



# CEUB

EDUCAÇÃO SUPERIOR

## Programação Orientada a Objetos

ceub.br

# Programação Orientada a Objetos

## Aula 05 – Encapsulamento e Modificadores de Acesso em C#

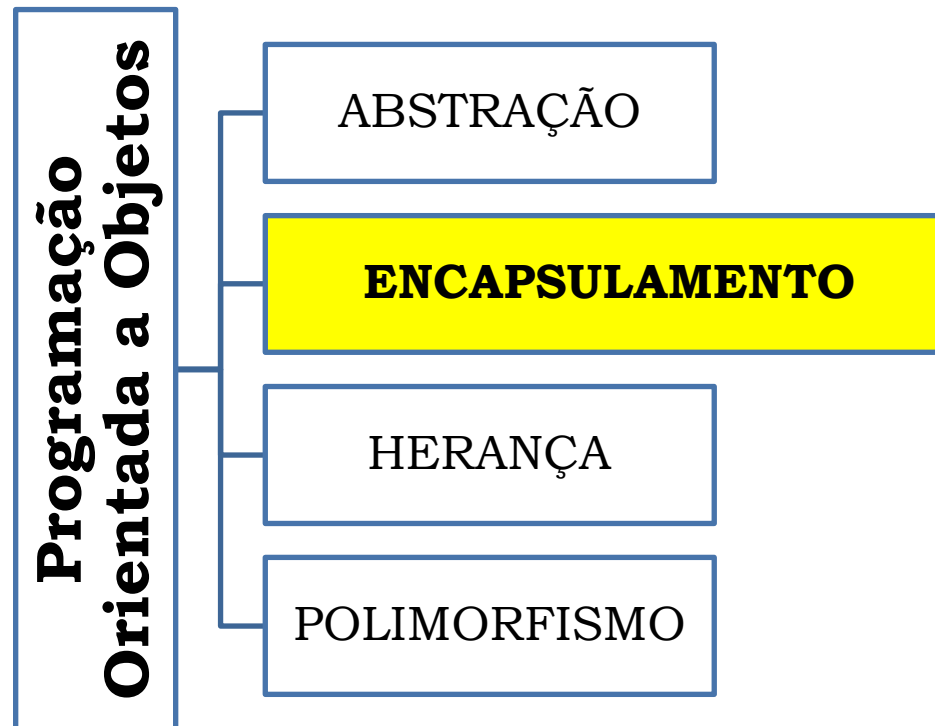
## Agenda

**Encapsulamento**  
**Modificadores de Acesso**  
**Métodos Getters e Setters**



## Os 4 pilares da Programação Orientada a Objetos

Para que uma linguagem possa ser enquadrada no paradigma de orientação a objetos, ela deve atender a **quatro tópicos** bastante importantes:



## Os 4 pilares da Programação Orientada a Objetos

### Encapsulamento

O *encapsulamento* é uma das principais técnicas que define a programação orientada a objetos.

Se trata de um dos elementos que adicionam segurança à aplicação em uma programação orientada a objetos pelo fato de **esconder** as **propriedades**, criando uma espécie de caixa preta.

Essa atitude evita o acesso direto a propriedade do objeto, adicionando uma outra camada de segurança à aplicação.

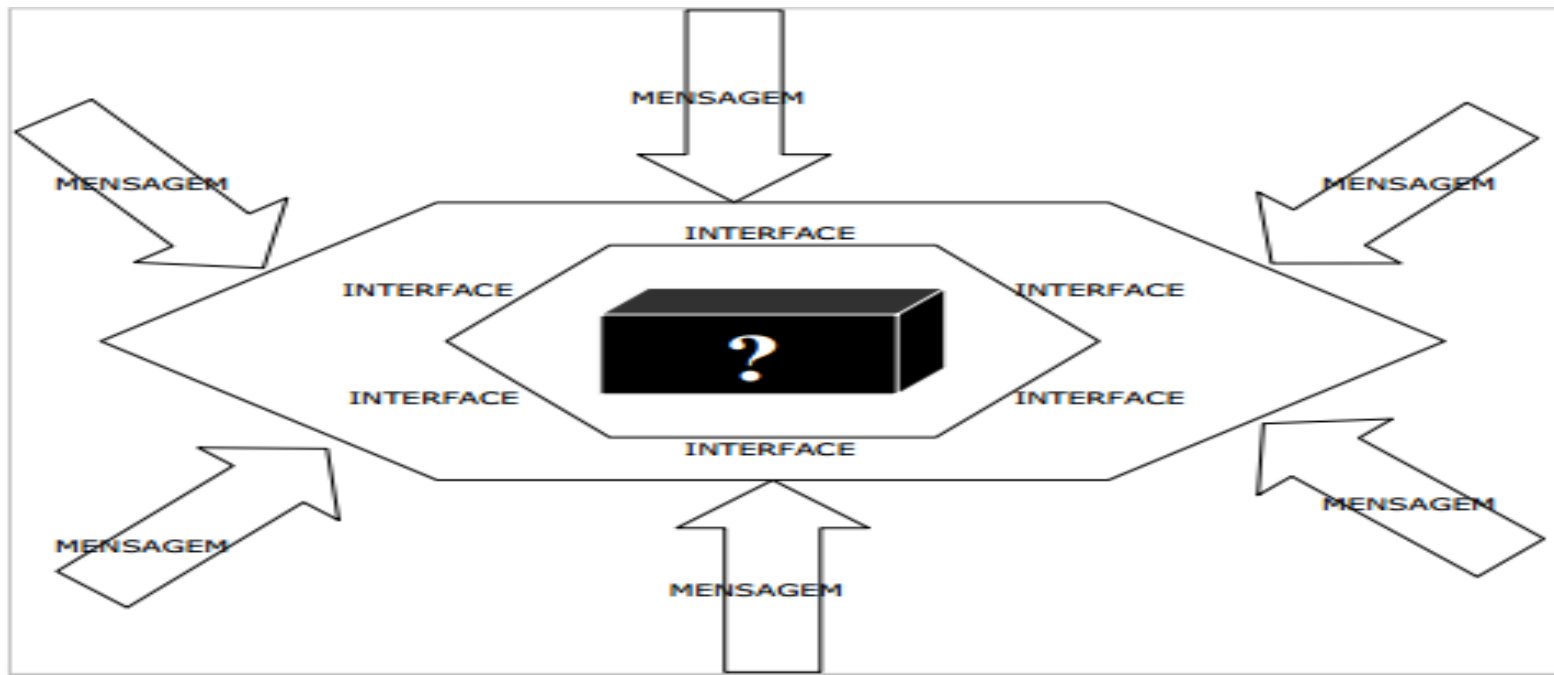
## Os 4 pilares da Programação Orientada a Objetos

### Encapsulamento

- Para fazermos um paralelo com o que vemos no mundo real, temos o encapsulamento em outros elementos.
- Por exemplo, quando clicamos no botão ligar da televisão, não sabemos o que está acontecendo internamente.
- Podemos então dizer que os métodos que ligam a televisão estão encapsulados.

## Encapsulamento

O **encapsulamento** permite a **visualização** de uma **classe** como uma caixa preta. Neste caso, sabe-se o que a classe faz, sem ter acesso ao seu comportamento interno, possibilitando esconder os detalhes da implementação realizada.



## Encapsulamento

- **Objetivo:**
  - **Controlar o acesso de atributos e métodos de um objeto, através de uma interface bem definida.**
- **Benefícios:**
  - **Manutenção de software;**
  - **Evolução de software;**



## Encapsulamento

- **Exemplo:**
  - **Motor de um automóvel.**
  - **O motorista não precisa ter conhecimento técnico de como funciona cada parte do motor, mas apenas saber qual é a sua finalidade e como usá-lo.**

## Encapsulamento

### **Vantagens:**

- **Proteger os atributos do objeto quanto à manipulação por outros objetos (proteção contra acesso não autorizado, valores inconsistentes, entre outras possibilidades).**
- **Esconder a estrutura interna do objeto de modo que a interação com este objeto seja relativamente simples e, à medida do possível, siga um padrão de desenvolvimento que facilite o entendimento dos programadores que o utilizem.**

## Abstração de dados e encapsulamento

- **As classes, normalmente, ocultam os detalhes de implementação dos seus usuários. Isso se chama ocultamento de informações.**
- **Exemplo:**
  - **O motorista de um veículo ao fazer uso do motor do carro está usando o motor para se locomover, porém não precisa saber dos seus detalhes de funcionamento.**

## Abstração de dados e encapsulamento

- Nesse exemplo, o cliente se preocupa com a funcionalidade que o motor oferece, mas não como essa funcionalidade é implementada.
- Esse conceito é conhecido como **abstração de dados**.
- A Programação Orientada a Objetos (POO) tem como principais atividades a criação de tipos e a expressão de interações entre objetos desses tipos.
- Essa atividade está diretamente associada à noção de tipo abstrato de dados (ADT abstract data type), que melhora o processo de desenvolvimento de programas, pois permite mais flexibilidade ao programador na criação de novos tipos de dados.

## Abstração de dados e encapsulamento

- **Assim, pode-se afirmar que um ADT captura duas noções: representação de dados e operações que podem ser realizadas nesses dados.**
- **Linguagens como Java, C# e outras linguagens utilizam classes para implementar tipos abstratos de dados.**

## Encapsulamento em C#

- O encapsulamento em C# ocorre nas classes.
- Quando o programador cria uma classe, ele especifica o Código e os dados que irão formar essa classe.
- Estes elementos serão chamados de membros da classe.
- O comportamento e a interface de uma classe são definidos pelos métodos que operam nas instâncias de dados.
- O encapsulamento em C# é implementado através dos seus modificadores de acesso público, protegido, privado e implícito (protect, private e public).

## Encapsulamento em C#

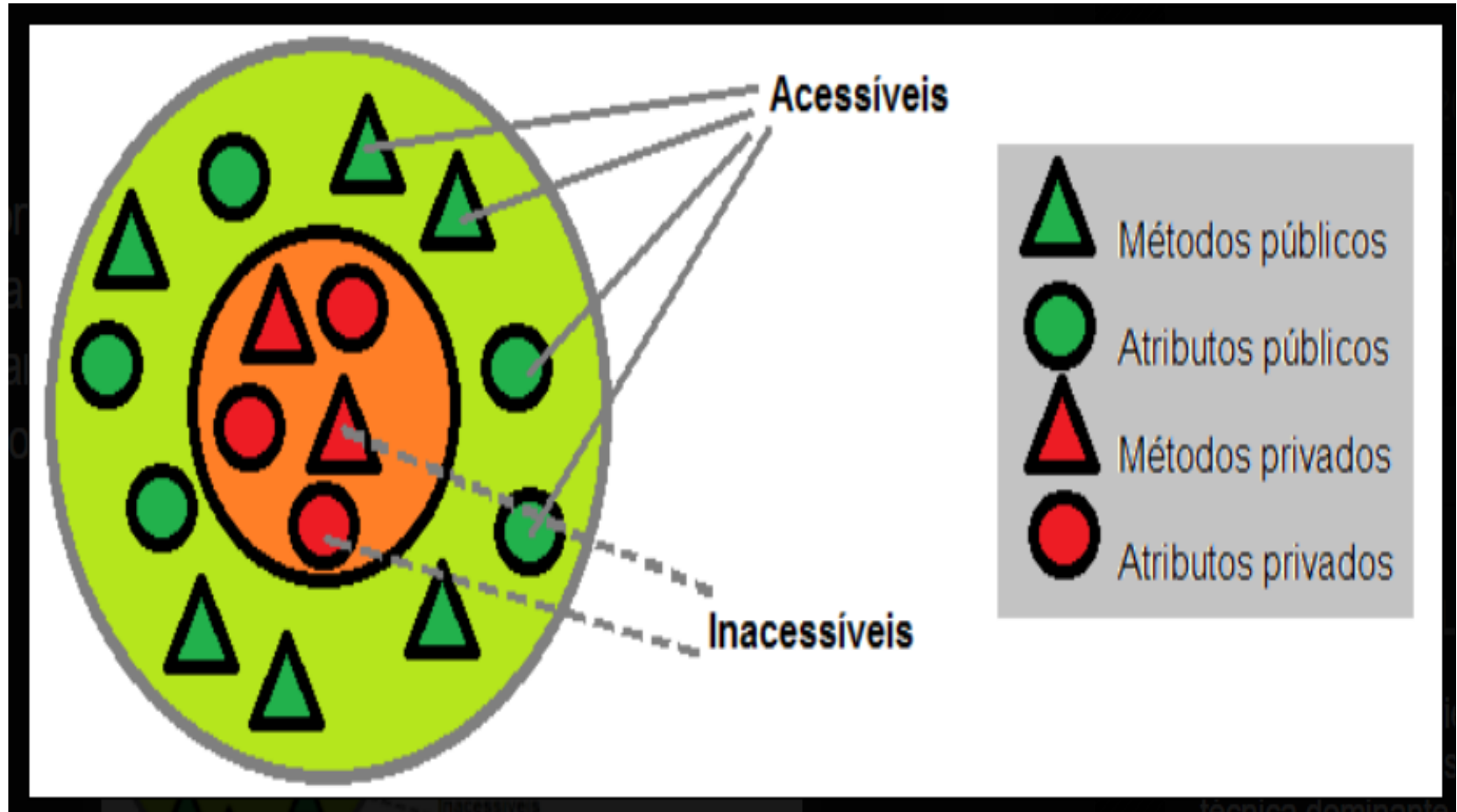
- **Considerando que o objetivo de uma classe é encapsular a complexidade, existem mecanismos para ocultar a complexidade da implementação que está dentro da classe.**
- **Cada método ou variável em uma classe pode ser definida como pública, privada ou protegida.**
- **A interface de uma classe possibilita que todos os usuários externos possam acessar livremente os dados da classe que os métodos públicos permitem.**
- **Já os métodos privados estabelecem que os dados somente podem ser acessados pelos métodos que são membros da classe.**

## Encapsulamento em C#

- **Considerando que os membros privados de uma classe só podem ser acessados por outras partes do programa através dos métodos públicos desta classe, o programador em C# pode fazer uso do encapsulamento para garantir que ações inapropriadas ou imprevistas não ocorram.**
- **Assim, o programador em C# deve ser bastante cuidadoso ao definir a interface pública de uma classe para não expor demasiadamente o funcionamento da classe.**



## Encapsulamento em C#



## Modificadores de acesso em C#

**A visibilidade de uma classe e de seus membros (atributos, propriedades, métodos) também podem ser chamadas de modificadores de acesso (qualificadores ou identificadores).**

**Esses modificadores podem ser do tipo:**

- **public,**
- **private,**
- **protected ou**
- **internal.**

## Modificadores de acesso em C#

- **Public:** Com este modificador, o acesso é livre em qualquer lugar do programa.
- **Private:** Com este modificador, o acesso é permitido somente dentro da classe onde ele foi declarado. Por padrão, é a visibilidade definida para métodos e atributos em uma classe.
- **Protected:** Com este modificador, apenas a classe que contém o modificador e os tipos derivados dessa classe (HERANÇA) tem o acesso.
- **Internal:** Com este modificador, o acesso é limitado apenas ao assembly atual.
- **Protected Internal:** Com este modificador, o acesso é limitado ao assembly atual e aos tipos derivados da classe que contém o modificador.

## Modificadores de acesso em C#

- O encapsulamento relaciona os dados (atributos) com o código (métodos) que os manipula.
- O encapsulamento também fornece outro recurso importante que é o controle de acesso.
- Através dos modificadores de acesso, os programadores podem controlar o acesso aos membros de uma classe.
- É através desse controle que o programador garante que não haverá um uso indesejado dos dados de uma determinada classe.
- Normalmente, uma classe é criada como uma espécie de caixa preta, que pode ser usada, porém, somente através dos seus métodos públicos que foram colocados à disposição.

## Modificadores de acesso em C#

- O modificador de acesso é uma instrução que define como um membro de uma classe poderá ser acessado.
- C# possui um rico conjunto destes modificadores.
- Alguns aspectos do controle de acesso estão relacionados à herança e ao conceito de pacotes.
- C# possui os seguintes modificadores de acesso: `public`, `private` e `protected`.

## Modificadores de acesso em C#

- Modificador de acesso ***public***
  - Este modificador permite que o **membro público** seja acessado por qualquer outro código do programa.
  - O modificador de acesso ***public*** é o *mais liberal* e que, portanto, exige maior responsabilidade do programador ao empregá-lo.

## Modificadores de acesso em C#

- Modificador de acesso **private**
  - Este modificador determina que o membro privado só pode ser acessado por métodos de dentro da própria classe.
  - O modificador de acesso **private** é o mais restritivo e que deve ser empregado sempre que possível.

## MODIFICADORES DE ACESSO

Imagine uma classe **ContaBancaria** que não tem um modificador de acesso;

Nesta classe, por mais que modifiquemos o atributo **saldo** através do método **sacar**, é possível atribuir qualquer valor ao atributo **saldo**.

```
class ContaBancaria {  
  
    public int Numero { get; private set; }  
    public string Titular { get; set; }  
    public double Saldo { get; private set; }  
    public ContaBancaria(int numero, string titular) {  
        Numero = numero;  
        Titular = titular;  
    }  
    public ContaBancaria(int numero, string titular, double saldo) : this(numero, titular) {  
        Saldo = saldo;  
    }  
}
```

```
    public void Deposito(double quantia) {  
        Saldo += quantia;  
    }  
    public void Saque(double quantia) {  
        Saldo -= quantia + 5.0;  
    }  
    public override string ToString() {  
        return "Conta "  
            + Numero  
            + ", Titular: "  
            + Titular  
            + ", Saldo: $ "  
            + Saldo.ToString("F2", CultureInfo.InvariantCulture);  
    }  
}
```



## MODIFICADORES DE ACESSO

**Isso não é conveniente, pois pode ser que um valor inválido seja atribuído ao valor do saldo.**

**A melhor forma de resolver isso seria forçar quem usa a classe ContaCorrente a chamar sempre o método sacar e não permitir que seja atribuído um valor diretamente ao atributo saldo.**

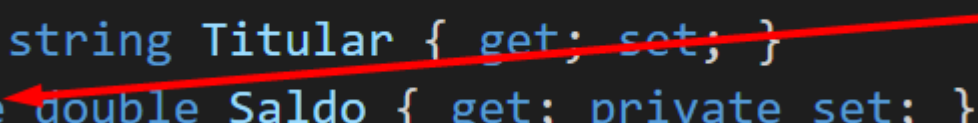
**Para fazer isso em C#, basta declarar que os atributos não podem ser acessados fora da classe usando a palavra chave `private`.**

## MODIFICADORES DE ACESSO

**A melhor forma de resolver isso seria forçar quem usa a classe a chamar sempre o método sacar e não permitir que seja atribuído um valor diretamente ao atributo saldo.**

**Para fazer isso em C#, basta declarar que os atributos não podem ser acessados fora da classe usando a palavra chave `private`.**

```
public int Numero { get; private set; }  
public string Titular { get; set; }  
private double Saldo { get; private set; }
```



## MODIFICADORES DE ACESSO

```
public int Numero { get; private set; }  
public string Titular { get; set; }  
private double Saldo { get; private set; }
```

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

Marcando um atributo como privado, fechamos o acesso a ele a partir de outras classes.

É uma prática quase que **obrigatória** proteger os atributos de suas classes com o **private**

Quem chama o método **sacar** não precisa saber que o saldo está sendo checado. Quem for usar essa classe, basta saber *o que o método faz*, e não *como ele faz*.

## MODIFICADORES DE ACESSO

```
class ContaBancaria {  
  
    public int Numero { get; private set; }  
    public string Titular { get; set; }  
    private double Saldo { get; private set; }  
    public ContaBancaria(int numero, string titular) {  
        Numero = numero;  
        Titular = titular;  
    }  
    public ContaBancaria(int numero, string titular, double saldo) : this(numero, titular) {  
        Saldo = saldo;  
    }  
    public void Deposito(double quantia) {  
        Saldo += quantia;  
    }  
    public void Saque(double quantia) {  
        Saldo -= quantia + 5.0;  
    }  
}
```

## MODIFICADORES DE ACESSO

**A palavra chave `private` também pode ser utilizada para modificar o acesso a um método.**

- **Isso é usado quando existe um método apenas auxiliar à própria classe, e não queremos que outros o enxerguem.**

**Há também o modificador `public`, que permite a todos acessarem um determinado atributo ou método**

**É muito comum que atributos sejam `private` e quase todos os métodos sejam `públic` (não é uma regra)**

- **Assim, toda conversa de um objeto com outro é feita através de troca de mensagem (acessando seus métodos)**

## Métodos Getters e Setters

### Implementação Manual – Não usual em C#

Para permitir o acesso aos atributos (já que eles são `private`) de uma maneira controlada, a prática mais comum é criar dois métodos

Um que retorna o valor **(get)**

E outro que muda o valor **(set)**

O padrão para esses métodos é colocar a palavra **get** ou **set** antes do nome do atributo

O padrão do método `get` não vale para variáveis do tipo boolean

- Esses atributos são acessados via `is` e `set`
- Exemplo: para verificar se uma lâmpada está acesa,
  - seriam criados os métodos `isLigado` e `setLigado`

## MÉTODOS GETTERS E SETTERS

### Exemplo:

Métodos getters	Métodos setters
<pre>public String getNome() {     return nome; }</pre>	<pre>public void setNome(String nome) {     this.nome = nome; }</pre>
<pre>public double getSalario() {     return salario; }</pre>	<pre>public void setSalario(double salario) {     this.salario = salario; }</pre>

## MÉTODOS GETTERS E SETTERS

### Exemplo:

Incluindo o modificador **private** nos atributos para uma classe **Cliente**.

```
public class Cliente {  
  
    private String nome;  
    private String endereco;  
    private float renda;  
    private String profissao;  
  
}
```



## MÉTODOS GETTERS E SETTERS

### Exemplo:

**E então, incluindo os métodos get para cada um dos atributos:**

```
//..
```

```
public String getNome() {  
    return nome;  
}
```

```
public String getEndereco() {  
    return endereco;  
}
```

```
public float getRenda() {  
    return renda;  
}  
public String getProfissao() {  
    return profissao;  
}
```

## MÉTODOS GETTERS E SETTERS

### ○ Exemplo:

- E também o método set:

```
//..  
public void setNome(String novoNome) {  
    nome = novoNome;  
}  
  
public void setEndereco(String novoEndereco) {  
    endereco = novoEndereco;  
}  
  
public void setRenda(float novaRenda) {  
    renda = novaRenda;  
}  
  
public void setProfissao(String novaProfissao) {  
    profissao = novaProfissao;  
}
```

## MÉTODOS GETTERS E SETTERS

### Exemplo:

```
public class ContaCorrente {  
    private float saldo;  
    private Cliente clienteConta;  
  
    public float getSaldo() {  
        return saldo;  
    }  
  
    public void setSaldo(float novoSaldo) {  
        saldo = novoSaldo;  
    }  
  
    public Cliente getClienteConta() {  
        return clienteConta;  
    }  
  
    public void setClienteConta(Cliente novoCliente) {  
        clienteConta = novoCliente;  
    }  
}
```

## MÉTODOS GETTERS E SETTERS

### Properties

## MÉTODOS GETTERS E SETTERS

### Properties

**As propriedades também são responsáveis pela encapsulamento dos atributos. São definições de métodos encapsulados, porém expondo uma sintaxe similar à de atributos e não de métodos**

**Uma propriedade é um membro que oferece um mecanismo flexível para ler, gravar ou calcular o valor de um campo particular.**

**As propriedades podem ser usadas como se fossem atributos públicos, mas na verdade elas são métodos especiais chamados "acessadores".**

**Isso permite que os dados sejam acessados facilmente e ainda ajuda a promover a segurança e a flexibilidade dos métodos.**

## MÉTODOS GETTERS E SETTERS

### Properties Exemplo

```
public class Pessoa
{
    //definição dos atributos
    private string _nome;
    private int _idade;
    //definição das propriedades
    public string Nome
    {
        get { return _nome; }
        set { _nome = value; }
    }
    public int Idade
    {
        get { return _idade; }
        set {
            if (value > 120 || value < 0)
                _idade = 0;
            else
                _idade = value;
        }
    }
}
```

```
//definição dos métodos
public string VerificarIdade()
{
    string msg;
    if (Idade > 18)
        msg = "Já pode Dirigir";
    else
        msg = "Não pode Dirigir";
    return msg;
}
```

## Auto Properties

**Auto Properties (ou propriedades automáticas) em C# são uma sintaxe abreviada para declarar uma propriedade com um campo de suporte implícito.**

**Com auto properties, você não precisa criar explicitamente um campo de classe separado para armazenar o valor da propriedade.**

**Neste caso, o compilador cria um campo de classe anônimo automaticamente, que é usado para armazenar o valor da propriedade.**

## Auto Properties

**Em C#, é possível declarar e inicializar um campo de classe diretamente na declaração da propriedade, usando a sintaxe "propriedade auto-implementada".**

**Essa sintaxe abreviada permite que você defina uma propriedade com um campo de suporte implícito, sem precisar criar explicitamente um campo de classe separado.**

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

**OBS: Em Java e Python, ainda é necessário declarar explicitamente os campos de classe antes de usá-los em propriedades. A propriedade em si é apenas uma abstração de um ou mais métodos que leem ou gravam o campo de classe.**



## Auto Properties

É uma forma simplificada de se declarar propriedades que não necessitam lógicas particulares para as operações get e set. Para gerar, selecione os atributos e com botão direito selecione opções rápidas e refatorações, encapsular campo, conforme descrito abaixo.

```
public class Pessoa
```

```
{  
    //definição dos atributos
```


```
    private string _nome;
```

```
    private int _idade;
```

```
0 referências
```

```
    public string Nome { get => _nome; set => _nome = value; }
```

```
1 referência
```



Ações Rápidas e Refatorações...	Ctrl+.
Renomear...	Ctrl+R, Ctrl+R
Remover e Classificar Usos	Ctrl+R, Ctrl+G
Exibir Código	F7

```
12 private int _idade;
```



Usar a propriedade auto	...
Gerar construtor "Pessoa(int)"	...
Gerar Equals(object)	...
Gerar Equals e GetHashCode	...
Encapsular campo: "_idade" (e usar propriedade)	...
Encapsular campo: "_idade" (mas ainda usar o campo)	Visualizar alterações
Suprimir ou Configurar os problemas	...

```
21 if (Idade > 18)  
22     msg = "Já pode Dirigir";
```

```
public class Pessoa
```

```
{
```

```
    //definição dos atributos
```

```
    private string _nome;
```

```
    private int _idade;
```

```
    public string Nome { get => _nome; set => _nome = value; }
```

```
    public int Idade { get => _idade; set => _idade = value; }
```

```
    //definição dos métodos
```

```
    public string VerificarIdade()
```

```
{
```

```
        string msg;
```

```
        if (Idade > 18)
```

```
            msg = "Já pode Dirigir";
```

```
        else
```

```
            msg = "Não pode Dirigir";
```

```
        return msg;
```

```
}
```

```
}
```

## EXERCÍCIO

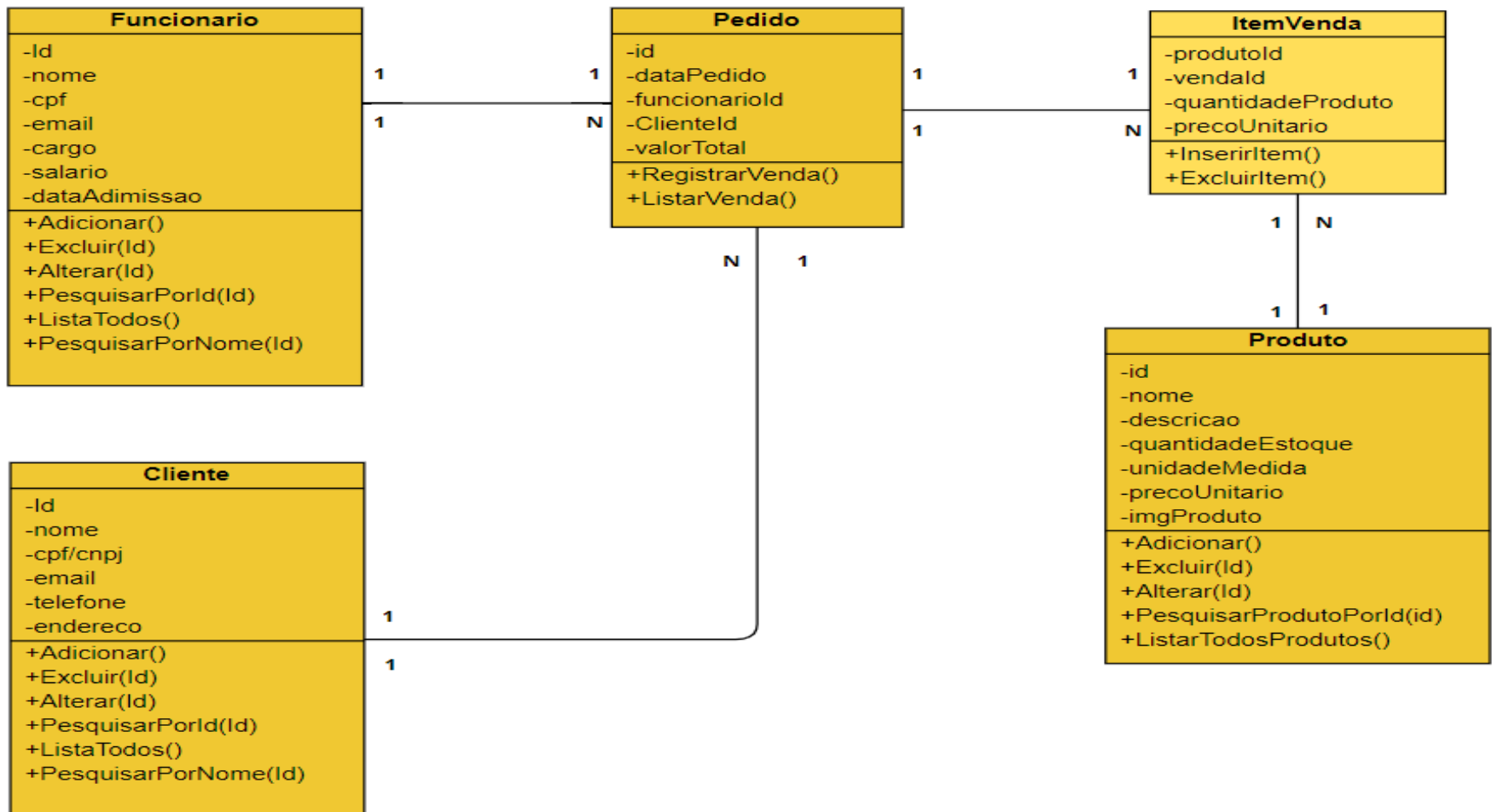
**1 - Crie um atributo numero na classe ContaCorrente. Defina o atributo numero como private. Crie seus métodos get e set.**

**2- Crie um classe Funcionário que tenha os seguintes métodos privados : nome, dt\_nascimento, salario, n\_filhos, área\_atuação. Crie seus métodos get e set.**

**3-Crie um classe teste funcionário que chame os elementos getters e setters da classe funcionário.**

## 2 - EXERCÍCIO

Crie um projeto e implemente as suas classes usando os princípios da abstração e encapsulamento, de acordo com o diagrama abaixo



## REFERÊNCIAS

<http://www.hardware.com.br/artigos/programacao-orientada-objetos/>

<http://www.fontes.pro.br/educacional/materialpaginas/C#/arquivos/jdbc/jdbc.php>

<http://www.dm.ufscar.br/~waldeck/curso/C#>

**PORTAL EDUCAÇÃO - Cursos Online : Mais de 900 cursos online com certificado** <http://www.portaleducacao.com.br/informatica/artigos/7852/moderadores-de-acesso#ixzz2AAmxO3JD>

<http://www.slideshare.net/regispires/C#-08-modificadores-acesso-e-membros-de-classe-presentation>

<https://www.devmedia.com.br/abstracao-encapsulamento-e-heranca-pilares-da-poo-em-C#/26366>