Conteúdo

Unidade 4

4 Modificadores de acesso e atributos de classe

Ao término desse capítulo, você será capaz de:

- Controlar o acesso aos seus métodos, atributos e construtores através dos modificadores private e public;
- Escrever propriedades de acesso a atributos do tipo getters e setters;
- Escrever construtores para suas classes;
- Utilizar variáveis e métodos estáticos.

4.1 - Controlando o acesso

Um dos problemas mais simples que temos no nosso sistema de contas é que a função saca permite sacar mesmo que o limite tenha sido atingido. A seguir você pode lembrar como está a classe Conta:

```
public class Conta
{
        public int numero;
        public Cliente titular;
        public double saldo;
        public double limite;
        // ..
        public void Saca(double quantidade)
        {
            this.saldo = this.saldo - quantidade;
        }
}
```

A classe a seguir mostra como é possível ultrapassar o limite usando o método saca:

```
public class TestaContaEstouro1
{
        public static void main(String[] args)
        {
            Conta minhaConta = new Conta();
            minhaConta.saldo = 1000.0;
            minhaConta.limite = 1000.0;
            minhaConta.saca(50000); // saldo + limite é só 2000!!
        }
}
```

Podemos incluir um if dentro do nosso método saca() para evitar a situação que resultaria em uma conta em estado inconsistente, com seu saldo abaixo do limite. Fizemos isso no capítulo de orientação a objetos básica.

Apesar de melhorar bastante, ainda temos um problema mais grave: ninguém garante que o usuário da classe vai sempre utilizar o método para alterar o saldo da conta. O código a seguir ultrapassa o limite diretamente:

```
class TestaContaEstouro2
{
        public static void main(String[] args)
        {
            Conta minhaConta = new Conta();
            minhaConta.limite = 100;
            minhaConta.saldo = -200; //saldo abaixo dos 100 de limite
        }
}
```

Como evitar isso? Uma idéia simples seria testar se não estamos ultrapassando o limite toda vez que formos alterar o saldo:

```
class TestaContaEstouro3
{
        static void Main(String[] args)
        {
            // a Conta
            Conta minhaConta = new Conta();
            minhaConta.limite = 100;
            minhaConta.saldo = 100;
            // quero mudar o saldo para -200
            double novoSaldo = -200;
            // testa se o novoSaldo ultrapassa o limite da conta
            if (novoSaldo < -minhaConta.limite)
            { //
                Console.WriteLine("Não posso mudar para esse saldo");
            }
            else
            {
                minhaConta.saldo = novoSaldo;
            }
        }
}
```

Esse código iria se repetir ao longo de toda nossa aplicação e, pior, alguém pode esquecer de fazer essa comparação em algum momento, deixando a conta na situação inconsistente. A melhor forma de resolver isso seria forçar quem usa a classe Conta a chamar o método saca e não permitir o acesso direto ao atributo. É o mesmo caso da validação de CPF.

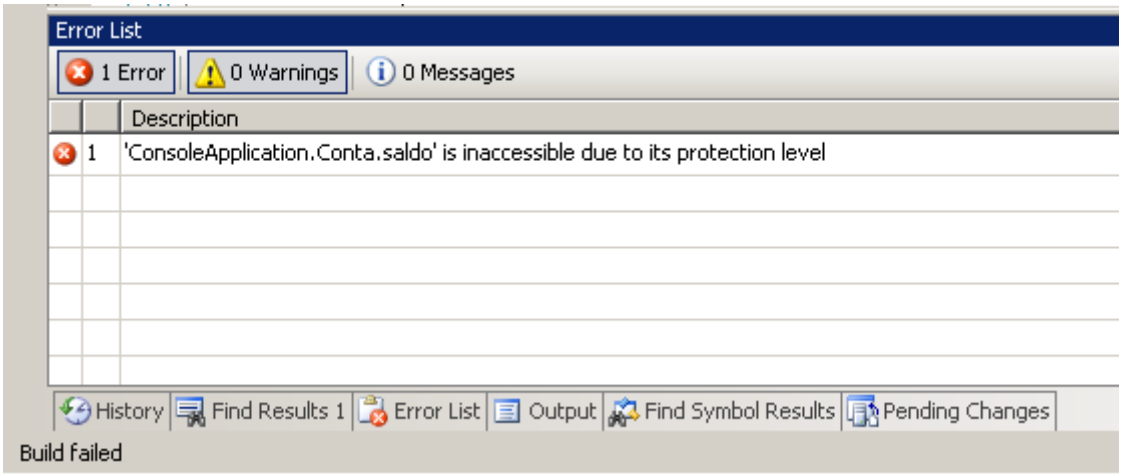
Para fazer isso no C#, basta declarar que os atributos não podem ser acessados de fora da classe usando a palavra chave private:

```
public class Conta
{
        private double saldo;
        private double limite;
        // ...
}
```

private é um modificador de acesso (também chamado de modificador de visibilidade).

Marcando um atributo como privado, fechamos o acesso ao mesmo de todas as outras classes, fazendo com que o seguinte código não compile:

```
class TestaAcessoDireto
{
        static void Main(String[] args)
        {
            Conta minhaConta = new Conta();
            //não compila! você não pode acessar
            //o atributo privado de outra classe
            minhaConta.saldo = 1000;
        }
}
```



Enquanto programando orientado a objetos, é prática quase que obrigatória proteger seus atributos com private. (discutiremos outros modificadores de acesso em outros capítulos).

Cada classe é responsável por controlar seus atributos, portanto ela deve julgar se aquele novo valor é válido ou não! Esta validação não deve ser controlada por quem está usando a classe e sim por ela mesma, centralizando essa responsabilidade e facilitando futuras mudanças no sistema.

Repare que, agora, quem chama o método saca não faz a menor idéia de que existe um limite que está sendo checado. Para quem for usar essa classe, basta saber o que o método faz e não como exatamente ele o faz (o que um método faz é sempre mais importante do que como ele faz: mudar a implementação é fácil, já mudar a assinatura de um método vai gerar problemas).

A palavra chave private também pode ser usada para modificar o acesso a um método. Tal funcionalidade é normalmente usada quando existe um método apenas auxiliar a própria classe e não queremos que outras pessoas o usem (ou apenas para

seguir a boa prática de expôr-se ao mínimo). Sempre devemos expôr o mínimo possível de funcionalidades, para criar um baixo acoplamento entre as nossas classes.

Da mesma maneira que temos o private, temos o modificador public, que permite a todos acessarem um determinado atributo ou método:

```
class Conta
{
        //...
        public void Saca(double quantidade)
        {
            if (quantidade > this.saldo + this.limite)
            {
                  //posso sacar até saldo+limite
                  Console.WriteLine("Não posso sacar fora do limite!");
            }
            else
            {
                this.saldo = this.saldo - quantidade;
            }
        }
}
```

E quando não há modificador de acesso?

As vezes tínhamos declarado variáveis e métodos sem nenhum modificador como private e public. Quando isto acontece, o seu método ou atributo fica com estado de visibilidade privado, que veremos mais pra frente, no capítulo de pacotes.

É muito comum, e faz todo sentido, que seus atributos sejam private e quase todos seus métodos sejam public (não é uma regra!). Desta forma, toda conversa de um objeto com outro é feita por troca de mensagens, isto é, acessando seus métodos. Algo muito mais educado que mexer diretamente em um atributo que não é seu!

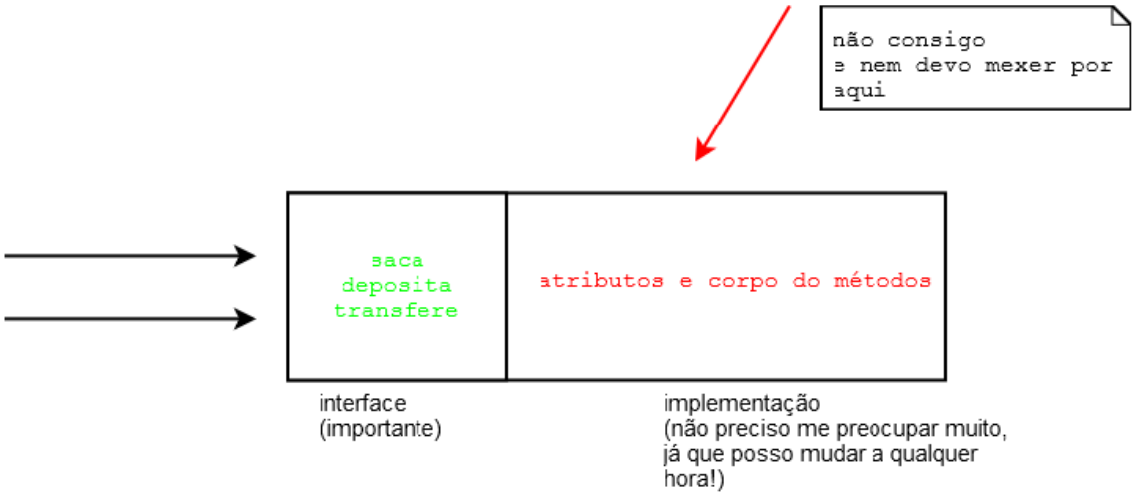
Melhor ainda! O dia em que precisarmos mudar como é realizado um saque na nossa classe Conta, adivinhe onde precisaríamos modificar? Apenas no método saca, o que faz pleno sentido. Como exemplo, imagine cobrar CPMF de cada saque: basta você modificar ali, e nenhum outro código, fora a classe Conta, precisará ser recompilado. Mais: as classes que usam esse método nem precisam ficar sabendo de tal modificação! Você precisa apenas recompilar aquela classe e substituir aquele arquivo .cs.

4.2 - Encapsulamento

O que começamos a ver nesse capítulo é a idéia de **encapsular**, isto é, esconder todos os membros de uma classe (como vimos acima), além de esconder como funcionam as rotinas (no caso métodos) do nosso sistema.

Encapsular é **fundamental** para que seu sistema seja suscetível a mudanças: não precisaremos mudar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que essa regra está **encapsulada**.

(veja o caso do método saca)



O conjunto de métodos públicos de uma classe é também chamado de **interface da classe**, pois esta é a única maneira a qual você se comunica com objetos dessa classe.

Programando voltado para a interface e não para a implementação

É sempre bom programar pensando na interface da sua classe, como seus usuários a estarão utilizando, e não somente em como ela irá funcionar.

A implementação em si, o conteúdo dos métodos, não tem tanta importância para o usuário dessa classe, uma vez que ele só precisa saber o que cada método pretende fazer, e não como ele faz, pois isto pode mudar com o tempo.

Essa frase vem do livro Design Patterns, de Eric Gamma. Um livro cultuado no meio da orientação a objetos.

Sempre que vamos acessar um objeto, utilizamos sua interface. Existem diversas analogias fáceis no mundo real:

- Quando você dirige um carro, o que te importa são os pedais e o volante (interface) e não o motor que você está usando (implementação). É claro que um motor diferente pode te dar melhores resultados, mas **o que ele faz** é a mesma coisa que um motor menos potente, a diferença está em **como ele faz**. Para trocar um carro a álcool para um a gasolina você

não precisa reapreender a dirigir! (trocar a implementação dos métodos não precisa mudar a interface, fazendo com que as outras classes continuem usando eles da mesma maneira).

- Todos os celulares fazem a mesma coisa (interface), eles possuem maneiras (métodos) de discar, ligar, desligar, atender, etc. O que muda é como eles fazem (implementação), mas repare que para o usuário comum pouco importa se o celular é GSM ou CDMA, isso fica encapsulado na implementação (que aqui são os circuitos).

Repare que, agora, temos conhecimentos suficientes para resolver aquele problema da validação de CPF:

```
public class Cliente
{
        private String nome;
        private String endereco;
        private String cpf;
        private int idade;

        public void MudaCPF(String cpf)
        {
            ValidaCPF(cpf);
            this.cpf = cpf;
        }

        private void ValidaCPF(String cpf)
        {
            // série de regras aqui, falha caso nao seja válido
        }
        // ..
}
```

Agora, se alguém tentar criar um Cliente e não usar o MudaCPF, vai receber um erro de compilação, já que o atributo CPF é **privado**. E o dia que você não precisar validar quem tem mais de 60 anos? Seu método fica o seguinte:

```
public void MudaCPF(String cpf)
{
      if (this.idade <= 60)
      {
            ValidaCPF(cpf);
      }
      this.cpf = cpf;
}
```

O controle sobre o CPF está centralizado: ninguém consegue acessá-lo sem passar por aí, a classe Cliente é a única responsável pelos seus próprios atributos!

4.3 - Propriedades

Para permitir o acesso aos atributos (já que eles são private) de uma maneira controlada, a prática mais comum é criar uma propriedade, que retorna o valor e outro que muda o valor.

O padrão para essas propriedades é de colocar o nome do atributo com a primeira letra em maiúscula. Por exemplo, a nossa conta com saldo, limite e titular fica assim:

```
public class Conta
{
        private double _saldo;
        private double _limite;
        private Cliente _titular;

        public double Saldo
        {
            get { return _saldo; }
            set { _saldo = value; }
        }

        public double Limite
        {
            get { return _limite; }
            set { _limite = value; }
        }

        public Cliente Titular
        {
            get { return _titular; }
            set { _titular = value; }
        }
}
```

É uma má prática criar uma classe e, logo em seguida, criar propriedades pros seus atributos. Você só deve criar uma propriedade se tiver a real necessidade. Repare que nesse exemplo Saldo não deveria ter sido criado, já que queremos que todos usem Deposita() e Saca().

Outro detalhe importante, uma propriedade X não necessariamente retorna o valor de um atributo que chama _x do objeto em questão. Isso é interessante para o encapsulamento. Imagine a situação: queremos que o banco sempre mostre como saldo o valor do limite somado ao saldo (uma prática comum dos bancos que costuma iludir seus clientes). Poderíamos sempre chamar c.Limite + c.Saldo, mas isso poderia gerar uma situação de “replace all” quando precisássemos mudar como o saldo é mostrado. Podemos encapsular isso em um método e, porque não, dentro da propriedade Saldo? Repare:

```
public class Conta
{
        private double _saldo;
        private double _limite;
        private Cliente _titular;

        public double Saldo
        {
            get { return _saldo + _limite; }
            set { _saldo = value; }
        }

        //deposita e saca...

        public Cliente Titular
        {
            get { return _titular; }
            set { _titular = value; }
        }
}
```

O código acima nem possibilita a chamada da propriedade Limite, ele não existe. E nem deve existir enquanto não houver essa necessidade. A propriedade Saldo não devolve simplesmente o saldo... e sim o que queremos que seja mostrado como se fosse o saldo. Utilizar propriedades não só ajuda você a proteger seus atributos, como também possibilita ter de mudar algo em um só lugar... chamamos isso de encapsulamento, pois esconde a maneira como os objetos guardam seus dados. É uma prática muito importante.

Nossa classe está agora totalmente pronta? Isto é, existe a chance dela ficar com menos dinheiro do que o limite? Pode parecer que não, mas, e se depositarmos um valor negativo na conta? Ficaríamos com menos dinheiro que o permitido, já que não esperávamos por isso. Para nos proteger disso basta mudarmos o método deposita() para que ele verifique se o valor é necessariamente positivo.

Depois disso precisaríamos mudar mais algum outro código? A resposta é não, graças ao encapsulamento dos nossos dados.

Cuidado com as propriedades!

Como já dito, não devemos criar propriedades sem um motivo explicito. ...

4.4 – Construtores

Quando usamos a palavra chave new, estamos construindo um objeto. Sempre quando o new é chamado, ele executa o **construtor da classe**. O construtor da classe é um bloco declarado com o **mesmo nome** que a classe:

```
public class Conta
{
        int numero;
        Cliente titular;
        double saldo;
        double limite;

        // construtor
        Conta()
        {
            Console.WriteLine("Construindo uma conta.");
        }
        // ..
}
```

Então, quando fizermos:

```
Conta c = new Conta();
```

A mensagem “construindo uma conta” aparecerá. É como uma rotina de inicialização que é chamada sempre que um novo objeto é criado. Um construtor pode parecer, mas **não é** um método.

O construtor default

Até agora, as nossas classes não possuíam nenhum construtor. Então como é que era possível dar new, se todo new chama um construtor **obrigatoriamente**?

Quando você não declara nenhum construtor na sua classe, o C# cria um para você. Esse construtor é o **construtor default**, ele não recebe nenhum argumento e o corpo dele é vazio.

A partir do momento que você declara um construtor, o construtor default não é mais fornecido.

O interessante é que um construtor pode receber um argumento, podendo assim inicializar algum tipo de informação:

```
public class Conta
{
        int numero;
        Cliente titular;
        double saldo;
        double limite;
        // construtor
        Conta(Cliente titular)
        {
            this.titular = titular;
        }
        // ..
}
```

Esse construtor recebe o titular da conta. Assim, quando criarmos uma conta, ela já terá um determinado titular.

```
Cliente cliente = new Cliente();
cliente.nome = "Carlos";
Conta c = new Conta(cliente);
Console.WriteLine(c.titular.nome);
```

4.5 - A necessidade de um construtor

Tudo estava funcionando até agora. Para que utilizamos um construtor?

A idéia é bem simples. Se toda conta precisa de um titular, como obrigar todos os objetos que forem criados a ter um valor desse tipo? Basta criar um único construtor que recebe essa String!

O construtor se resume a isso! Dar possibilidades ou obrigar o usuário de uma classe a passar argumentos para o objeto durante o processo de criação do mesmo.

Por exemplo, não podemos abrir um arquivo para leitura sem dizer qual é o nome do arquivo que desejamos ler! Portanto, nada mais natural que passar uma String representando o nome de um arquivo na hora de criar um objeto do tipo de leitura de arquivo, e que isso seja obrigatório.

Você pode ter mais de um construtor na sua classe e, no momento do new, o construtor apropriado será escolhido.

Construtor: um método especial?

Um construtor não é um método. Algumas pessoas o chamam de um método especial, mas definitivamente não é, já que não possui retorno e só é chamado durante a construção do objeto.

Chamando outro construtor

Um construtor só pode rodar durante a construção do objeto, isto é, você nunca conseguirá chamar o construtor em um objeto já construído. Porém, durante a construção de um objeto, você pode fazer com que um construtor chame outro, para não ter de ficar copiando e colando:

```
public class Conta
{
        int numero;
        Cliente titular;
        double saldo;
        double limite;
```

```
        // construtor
        Conta(Cliente titular)
        {
            // faz mais uma série de inicializações e configurações
            this.titular = titular;
        }
        public Conta(int numero, Cliente titular)
        {
            // chama o construtor que foi declarado acima
            this(titular);
            this.numero = numero;
        }
        //..
}
```

Existe um outro motivo, o outro lado dos construtores: facilidade. Às vezes, criamos um construtor que recebe diversos argumentos para não obrigar o usuário de uma classe a chamar diversos métodos do tipo 'set'.

No nosso exemplo do CPF, podemos forçar que a classe Cliente receba no mínimo o CPF, dessa maneira um Cliente já será construído e com um CPF válido.

4.6 - Atributos de classe

Nosso banco também quer controlar a quantidade de contas existentes no sistema. Como poderíamos fazer isto? A idéia mais simples:

```
Conta c = new Conta();
totalDeContas = totalDeContas + 1;
```

Aqui, voltamos em um problema parecido com o da validação de CPF. Estamos espalhando um código por toda aplicação, e quem garante que vamos conseguir lembrar de incrementar a variável totalDeContas toda vez?

Tentamos então, passar para a seguinte proposta:

```
class Conta
{
        private int totalDeContas;
        //...
        Conta()
        {
            this.totalDeContas = this.totalDeContas + 1;
        }
}
```

Quando criarmos duas contas, qual será o valor do totalDeContas de cada uma delas? Vai ser 1. Pois cada uma tem essa variável. **O atributo é de cada objeto**.

Seria interessante então, que essa variável fosse **única**, compartilhada por todos os objetos dessa classe.

Dessa maneira, quando mudasse através de um objeto, o outro enxergaria o mesmo valor. Para fazer isso em C#, declaramos a variável como static.

```
private static int totalDeContas;
```

Quando declaramos um atributo como static, ele passa a não ser mais um atributo de cada objeto, e sim um **atributo da classe**, a informação fica guardada pela classe, não é mais individual para cada objeto.

Para acessarmos um atributo estático, não usamos a palavra chave this, mas sim o nome da classe:

```
public class Conta
{
        private static int totalDeContas;
        //...
        public Conta()
        {
            Conta.totalDeContas = Conta.totalDeContas + 1;
        }
}
```

Já que o atributo é privado, como podemos acessar essa informação a partir de outra classe? Precisamos de um getter para ele!

```
public class Conta
{
        private static int totalDeContas;
        //...
        public Conta()
        {
            Conta.totalDeContas = Conta.totalDeContas + 1;
        }
        public int BuscaTotalDeContasCriadas
        {
            get { return Conta.totalDeContas; }
        }
}
```

Como fazemos então para saber quantas contas foram criadas?

```
Conta c = new Conta();
int total = c.BuscaTotalDeContas;
```

Precisamos criar uma conta antes de chamar o método! Isso não é legal, pois gostaríamos de saber quantas contas existem sem precisar ter acesso a um objeto conta. A idéia aqui é a mesma, transformar esse método que todo objeto conta tem em um método de toda a classe. Usamos a palavra static de novo, mudando o método anterior.

```
public static int BuscaTotalDeContasCriadas
{
      Get { return Conta.totalDeContas; }
}
```

Para acessar esse novo método:

```
int total = Conta.GetTotalDeContas();
```

Repare que estamos chamando um método não com uma referência para uma Conta, e sim usando o nome da classe.

Métodos e atributos estáticos

Métodos e atributos estáticos só podem acessar outros métodos e atributos estáticos da mesma classe, o que faz todo sentido já que dentro de um método estático não temos acesso a referência "this", pois um método estático é chamado através da classe, e não de um objeto.

O static realmente traz um "cheiro" procedural, porém em muitas vezes é necessário.

4.7 - Um pouco mais...

1) Em algumas empresas, o UML é amplamente utilizado. Às vezes, o programador recebe o UML já pronto, completo, e só deve preencher a implementação, devendo seguir à risca o UML. O que você acha dessa prática? Vantagens e desvantagens.

2) Se uma classe só tem atributos e métodos estáticos, que conclusões podemos tirar? O que lhe parece um método estático?

4.8 – Exercícios

1) Adicione o modificador de visibilidade (private, se necessário) para cada atributo e método da classe Funcionario. Tente criar um Funcionario no Main e modificar ou ler um de seus atributos privados. O que acontece?

2) Crie as propriedades getters e setters necessários da sua classe Funcionario. Por exemplo:

```
public class Funcionario
{
        private double _salario;

        public double Salario
        {
            get { return this._salario; }
```

```
            set { this._salario = value; }
        }
}
```

3) Modifique suas classes que acessam e modificam atributos de um Funcionario para utilizar as propriedades getters e setters.

Por exemplo:

```
f.salario = 100;
Console.WriteLine(f.salario);
```

Passar para:

```
f.Salario = 100;
Console.WriteLine(f.Salario);
```

4) Faça com que sua classe Funcionario possa receber, opcionalmente, o nome do Funcionario durante a criação do objeto. Utilize construtores para obter esse resultado.

Dica: utilize um construtor sem argumentos também, para o caso de a pessoa não querer passar o nome do Funcionario.

Seria algo mais ou menos assim:

```
public class Funcionario
{
        public Funcionario()
        {
            // construtor sem argumentos
        }
        public Funcionario(String nome)
        {
            // construtor que recebe o nome
        }
}
```

Por que você precisa do construtor sem argumentos para que a passagem do nome seja opcional?

5) (opcional) Adicione um atributo na classe Funcionario de tipo int que se chama identificador. Esse identificador deve ter um valor único para cada instância do tipo Funcionario. O primeiro Funcionario instanciado tem identificador 1, o segundo 2, e assim por diante. Você deve utilizar os recursos aprendidos aqui para resolver esse problema. Crie um getter para o identificador. Devemos ter um setter?

6) (opcional) Crie as propriedades getters e setters da sua classe Empresa e coloque seus atributos como private. Lembre-se de que não necessariamente todos os atributos devem ter getters e setters. Por exemplo, na classe Empresa, seria interessante ter um setter e getter para a sua array de funcionários? Não seria mais interessante ter um método como este?

```
public class Empresa
{
        // ...
        public Funcionario BuscaFuncionario(int posicao)
        {
            return this.funcionarios[posicao];
        }
}
```

7) (opcional) Na classe Empresa, em vez de criar um array de tamanho fixo, receba como parâmetro no construtor o tamanho do array de Funcionario. Agora, com esse construtor, o que acontece se tentarmos dar new Empresa() sem passar argumento algum? Por quê?

8) (opcional) Como garantir que datas como 31/2/2005 não sejam aceitas pela sua classe Data?

9) (opcional) Crie a classe PessoaFisica. Queremos ter a garantia de que pessoa física alguma tenha CPF invalido, nem seja criada PessoaFisica sem cpf inicial. (você não precisa escrever o algoritmo de validação de cpf, basta passar o cpf por um método valida(String x)....)

4.9 – Desafios

1) Porque esse código não compila?

```
class Teste
{
        int x = 37;
        static void Main(String[] args)
        {
            Console.WriteLine(x);
        }
}
```

2) Imagine que tenha uma classe FabricaDeCarro e quero garantir que só existe um objeto desse tipo em toda a memória. Não existe uma palavra chave especial para isto em C#, então teremos de fazer nossa classe de tal maneira que ela respeite essa nossa necessidade. Como fazer isso? (pesquise: singleton design pattern)