, Nome, NomeEscola);
        }

        public int DevolveNumero()
        {
                return Numero;
        }

        public void ModificaNumero(int Numero)
        {
                this.Numero = Numero;
        }
}
```

A classe *Aluno* possui três variáveis declaradas como **private**: Numero, Nome e NomeEscola. Isto significa que só podem ser acedidas a partir da própria classe, ou seja, a informação é escondida dentro da classe. Se estivesse declarada como **public** podia ser acedida a partir de uma classe externa. Pode-se observar a existência de um construtor: public Aluno(). A sua função é a de criar uma nova instância de uma classe, ou seja, cria um objeto dessa classe. Também irá guardar a informação dos atributos que são passados no construtor no interior da classe, tal como é visto no código, já que as variáveis da classe tomam o valor das passadas no construtor. Para criar o objeto do tipo *Aluno* basta fazer:

```
Aluno aluno1 = new Aluno(10, "Filipe", "Escola XPTO");
```

Depois existem três métodos. O primeiro, MostraAluno(), quando é chamado imprime a frase com os dados do aluno, ou seja, ao invocar aluno1.MostraAluno() vai imprimir a seguinte frase:

"O aluno número 10 chama-se Filipe e estuda na escola Escola XPTO."

O método DevolveNumero() é chamado sempre que seja necessário saber o número do aluno. Como a variável *Numero* da classe Aluno está declarada como private, é necessário criar esse método para saber o seu valor. A seguinte invocação:

```
Console.WriteLine("O aluno tem o número {0}.",aluno1.DevolveNumero());
```

resulta na seguinte frase: *"O aluno tem o número 10."*

O último método, ModificaNumero(int Numero), é usado para modificar o número do aluno, já que a variável *Numero* está declarada como **private** e é necessário este método que para entidades exteriores à classe possam modificar o seu conteúdo. Desta forma, a seguinte sequência:

```
aluno1.ModificaNumero(20);
aluno1.MostraAluno();
```

imprime a seguinte frase:

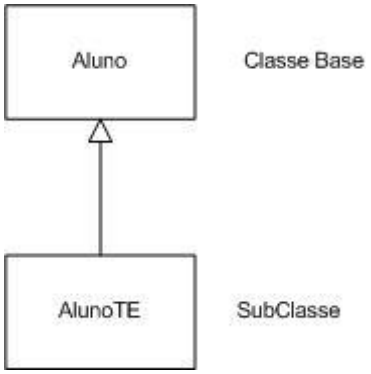
"O aluno número 20 chama-se Filipe e estuda na escola Escola XPTO."

De notar que no método que modifica a idade do aluno foi usado a cláusula **this**, que significa o próprio objeto. Como a variável da classe e a passada por referência no método tinham o mesmo nome foi necessário indicar que quem tomava o novo valor era a variável do objecto. É de notar que as outras classes podem chamar estes métodos porque estão declarados como **public**, caso estivessem declarados como **private** os métodos só podiam ser acedidos a partir da própria classe.

Herança

Esta é uma das características mais visíveis e com mais uso quando se desenvolve uma aplicação usando POO. Para compreender melhor o conceito de Herança, vai-se tomar em conta a classe *Aluno* anteriormente descrita. Imagine-se, agora, que é necessário criar mais uma classe para os alunos que são trabalhadores estudantes e que essa classe apenas vai ter mais um campo (o nome da empresa onde trabalha) que a classe *Aluno*. Será que faz sentido estar a criar uma classe *AlunoTE* que repita os mesmos atributos que a classe *Aluno*? Afinal de contas, um trabalhador estudante é um aluno, e como tal possui todas as características de um aluno normal. Assim surge o conceito de **herança**, ou seja, é possível criar uma nova classe que derive de outra. Contudo nem sempre o conceito de **herança** é bem utilizado. Apenas se deve usar quando uma classe é uma especialização da outra, isto é, quando possui

todos os elementos que a outra possui mais alguns, sejam estes métodos ou dados. É usual chamar à classe geral de **superclasse** ou de **classe base**, enquanto as que derivam destas são chamadas de **subclasses**.



Para perceber melhor o conceito de **herança**, em seguida está exemplificado o código da classe *AlunoTE* que deriva da classe ***Aluno***:

```
class AlunoTE : Aluno
{
        private string NomeEmpresa; //Nome da empresa

        public AlunoTE(int numeroAluno, string nomeAluno, string nomeEscola, string NomeEmpresa)
                        : base(numeroAluno, nomeAluno, nomeEscola)
        {
                this.NomeEmpresa = NomeEmpresa;
         }

         public void MostraAlunoTE()
         {
                 Console.WriteLine("O aluno trabalha na empresa {0}.", NomeEmpresa);
         }
}
```

Como se pode observar, a classe *AlunoTE* é definida como sendo subclasse de *Aluno* logo no início: class AlunoTE : Aluno. Desta forma herdou todas as variáveis e métodos presentes na classe *Aluno*. Em seguida é definido um atributo, que é o nome da empresa em que o aluno trabalha. Depois vem o método construtor, que recebe como argumentos todos os atributos da classe *Aluno* mais o atributo *NomeEmpresa*. Nota-se uma novidade que é o uso do : base(). O que isto faz é inicializar as variáveis na classe base antes de fazer o mesmo na própria classe. Depois disso é feita a inicialização da variável *NomeEmpresa* no corpo do construtor da própria classe. Também foi criado um método local que imprime o nome da empresa onde o aluno trabalha. Por fim, já se podem criar objectos da classe *AlunoTE*, e fazer uso das funcionalidades que a herança permite, como por exemplo, o seguinte código:

```
AlunoTE aluno2 = new AlunoTE(15, "Joel", "Escola XPTO", SoftPROGRAMAR");

aluno2.MostraAluno();
aluno2.MostraAlunoTE();
```

vai imprimir:

"O aluno número 15 chama-se Joel e estuda na escola Escola XPTO." "O aluno trabalha na empresa SoftPROGRAMAR."

Polimorfismo

Por polimorfismo entende-se a capacidade de objectos diferentes se comportarem de modos distintos quando lhes é chamado o mesmo método. Por exemplo, a classe AlunoTE possui um método que imprime o nome da empresa, para imprimir o número, o nome e o nome da escola tem de chamar o método da classe base. Mas pode ser feita a redefinição do método MostraAluno() e fazer com que se imprimam todos os dados, tanto da classe base como da derivada. Em C# basta criar um método com o mesmo nome na classe derivada (declarado como override), declarando ao mesmo tempo o método da classe base como virtual. Na classe *Aluno* o método ficaria assim:

```
public virtual void MostraAluno()
{
   Console.WriteLine("O aluno número {0} chama-se {1} e estuda na escola {2}", Numero, Nome, NomeEscola);
}
```

Na classe *AlunoTE* seria criado o novo método da seguinte forma:

```
public override void MostraAluno()
{
   Console.WriteLine("O aluno número {0} chama-se {1}, estuda na escola {2} e trabalha na empresa {3}.",
Numero, Nome, NomeEscola, NomeEmpresa);
}
```

Desta forma quando for criado um aluno do tipo *AlunoTE* e invocar esse método:

```
AlunoTE aluno3 = new AlunoTE(5, “Pedro”, “Escola XPTO”, “SoftPROGRAMAR”);
aluno3.MostraAluno();
```

vai obter a seguinte mensagem:

“O aluno número 5 chama-se Pedro, estuda na escola Escola XPTO e trabalha na empresa SoftPROGRAMAR.”

Níveis de acesso a membros

Ao longo deste artigo têm sido dados exemplos de atributos, métodos e classes declarados como **public** e **private**. No entanto existem outros níveis de acesso a membros, como é o caso do **protected**, **internal** e **protected internal** (os dois últimos são características do C#, em Java por exemplo não existem).

Nível de acesso	Visibilidade
Private	Dentro da própria classe
Protected	Dentro da própria classe e em classes derivadas
Public	Dentro e fora da própria classe
Internal	São como os membros public, mas apenas visíveis dentro do módulo corrente
Protected internal	São como os membros protected, mas apenas visíveis dentro do módulo corrente

Tabela 1 – Níveis de acesso a membros e respectiva visibilidade.

Membros estáticos

Por vezes surge a necessidade que determinadas classes partilhem informação. Por exemplo, imaginando que todos os alunos andam na mesma escola, no caso da classe *Aluno* existe um campo que é comum a todos os alunos da mesma escola, que é o nome da escola (*NomeEscola*). Sempre que haja uma alteração no nome da escola, deve ser visível em todas as instâncias da classe. Surge então o conceito de **static**. No exemplo da classe *Aluno*, imagine-se que o nome da escola era declarado como **static**:

```
private static string NomeEscola;
```

Este atributo vai ser partilhado por todas as instâncias da classe *Aluno*. Para alterar o seu conteúdo, declara-se também um método estático:

```
public static void MidificaEscola(string novoNome)
{
   this.NomeEscola = novoNome;
}
```

Se houverem dois objetos e o método for chamado apenas num deles, o outro objeto também vai alterar o nome da escola.

Construtores estáticos

Tal como foi dito anteriormente, um membro estático não está associado a nenhum objeto em concreto, por isso faz todo o sentido o membro estático ser inicializado dentro de um construtor estático. Um construtor estático apenas difere dos apresentados anteriormente pelo facto de ser declarado como **static**.

```
private static string Atributo;

static ClasseXPTO()
{
    Atributo = "AtributoX";
}
```

Conclusão

Com este documento pretendeu-se que o leitor fique a perceber, de uma forma superficial, o paradigma de programação orientado a objetos. Partiu-se do princípio que o leitor já possui alguns conhecimentos em C#, ou então em linguagens cuja sintaxe é parecida: C/C++ ou Java, podendo desta forma dar alguns exemplos práticos. Nem todos os aspetos foram tratados neste documento, pelo que se aconselha a pesquisar sobre este assunto, nomeadamente (e os mais importantes) no que diz respeito a classes seladas (usadas para prevenir derivação de uma classe, ocorre um erro se uma classe selada for especificada como uma classe base de uma outra classe), classes abstratas (usadas para indicar que esta classe é orientada apenas para ser uma classe base de outras classes e não pode ser instanciada) e interfaces (contêm apenas assinaturas dos métodos, a implementação dos mesmos é feita na classe que implementa a interface).