Avelar Brotero
ESCOLA SECUNDÁRIA - COIMBRA

Programação e Sistemas de Informação

CURSO PROFISSIONAL TÉCNICO DE
GESTÃO E PROGRAMAÇÃO DE SISTEMAS INFORMÁTICOS

Introdução à Programação Orientada a Objetos

MÓDULO 9

Professor: João Martiniano

Conteúdos abordados neste módulo

- Introdução à programação orientada a objetos
- Conceitos fundamentais
- Diagramas de classe
- Classes e respetivos membros:
  - Campos
  - Constantes
  - Métodos
  - *Access modifiers*
  - Acesso aos dados de uma classe (*getters* e *setters*, propriedades)
- Instanciar uma classe
- Construtor de uma classe
- Finalizador (ou destrutor) de uma classe
- *Method overloading*
- Classes e membros estáticos
- *Naming guidelines*

Introdução

- *Object Oriented Programming* (OOP) / Programação Orientada a Objetos (POO)
- A programação orientada a objetos (POO) é um **paradigma** de programação
- Neste paradigma o programador escreve programas e organiza o código com base em **objetos**
- Os objetos possuem **dados** e executam **operações**

O que é um paradigma?

- De uma forma muito simplificada podemos dizer que é um modelo de pensamento
- Existem vários paradigmas e em todas as áreas do conhecimento (economia, matemática, informática, medicina, ciências sociais, etc.)

Introdução

- A Programação Orientada a Objetos não nasceu abruptamente a partir do nada

- Nos primórdios da informática, os programas eram relativamente simples, refletindo os tipos de problemas a que a informática tentava dar resposta
- A evolução foi no sentido de:
  - o custo e o tamanho dos equipamentos informáticos foi decrescendo
  - foram surgindo novos e mais sofisticados equipamentos

- Consequentemente foram sendo criados programas informáticos para novas e diferentes áreas
- Paralelamente, foram sendo criadas novas linguagens de programação

Introdução

- Consequentemente foram sendo criadas aplicações informáticas cada vez maiores e cada vez mais complexas
- Paralelamente ao surgimento de novas linguagens de programação, foram sendo desenvolvidas diferentes técnicas de programação
- A evolução gradual, tanto das linguagens como das técnicas, bem como da crescente complexidade dos problemas, levaram ao surgimento da Programação Orientada a Objetos

- Importa saber que, para diferentes problemas, existem diferentes soluções
- A Programação Orientada a Objetos não é a única ferramenta ao dispor dos programadores nem é a mais adequada a todos os tipos de problemas
- O bom programador deve familiarizar-se com diferentes técnicas, paradigmas e linguagens de modo a atacar os problemas da forma mais eficaz possível

Introdução

- A Programação Orientada a Objetos (POO) pretende modelar o mundo ou um problema, para uma linguagem de computador
- Na POO o conceito central é o de objeto
- O objeto contém **dados** (**atributos**) e **operações** (**métodos**)

- Existem várias linguagens de programação que suportam POO
- Existem contudo, diferenças, na forma como implementam a POO
- Quer isto dizer, que, um programador, ao aprender a programar com objetos numa determinada linguagem, terá de adaptar os conceitos que aprendeu quando programar noutra linguagem

Conceitos fundamentais

- Na programação orientada a objetos existem 4 conceitos fundamentais:
  - abstração (*abstraction*)
  - encapsulamento (*encapsulation*)
  - herança (*inheritance*)
  - polimorfismo (*polymorphism*)

Abstração (*abstraction*)

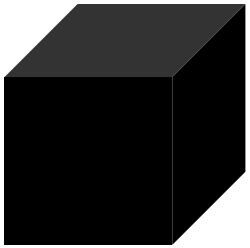
- A abstração consiste em esconder os detalhes e apenas mostrar o que é relevante
- A abstração foca-se no que é exterior ao objeto
- Por exemplo:
  - o acto de ligar um computador envolve muitas e variadas operações
  - vários componentes do computador são envolvidos nesta tarefa e trabalham de forma concertada para atingir este objetivo
  - no entanto, tudo isto é escondido do utilizador
  - para o utilizador basta pressionar um botão e todo o processo é conduzido de forma automática a partir daí

Encapsulamento (*encapsulation*)

- Um objeto contém dados e executa operações
- Estes dois componentes estão contidos no objeto
- O objeto apenas mostra ao exterior aquilo que quiser:
  - numa situação extrema, o objeto pode tornar público todo o seu conteúdo

http://www.modernisticon.com/ktv-clear-transparent-see-through-tv-set-space-age-electronics/

  - numa outra situação extrema todo o conteúdo do objeto está oculto



Encapsulamento (*encapsulation*)

- Ambas as situações anteriores não fazem muito sentido
- A situação ideal consiste em revelar para o exterior apenas os componentes adequados e esconder o funcionamento interno do objeto
- É nisto que consiste o encapsulamento
- O encapsulamento e a abstração são dois conceitos complementares
- Por exemplo:
  - o funcionamento interno de um televisor está escondido do utilizador
  - o televisor partilha alguns dados com o exterior:
    - estado (ligado/desligado), nível de som, canal selecionado, etc.
  - o televisor disponibiliza determinadas operações ao utilizador:
    - ligar/desligar
    - aumentar/diminuir o som
    - selecionar o canal
    - controlo de brilho/contraste/saturação de cor
    - etc.

Herança (*inheritance*)

- Os objetos podem ser organizados hierarquicamente
- Ou seja, um objeto pode ser herdeiro de outro objeto
- Ou seja, é um objeto filho (*child objet*)
- O objeto filho herda os dados e as operações do objeto pai
- Estes podem ser utilizados ou redefinidos (*polimorfismo*)
- Pode também definir os seus próprios dados e operações
- A herança promove a reutilização e extensão de objetos

Polimorfismo (*polymorphism*)

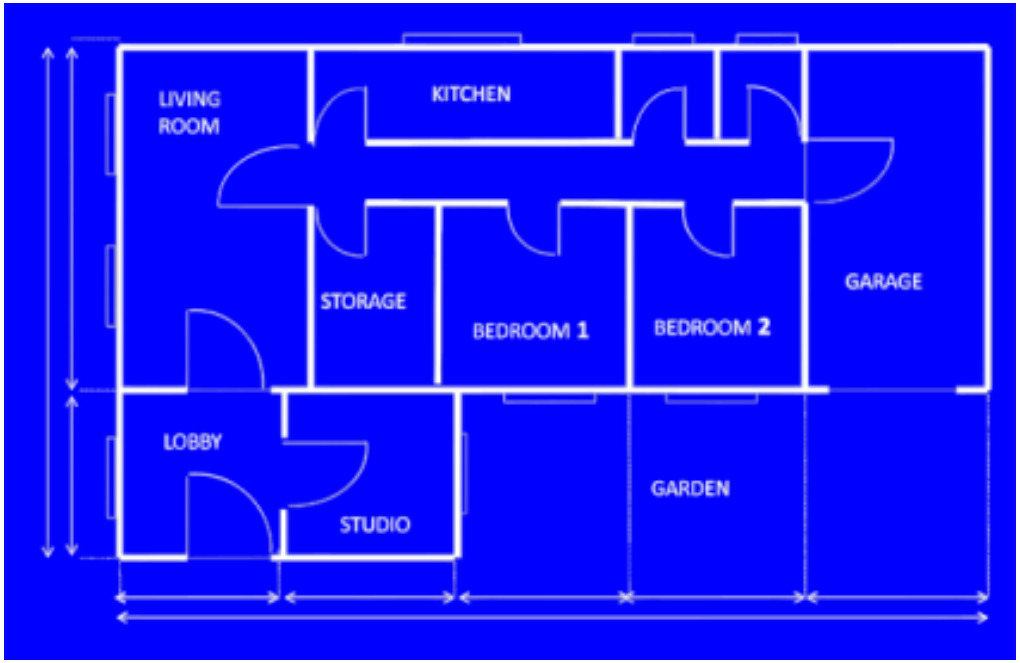
- O conceito de polimorfismo está relacionado com o conceito de herança
- Existem diferentes formas de polimorfismo, mas pode ser entendido da seguinte forma:
  - num conjunto de objetos organizados hierarquicamente (com herança) todos podem ter uma operação com o mesmo nome, mas implementá-la de forma diferente

- Diferentes linguagens de programação implementam o conceito de polimorfismo de diferentes maneiras

Conceitos fundamentais

Classe

- A classe é a definição a partir da qual são gerados objetos concretos
- É como a planta de uma casa: é um modelo a partir do qual são construídas casas



Conceitos fundamentais

Objeto

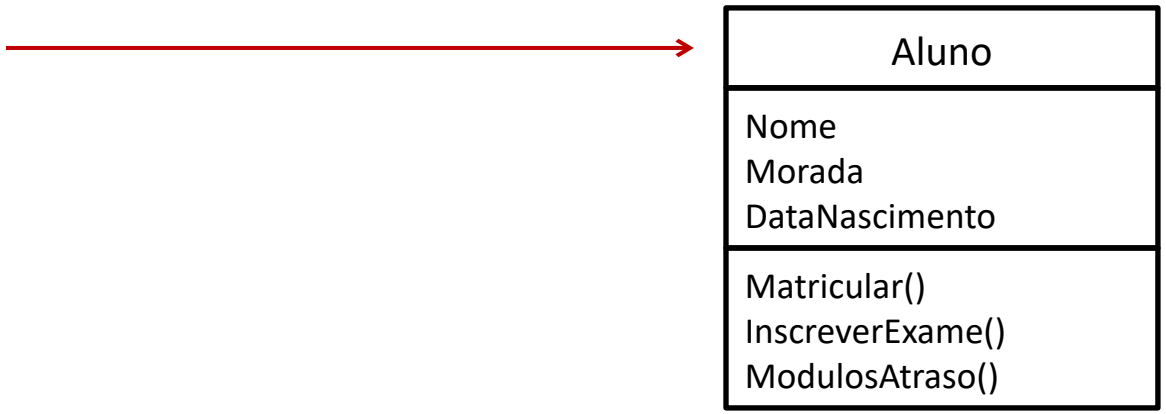
- Um objeto é uma instância de uma classe
- A partir de uma classe podem ser instanciados vários objetos
- Todos os objetos apesar de partilharem as mesmas características fundamentais da sua classe, podem conter informações próprias
- Por exemplo, a partir da mesma planta podem ser construídas casas idênticas:

- Mas as casas também podem ter diferentes caracteristicas, como diferentes cores, portas e janelas diferentes, etc.

DIAGRAMAS DE CLASSE

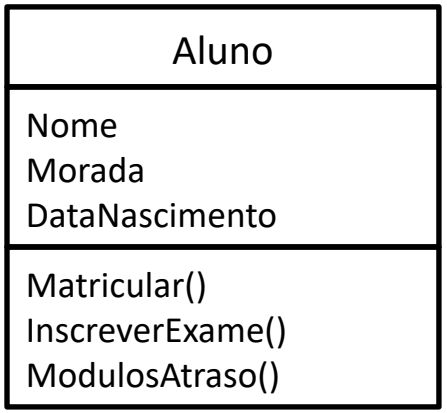
Diagramas de classe

- Os diagramas de classe são uma forma de representar, visualmente:
  - uma classe e os seus componentes
  - a forma como uma classe se relaciona com outras classes
- Um diagrama de classe é composto por 3 partes:
  - o nome da classe



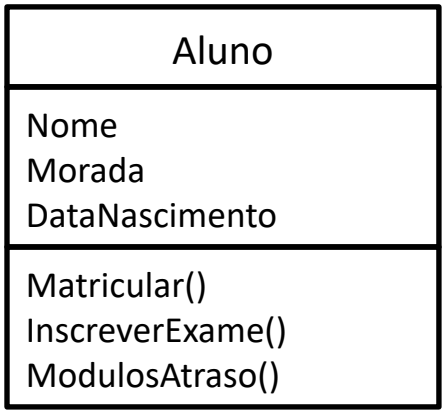
Diagramas de classe

- Os diagramas de classe são uma forma de representar, visualmente:
  - uma classe e os seus componentes
  - a forma como uma classe se relaciona com outras classes
- Um diagrama de classe é composto por 3 partes:
  - o nome da classe
  - os dados da classe: propriedades e campos



Diagramas de classe

- Os diagramas de classe são uma forma de representar, visualmente:
  - uma classe e os seus componentes
  - a forma como uma classe se relaciona com outras classes
- Um diagrama de classe é composto por 3 partes:
  - o nome da classe
  - os dados da classe: propriedades e campos
  - operações da classe: métodos

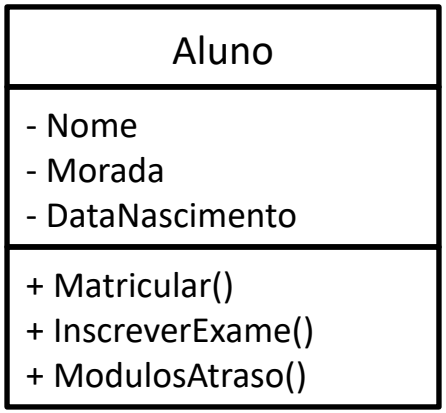


Diagramas de classe: Visibilidade

- Nos diagramas de classe é possível especificar a visibilidade dos membros de uma classe por meio dos seguintes símbolos:

  - `-` private
  - `+` public
  - `#` protected

- Exemplo:

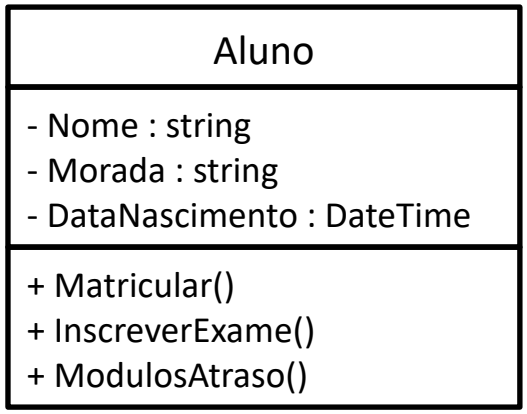


Diagramas de classe: Tipos de campos e propriedades

- Nos diagramas de classes podem-se especificar os tipos de dados dos campos, constantes e propriedades
- Sintaxe:

```
membro : tipo
```

- Exemplo:

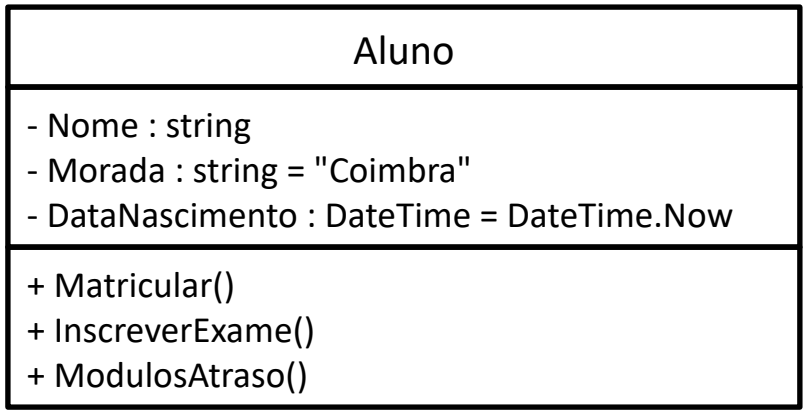


Diagramas de classe: Valores por defeito

- Para cada campo ou propriedade podem ser especificados valores por defeito
- Sintaxe:

```
membro : tipo = valor
```

- Exemplo:

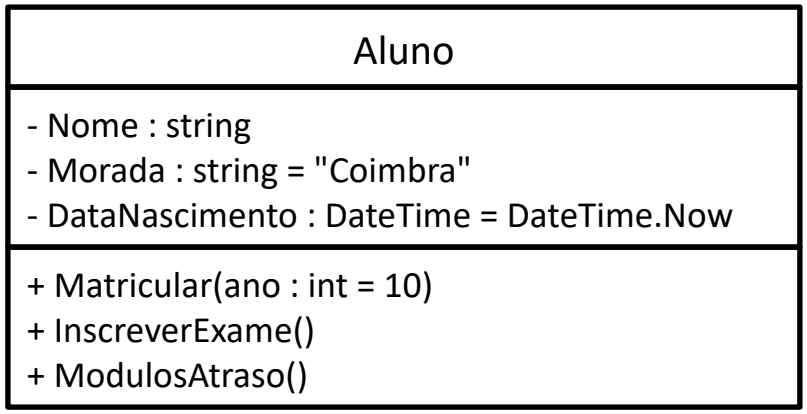


Diagramas de classe: Parâmetros de métodos

- Quanto aos métodos de uma classe, também se podem especificar os parâmetros e os respetivos tipos, bem como valores por defeito
- Sintaxe:

```
parâmetro : tipo [ = valor]
```

- Exemplo:

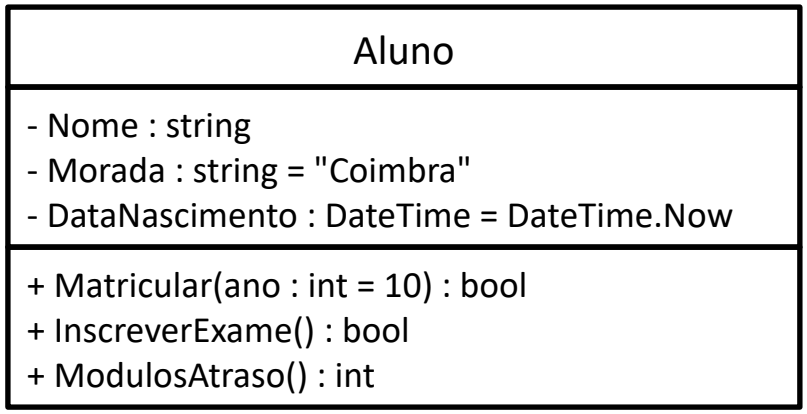


Diagramas de classe: Tipo retornado por um método

- Os diagramas de classe permitem que se especifique o tipo de dados que os métodos retornam
- Sintaxe:

```
método() : tipo
```

- Exemplo:



CLASSE

Classe

- Sintaxe simplificada para definir uma classe em C#:

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

- Em que:

  `access_modifier` (opcional) Modificador de acesso à classe: `public`, `protected`, `private`, etc.

- Exemplo:

```
public class Livro
{

}
```

Classe

- Observações:
  - o *access modifier* `public` não é obrigatório, mas ao utilizá-lo asseguramo-nos de que a classe é pública e pode ser acedida sem constrangimentos pelo restante código
  - existem outros *access modifiers*, os quais serão analisados posteriormente

✔ **Naming Guidelines**	O nome de uma classe deve ser escrito em **PascalCasing**

- Atenção:
  - a sintaxe para definição de uma classe, apresentada na página anterior, está bastante simplificada
  - a linguagem C# oferece mais opções, as quais não são relevantes nesta fase do estudo da programação orientada a objetos

Classe

- Para declarar uma instância de uma classe é utilizada a instrução `new`:

```
Livro livro1 = new Livro();
```

Classe

Exercício

- Crie uma classe chamada `Aluno`

Classe

Exercício: Resolução

- Crie uma classe chamada `Aluno`

```
public class Aluno
{

}
```

Membros de uma classe

- O que compõe uma classe?
- Uma classe é composta por membros, existindo vários tipos de membros (todos opcionais)
- Pegando novamente na sintaxe de uma classe, esta fica com o seguinte formato:

```
[access_modifier] class nome
{
    [constantes]
    [campos]
    [propriedades]
    [construtores]
    [finalizador]
    [métodos]
    [eventos]
    [outros...]
}
```

- Estes componentes são analisados seguidamente

Ordem de colocação dos membros de uma classe

- Apesar de não ser obrigatório, existe a seguinte recomendação quanto à ordem de colocação dos membros de uma classe:
  - constantes
  - campos
  - construtores
  - finalizador
  - eventos
  - enumerações
  - propriedades
  - métodos
  - structs

- Observações:
  - esta lista não especifica todos os membros que uma classe pode conter
  - diferentes programadores seguem diferentes ordens de organização

CAMPOS

Campos

- Uma classe pode conter vários tipos de membros
- Vamos analisar os membros que nos permitem armazenar dados:
  - os campos
  - as propriedades
- Um campo é uma variável dentro de uma classe
- Pode ser de qualquer tipo:
  - tipos predefinidos na linguagem (`bool`, `int`, `char`, etc.)
  - ou um tipo definido pelo programador

Campos

- Sintaxe simplificada para declarar um campo:

```
[access_modifier] tipo nome [= valor]
```

- Em que:

`access_modifier`	(opcional) Modificador de acesso do campo: `public`, `private`, `protected`, etc.
`tipo`	O tipo do campo
`valor`	(opcional) O valor de inicialização do campo

✔ **Naming Guidelines**	O nome de um campo deve ser escrito em **PascalCasing**

Campos

- No seguinte exemplo, são declarados quatro campos na classe `Livro`:
  - `Titulo`: o título do livro
  - `Ano`: o ano de edição do livro
  - `Preco`: o preço do livro (sem IVA)
  - `TaxaIva`: a taxa de IVA do livro (inicializado com o valor de taxa reduzida, de 6%)

```
public class Livro
{
    string Titulo;
    int Ano;
    decimal Preco;
    decimal TaxaIva = 0.06M;
}
```

O campo `TaxaIva` é inicializado com um valor

Campos

- Caso os campos fossem públicos, poderiam ser utilizados fora da classe (ou seja, ler e atribuir valores)
- Exemplo:

```
public class Livro
{
    public string Titulo;
    public int Ano;
    public decimal Preco;
    public decimal TaxaIva = 0.06M;
}
```

```
Livro livro1 = new Livro();

livro1.Titulo = "Um Pouco Mais de Azul";
livro1.Ano = 1992;
```

- No entanto, como iremos ver mais à frente, os campos de uma classe não devem ser utilizados diretamente, fora da mesma

Campos

Exercício

- Acrescente à classe `Aluno`, os campos adequados para armazenar as seguintes informações (escolha os tipos de dados adequados):
  - nome
  - morada
  - email
  - telefone1
  - telefone2
  - data de nascimento

Campos

Exercício: Resolução

```
public class Aluno
{
    string Nome;
    string Morada;
    string Email;
    int Telefone1;
    int Telefone2;
    DateTime DataNascimento;
}
```

CONSTANTES

Constantes

- Uma constante é um valor que nunca muda
- As classes podem conter constantes

- Sintaxe para declarar uma constante:

```
[access_modifier] const tipo nome = valor
```

- Em que:

`access_modifier`	(opcional) Modificador de acesso da constante: `public`, `private`, `protected`, etc.
`tipo`	O tipo da constante. Tipos possíveis: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`, uma enumeração ou um *reference type*
`valor`	O valor de inicialização da constante

- Nota: é obrigatório inicializar uma constante com um valor

Constantes

- É necessário ter em atenção que:
  - é possível declarar várias constantes simultaneamente
  - não é possível modificar o valor de uma constante

- No seguinte exemplo são declaradas várias constantes na classe `A`:

```
public class A
{
    const double PI = 3.1415926535897931;

    public const string pais = "Portugal";

    private const int c1 = 55, c2 = 99, c3 = 110;
}
```

Constantes

- Uma característica a ter em conta nas constantes é que estas acabam por funcionar como membros estáticos
- Ou seja, estão disponíveis para todas as instâncias de uma classe

ENUMERAÇÕES

Enumerações

- Dentro de uma classe podem-se definir enumerações
- Exemplo:

```
public class Livro
{
    string Titulo;
    int Ano;
    decimal Preco;
    decimal TaxaIva = 0.06M;
    public GeneroLivro Genero

    public enum GeneroLivro
    {
        Indefinido,
        Romance,
        Ensaio,
        FiccaoCientifica,
        Policial
    }
}
```

O tipo do campo `Genero`...

...é definido dentro da própria classe

Enumerações

- A decisão de colocar uma enumeração dentro ou fora de uma classe prende-se com a seguinte questão:
  - **a enumeração diz respeito apenas a essa classe?**

- **Se sim**: faz sentido declarar a enumeração dentro da classe
- **Se não**: se pode ser utilizada por outras classes é conveniente declarar a enumeração fora da classe (ver exemplo seguinte)

Enumerações

- No seguinte exemplo é declarada a enumeração `TipoCombustivel`, fora de qualquer classe
- Esta enumeração é utilizada pela classe `Automovel`
- Mas também pode vir a ser utilizada por outras classes

```
public enum TipoCombustivel
{
    Nulo,
    Gasolina,
    Gasoleo,
    Eletricidade,
    Hidrogeneo
}

public class Automovel
{
    TipoCombustivel Combustivel;
}
```

MÉTODOS

Métodos

- As classes possuem dados e executam operações
- Os métodos são o componente que implementam as operações
- Como já vimos no Módulo 3 - Programação Estruturada um método é:
  - um bloco de código
  - pode receber dados (os parâmetros)
  - pode devolver dados (o valor de retorno)

Métodos: Anatomia de um método

- Um método é composto por vários componentes
- Estes componentes formam **a assinatura do método**

```
access modifier  tipo  nome([parâmetro(s)])
```

Um dos seguintes:
- `public`
- `private`
- `protected`
- `internal`
- `protected internal`

Tipo de dados retornados pelo método (`void` se não retorna dados)

Nome do método

Opcional. Lista de parâmetros, separados por vírgulas

- Exemplo:

```
public string MostrarApelido(string nome)
```

✔ Naming Guidelines	O nome de um método deve ser escrito em **PascalCasing**

Métodos: Exemplos

- Exemplo 1: método que não retorna dados e não recebe parâmetros

```
public void Desenhar()
```

- Exemplo 2: método que não retorna dados e recebe 2 parâmetros (do tipo `int`)

```
public void Deslocar(int distanciaHorizontal, int distanciaVertical)
```

- Exemplo 3: método que retorna um número inteiro e recebe 2 parâmetros (do tipo `int`)

```
public int Somar(int a, int b)
```

Métodos: Exemplos

- No seguinte exemplo, é acrescentado à classe `Livro`, o método `PrecoComIva()`, o qual retorna o preço do livro, acrescido da taxa de IVA:

```
public class Livro
{
    string Titulo;
    int Ano;
    decimal Preco;
    decimal TaxaIva = 0.06M;

    public decimal PrecoComIva()
    {
        return Preco + (Preco * TaxaIva);
    }
}
```

- Para invocar o método `PrecoComIva()`:

```
Livro livro1 = new Livro();

decimal p = livro1.PrecoComIva();
```

Métodos

Exercício

- Acrescente à classe `Aluno` 2 métodos:
  - um método para matricular o aluno: recebe o ano em que o aluno se pretende matricular e devolve um valor booleano que indica se a matrícula foi feita com sucesso
  - um método que devolve o número de módulos em atraso do aluno

- Tenha em atenção:
  - o nome do método
  - os parâmetros
  - tipo de dados retornados
- Estes métodos não contêm código no interior, apenas uma instrução `return` (para evitar erros de compilador)

Métodos

Exercício: Resolução

```
public class Aluno
{
    string Nome;
    string Morada;
    string Email;
    int Telefone1;
    int Telefone2;
    DateTime DataNascimento;

    public bool Matricular(int ano)
    {
        return true; // Apenas para evitar erros de compilador
    }

    public int ModulosAtraso()
    {
        return 0; // Apenas para evitar erros de compilador
    }
}
```

ACCESS MODIFIERS

Access modifiers

- Pode ser especificado para as classes e os vários componentes
- Um *access modifier* controla o acesso à classe e aos membros da classe

```
[access_modifier] class Classe
{
    [access_modifier] tipo Campo
    [access_modifier] tipo Constante

    [access_modifier] tipo Método()
    {
        ...
    }
}
```

Access modifiers

- Tipos de *access modifiers* possíveis:
  - `private`
  - `public`
  - `protected`
  - `internal`
  - `protected internal`

- Neste documento apenas serão analisados os dois primeiros (`private` e `public`)

Access modifiers

`private`

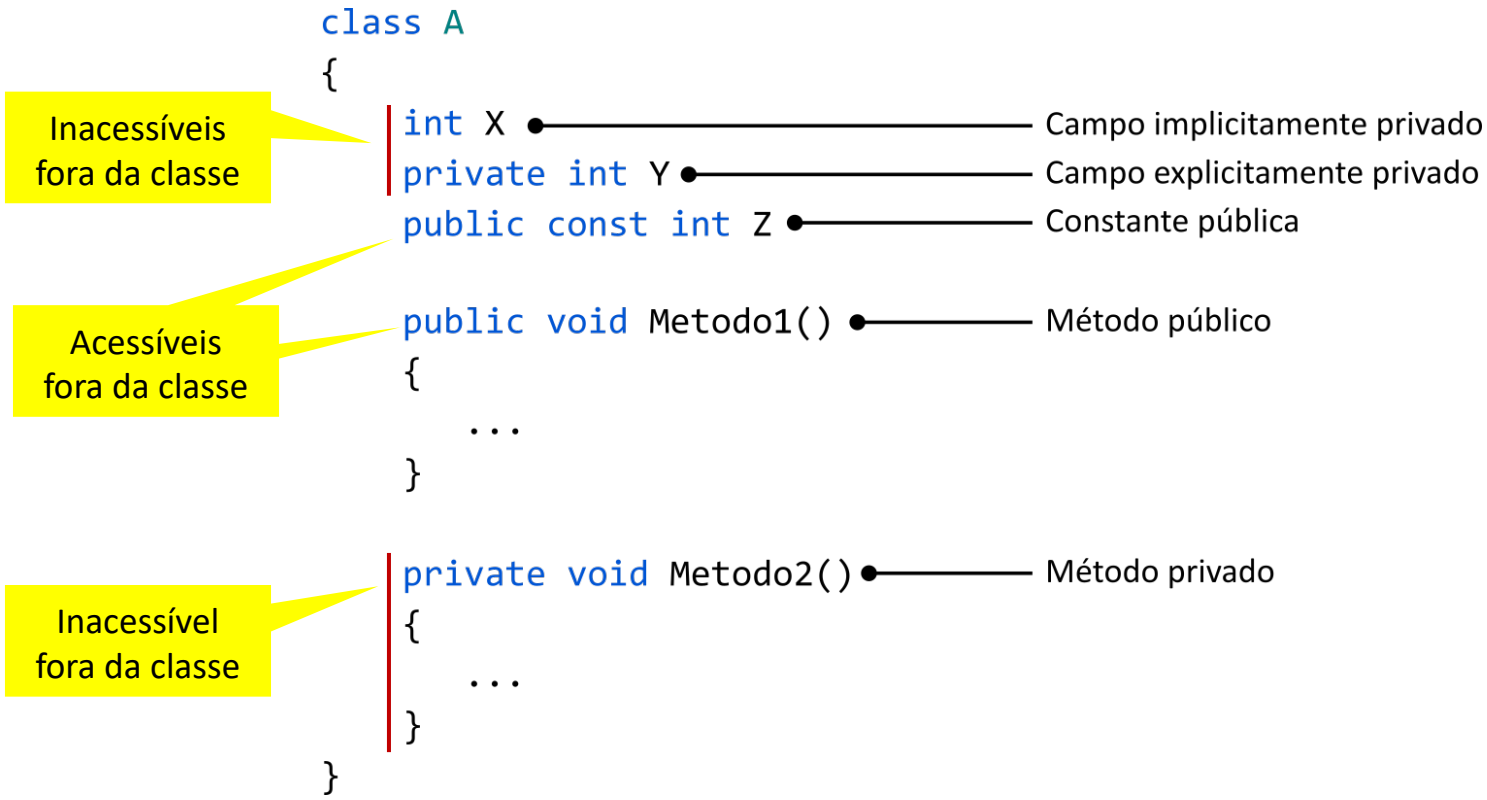
- Os membros de uma classe apenas estão acessíveis dentro da própria classe
- É o *access modifier* por defeito para qualquer membro de uma classe

`public`

- Os membros de uma classe com o *access modifier* `public` estão acessíveis fora da classe

Access modifiers

- Exemplo:



Access modifiers

Recomendações importantes

- Os campos representam os dados de um objeto, logo devem ser **privados**
  - Isto porque não deve ser possível modificar diretamente a partir do exterior os dados internos de uma classe
  - É o princípio subjacente ao conceito de **encapsulamento**
- Deve-se indicar explicitamente que uma propriedade/método é privada/o, utilizando o *access modifier* `private`

Access modifiers

- Com o que sabemos agora sobre *access modifiers*, vamos modificar ligeiramente o exemplo da classe `Livro`, indicando explicitamente que os campos são privados:

```
public class Livro
{
    private string Titulo;
    private int Ano;
    private decimal Preco;
    private decimal TaxaIva = 0.06M;

    public decimal PrecoComIva()
    {
        return Preco + (Preco * TaxaIva);
    }
}
```

Access modifiers

- Vamos também modificar a classe `Aluno`, indicando explicitamente que os campos são privados:

```
public class Aluno
{
    private string Nome;
    private string Morada;
    private string Email;
    private int Telefone1;
    private int Telefone2;
    private DateTime DataNascimento;

    public bool Matricular(int ano)
    {
        return true; // Apenas para evitar erros de compilador
    }

    public int ModulosAtraso()
    {
        return 0; // Apenas para evitar erros de compilador
    }
}
```

ACESSO AOS DADOS DE UMA CLASSE

Acesso aos dados de uma classe

- Dado o que foi aprendido anteriormente sobre *access modifiers*, surge naturalmente a seguinte questão:
  - **Se os campos devem ser privados, como é possível aceder aos dados de uma classe?**
- Em primeiro lugar importa saber que há dados de uma classe que nunca devem ser vistos/acedidos pelo exterior
- Ou seja, são dados internos que apenas interessam para as operações internas de uma classe
- Por outro, existem dados que podem e devem ser "consumidos" pelo exterior
- Por exemplo, na classe `Livro`, deve ser possível ler e modificar o título, o ano, o preço, etc., de qualquer livro
- Como fazer então?
- Existem duas formas:
  - através de *getters* e *setters*
  - através de propriedades

Solução recomendada

GETTERS & SETTERS

Acesso aos dados de uma classe: *getters* e *setters*

- Os *getters/setters* são métodos que nos permitem:
  - obter um campo
  - atribuir um valor a campo

- Por convenção têm o seguinte nome:
  - `GetNome_do_campo()`: obter o valor do campo
  - `SetNome_do_campo()`: atribuir um valor ao campo

Acesso aos dados de uma classe: *getters* e *setters*

- Exemplo: vamos criar um *getter* e um *setter* para o campo `Titulo` na classe `Livro`

```
public class Livro
{
    private string Titulo;

    ... // restantes campos

    public string GetTitulo()
    {
        return Titulo;
    }

    public void SetTitulo(string titulo)
    {
        Titulo = titulo;
    }

    ... // restante código
}
```

getter

setter

Acesso aos dados de uma classe: *getters* e *setters*

- Agora vamos criar *getters* e *setters* para todos os campos da classe `Livro` (o código está condensado por questões de espaço):

```
public class Livro
{
    private string Titulo;
    private int Ano;
    private decimal Preco;
    private decimal TaxaIva = 0.06M;

    public string GetTitulo() { return Titulo; }
    public void SetTitulo(string titulo) { Titulo = titulo; }
    public int GetAno() { return Ano; }
    public void SetAno (int ano) { Ano = ano; }
    public decimal GetPreco() { return Preco; }
    public void SetPreco(decimal preco) { Preco = preco; }
    public decimal GetTaxaIva() { return TaxaIva; }
    public void SetTaxaIva(decimal taxaIva) { TaxaIva = taxaIva; }

    ... // restante código
}
```

Acesso aos dados de uma classe: *getters* e *setters*

- O uso de *getters* e *setters* apresenta duas vantagens:
  - primeiro está-se a respeitar o encapsulamento: os *getters* e *setters* funcionam como uma barreira contra o acesso direto aos dados de uma classe
  - como são métodos, permitem acrescentar código
- Relativamente a este ponto, imaginemos que na classe `Livro` pretendemos efetuar a seguinte restrição:
  - um livro nunca pode ter um preço igual ou menor a zero
- Poderíamos concretizar esta restrição, implementando a seguinte **validação**:

```
public void SetPreco(decimal preco)
{
    if (preco <= 0)
    {
        // sinalizar um erro e retornar
    }

    Preco = preco;
}
```

Só atribuímos um novo preço ao livro se este for superior a zero

PROPRIEDADES

Acesso aos dados de uma classe: Propriedades

- Desvantagens dos *getters* e *setters*:
  - o código fica mais extenso
  - é entediante estar a criar dois métodos para cada propriedade
- É para dar resposta a estas desvantagens que foram criadas as **propriedades**

- Uma propriedade combina as características de um **campo** e de um **método**

- As propriedades têm vindo a evoluir, conforme evolui a linguagem C#
- Existem duas formas de definir propriedades:
  - em conjunto com um campo (*backing field*)
  - *auto-implemented properties* (*auto-properties*)

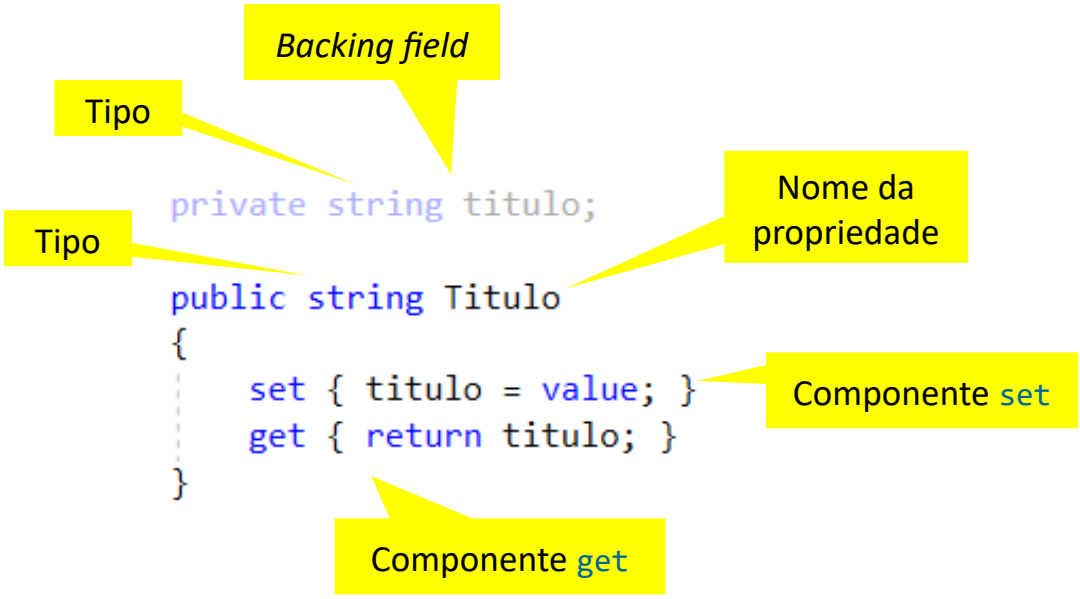
Propriedades: *Backing field*

- Neste primeiro tipo de propriedade é definido um campo privado e uma propriedade pública associada ao campo
- Assim é possível ler ou atribuir um valor ao campo

- Sintaxe:

```
tipo nome
{
    set
    {
        código
    }
    get
    {
        código
        return valor
    }
}
```

Exemplo:



Propriedades: *Backing field*

- A propriedade tem dois componentes:
  - `set`: atribuir um valor ao campo
  - `get`: devolver o valor do campo

- No componente `set` o valor a atribuir ao campo vem no parâmetro `value`

- Repare que em ambos os componentes é possível executar código
- É assim possível, por exemplo, incluir código para validar ou filtrar os valores que são atribuídos ao campo

Propriedades: *Backing field*

- Voltemos novamente ao exemplo de *getter* e *setter* para o campo `Titulo` na classe `Livro` e a correspondente implementação da propriedade `Titulo`:

getter/setter

```
public class Livro
{
    private string Titulo;

    public string GetTitulo()
    {
        return Titulo;
    }

    public void SetTitulo(string titulo)
    {
        Titulo = titulo;
    }
```

Propriedade `Titulo`

```
public class Livro
{
    private string titulo;

    public string Titulo
    {
        set { titulo = value; }
        get { return titulo; }
    }
```

- Como é possível ver pelo exemplo da direita, há uma redução na quantidade de código a introduzir

Propriedades: *Backing field*

- Exemplo de utilização da propriedade `Titulo`:

```
Livro livro1 = new Livro();

livro1.Titulo = "A Ilustre Casa de Ramires";

Console.WriteLine(livro1.Titulo);
```

Atribuir um valor à propriedade

Ler o valor da propriedade

Propriedades: *Backing field*

- O seguinte exemplo demonstra como introduzir código na propriedade, para efetuar validação dos dados:

```
private decimal preco;

public decimal Preco
{
    set
    {
        if (value <= 0)
            preco = 1;
        else
            preco = value;
    }
    get { return preco; }
}
```

- Nota: o componente `set` poderia ser simplificado, utilizando o operador `?`

```
set { preco = (value <= 0) ? 1 : value; }
```

Propriedades: *Auto-properties*

- As *auto-implemented properties* ou *auto-properties* são propriedades em que o compilador C# gera automaticamente um campo (*backing field*) escondido
- O programador apenas necessita de especificar a propriedade
- **Recomenda-se a utilização de *auto-properties* sempre que possível**
- A propriedade contém obrigatoriamente o componente `get` e opcionalmente o componente `set`

- Sintaxe simplificada para definição de uma *auto-property*:

```
tipo nome
{
   get;
   [ [private] set;]
} [ = valor]
```

Propriedades: *Auto-properties*

- Exemplo de definição da propriedade `Titulo`:

```
public class Livro
{
    public string Titulo { get; set; }

    ... // restante código
```

- Exemplo de como utilizar a propriedade `Titulo`:

```
Livro livro1 = new Livro();

livro1.Titulo = "A Ilustre Casa de Ramires";

Console.WriteLine(livro1.Titulo);
```

Atribuir um valor à propriedade

Ler o valor da propriedade

Propriedades: *Auto-properties*

- Os seguintes exemplos demonstram a redução na quantidade de código, utilizando: *getter* e *setter*, *backing field* e propriedade, *auto-property*

getter/setter

```
public class Livro
{
    private string Titulo;

    public string GetTitulo()
    {
        return Titulo;
    }

    public void SetTitulo(string titulo)
    {
        Titulo = titulo;
    }
```

Backing field

```
public class Livro
{
    private string titulo;

    public string Titulo
    {
        set { titulo = value; }
        get { return titulo; }
    }
```

Auto-property

```
public class Livro
{
    public string Titulo { get; set; }
```

Propriedades: *Auto-properties*

- Exemplo: definir uma propriedade e atribuir um valor por defeito

```
public int Numero { get; set; } = 15;
```

- Exemplo: definição de uma propriedade cujo valor apenas pode ser modificado dentro da classe

```
public class Livro
{
    public int Codigo { get; private set; }

    public void UmMetodo()
    {
        Codigo = 99;
    }
}
```

Apenas é possível atribuir um valor dentro da classe

Fora da classe

```
Livro livro1 = new Livro();
livro1.Codigo = 37;
```

Erro: não é possível atribuir aqui um valor

Propriedades: *Auto-properties*

- No seguinte exemplo:
  - definir uma propriedade *readonly*, ou seja, sem o componente `set`
  - o valor desta propriedade apenas pode ser atribuído na declaração ou no construtor da classe

```
public class Livro
{
    public string ISBN { get; }

    public Livro(string titulo)
    {
        Titulo = titulo;
        ISBN = "abc";
    }

    public void UmMetodo()
    {
        ISBN = "abc-1234-001";
    }
}
```

Sem o componente `set`

Atribuir um valor à propriedade, no construtor

Erro: não é possível atribuir aqui um valor

Propriedades

- Implementando *auto-properties* na classe `Livro`, esta fica com muito menos código:

```
public class Livro
{
    public string Titulo { get; set; }
    public int Ano { get; set; }
    public decimal Preco { get; set; }
    public decimal TaxaIva { get; set; } = 0.06M;

    public decimal PrecoComIva()
    {
        return Preco + (Preco * TaxaIva);
    }
}
```

- Tenha em atenção que declaramos as propriedades com o *access modifier* `public` para que possamos ter acesso a estas, fora da classe

Acesso aos dados de uma classe: Propriedades

Exercício

- Dada a última versão da classe `Aluno` (em baixo) transforme os campos em propriedades:

```
public class Aluno
{
    private string Nome;
    private string Morada;
    private string Email;
    private int Telefone1;
    private int Telefone2;
    private DateTime DataNascimento;

    public bool Matricular(int ano)
    {
        return true; // Apenas para evitar erros de compilador
    }

    public int ModulosAtraso()
    {
        return 0; // Apenas para evitar erros de compilador
    }
}
```

Acesso aos dados de uma classe: Propriedades

Exercício: Resolução

```
public class Aluno
{
    public string Nome { get; set; }
    public string Morada { get; set; }
    public string Email { get; set; }
    public int Telefone1 { get; set; }
    public int Telefone2 { get; set; }
    public DateTime DataNascimento { get; set; }

    public bool Matricular(int ano)
    {
        return true; // Apenas para evitar erros de compilador
    }

    public int ModulosAtraso()
    {
        return 0; // Apenas para evitar erros de compilador
    }
}
```

INSTANCIAR UMA CLASSE

Instanciar uma classe

- Para utilizar uma classe é necessário criar uma instância da classe, utilizando a instrução `new`
- Ao criar uma instância de uma classe está-se a criar um objeto

- Sintaxe:

```
Nome_da_Classe variável = new Nome_da_Classe()
```

- Exemplo:

```
Livro livro1 = new Livro();
```

Instanciar uma classe

- Podemos criar várias instâncias de uma classe, ou seja, vários objetos
- Cada instância é uma entidade separada
- Cada instância pode conter dados distintos das outras instâncias
- Exemplo de declaração de 3 instâncias da classe `Livro`:

```
Livro livro1 = new Livro();
Livro livro2 = new Livro();
Livro livro3 = new Livro();
```

Construtor

- Um construtor é um método especial utilizado para inicializar uma instância de uma classe
- A saber:
  - um construtor tem obrigatoriamente o mesmo nome da classe
  - não devolve dados
  - é possível definir vários construtores para a mesma classe (*constructor overloading*)
- Não é obrigatório criar um construtor para uma classe
- Nesse caso, é gerado um *default constructor*, ou seja, um construtor por defeito
- Exemplo:

```
Livro livro1 = new Livro();
```

Default constructor da classe `Livro`

Construtor

- Quando é que deve ser declarado um construtor?
- Quando é necessário efetuar operações de inicialização ao criar uma nova instância de uma classe, tais como:
  - inicializar a instância com dados
  - executar determinadas operações

Construtor

- No seguinte exemplo é criado um construtor para a classe `Livro`, no qual são inicializados os dados de um novo livro:

```
public class Livro
{
    public string Titulo { get; set; }
    public int Ano { get; set; }
    public decimal Preco { get; set; }
    public decimal TaxaIva { get; set; } = 0.06M;

    public Livro(string titulo, int ano, decimal preco, decimal taxaIva)
    {
        Titulo = titulo;
        Ano = ano;
        Preco = preco;
        TaxaIva = taxaIva;
    }

    public decimal PrecoComIva()
    {
        return Preco + (Preco * TaxaIva);
    }
}
```

Construtor

- Com base no construtor definido na página anterior, vamos criar 3 novas instâncias da classe `Livro` (ou seja, 3 novos livros):


```
Livro livro1 = new Livro("Os Maias", 2019, 10.38M, 0.06M);

Livro livro2 = new Livro("Python - Algoritmia e Programação Web", 2015,
24.06M, 0.06M);

Livro livro3 = new Livro("TypeScript: O JavaScript Moderno para Criação
de Aplicações", 2017, 18.82M, 0.06M);
```

Instanciar uma classe

Exercício

- Dada a última versão da classe `Aluno`, crie um construtor para inicialização dos dados e crie 3 instâncias (3 alunos)

```
public class Aluno
{
    public string Nome { get; set; }
    public string Morada { get; set; }
    public string Email { get; set; }
    public int Telefone1 { get; set; }
    public int Telefone2 { get; set; }
    public DateTime DataNascimento { get; set; }

    public bool Matricular(int ano)
    {
        return true; // Apenas para evitar erros de compilador
    }

    public int ModulosAtraso()
    {
        return 0; // Apenas para evitar erros de compilador
    }
}
```

Instanciar uma classe

Exercício: Resolução

```
public class Aluno
{
    public string Nome { get; set; }
    public string Morada { get; set; }
    public string Email { get; set; }
    public int Telefone1 { get; set; }
    public int Telefone2 { get; set; }
    public DateTime DataNascimento { get; set; }

    public Aluno(string nome, string morada, string email, int telefone1, int
telefone2, DateTime dataNascimento)
    {
        Nome = nome;
        Morada = morada;
        Email = email;
        Telefone1 = telefone1;
        Telefone2 = telefone2;
        DataNascimento = dataNascimento;
    }

    ... // Restante código: métodos
}
```

Fornecer um *default constructor*

- Imaginemos a seguinte classe:

```
public class Pessoa
{
    private string Nome;
}
```

- Ao criar uma instância, é utilizado o *default constructor*:

```
Pessoa pessoa1 = new Pessoa();
```

Instância

Default constructor

Fornecer um *default constructor*

- Agora imaginemos que definimos um construtor para a classe `Pessoa`:

```
public class Pessoa
{
    private string Nome;

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

- Como definimos um construtor, o *default constructor* já não é gerado
- O código anterior para criar uma instância vai, pois, dar origem a um erro:

```
Pessoa pessoa1 = new Pessoa();
```

Erro: já não existe o *default constructor*

- Isto levanta um problema pois podemos ter situações em que é necessário utilizar o *default constructor* (por exemplo, código antigo)

Fornecer um *default constructor*

- A solução é muito simples e é considerada boa prática:
  - fornecer um *default constructor*
- Assim sendo, a classe `Pessoa` passa a ter 2 construtores:

```
public class Pessoa
{
    private string Nome;

    public Pessoa()
    { }

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

Default constructor

- E pode ser criadas instâncias de duas formas:

```
Pessoa pessoa1 = new Pessoa();
Pessoa pessoa2 = new Pessoa("Gonçalo Mendes Ramires");
```

FINALIZER

Finalizer

- Um *finalizer* (ou finalizador) é um método que é executado quando uma instância de uma classe é destruída
- Este tipo de membro de uma classe também é conhecido como *destructor* (destrutor)

- Sintaxe:

```
~nome_classe()
{
    código
}
```

Finalizer

- Exemplo:

```
public class Livro
{
    ... // Propriedades

    ... // Construtor

    ~Livro()
    {
        // Fazer qualquer coisa...
    }
}
```

Finalizer

- Importa saber que:
  - os finalizadores são chamados automaticamente pela .NET Framework não podendo ser invocados pelo programador
  - não recebem parâmetros
  - uma classe apenas pode conter um finalizador

- E finalmente:
  - normalmente não é necessário definir um *finalizer*
  - apenas nos casos em que a classe utiliza recursos não geridos pela .NET Framework

METHOD OVERLOADING

Method overloading

- O *method overloading* consiste em criar diferentes versões de um método
- O nome do método permanece o mesmo mas muda o tipo ou o número de parâmetros
- Ou seja, **muda a assinatura do método**
- A .NET Framework contém vários casos de *method overloading*
- Por exemplo, na classe `Console`, existem vários *overloads* do método `WriteLine()`:
  - `Console.WriteLine()`
  - `Console.WriteLine(bool value)`
  - `Console.WriteLine(char value)`
  - `Console.WriteLine(char[] buffer)`
  - `Console.WriteLine(string value)`
  - `Console.WriteLine(string format, object arg0)`
  - `etc.`
- Também é possível efetuar *overloading* do construtor de uma classe

Method overloading: *Overload* do construtor

- No exemplo seguinte são definidos três construtores para a classe `Livro`:

```
public class Livro
{
    public string Titulo { get; set; }
    public int Ano { get; set; }
    public decimal Preco { get; set; }
    public decimal TaxaIva { get; set; } = 0.06M;

    public Livro(string titulo) { Titulo = titulo; }

    public Livro(string titulo, decimal preco)
    {
        Titulo = titulo;
        Preco = preco;
    }

    public Livro(string titulo, int ano, decimal preco, decimal taxaIva)
    {
        Titulo = titulo;
        Ano = ano;
        Preco = preco;
        TaxaIva = taxaIva;
    }
    ... // restante código
}
```

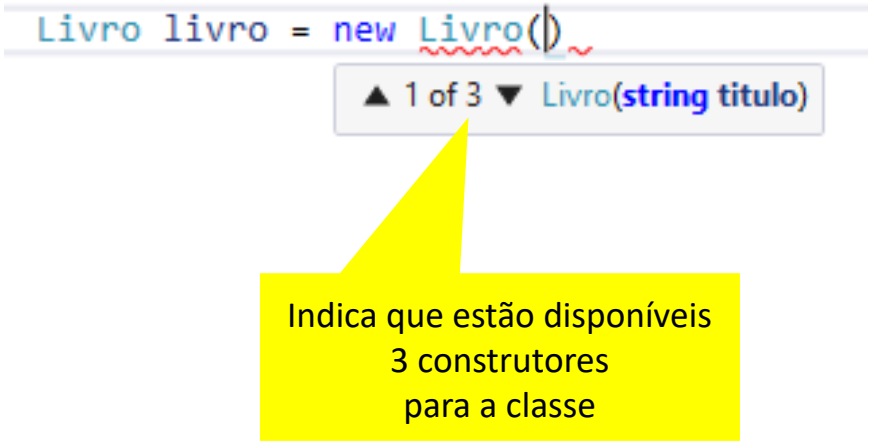
1º Construtor

2º Construtor

3º Construtor

Method overloading: *Overload* do construtor

- Ao criar uma instância da classe `Livro`, o mecanismo Intellisense do Visual Studio indica que existem 3 construtores para a classe:



Method overloading: *Overload* de métodos

- No seguinte exemplo, a classe `Calculos` contém 3 versões do método `Somar()`:

```
public class Calculos
{
    private int x = 5;

    public int Somar(int a)
    {
        return x + a;
    }

    public int Somar(int a, int b)
    {
        return x + a + b;
    }

    public double Somar(double a)
    {
        return x + a;
    }
}
```

1 parâmetro

2 parâmetros

1 parâmetro do tipo `double`

CLASSES E MEMBROS ESTÁTICOS

Classes e membros estáticos

- Uma classe estática é uma classe que não pode ser instanciada
- Uma classe estática apenas pode conter membros estáticos
- Os membros de uma classe estática são acedidos utilizando o nome da classe

- Para especificar que uma classe ou membro de uma classe é estático, é utilizada a *keyword* `static`

- A classe `Math` constitui um exemplo de uma classe estática

Classes e membros estáticos

Qual a utilidade de uma classe estática?

- É uma classe que contém recursos/funcionalidades, úteis para um programa
- Mas que não faz sentido criar diferentes instâncias
- Apenas faz sentido utilizar diretamente a classe (e os seus membros)

Classes e membros estáticos

Exemplo de uma classe estática

- Imaginemos uma classe que efetua conversões (de metros para milhas, de kilogramas para libras, etc.)
- Esta classe chama-se `Conversoes` e pode conter vários métodos
- Neste exemplo contém apenas dois métodos, estáticos:
  - `MetrosMilhas()`: converte metros para milhas (*miles*)
  - `KilogramasLibras()`: converter kilogramas para libras (*pounds*)

Classes e membros estáticos

Exemplo de uma classe estática

```
static class Conversoes
{
    public static double MetrosMilhas(double metros)
    {
        return metros * 0.00062137D;
    }

    public static double KilogramasLibras(double kg)
    {
        return kg * 2.20462262D;
    }
}
```

- Para utilizar os membros da classe estática:

```
// Converter 1000 metros em milhas
double milhas = Conversoes.MetrosMilhas(1000);

// Converter 3.7 kg em libras
double libras = Conversoes.KilogramasLibras(3.7);
```

Repare que invocamos os métodos utilizando diretamente o nome da classe

Classes e membros estáticos

- É ilegar tentar criar uma instância de uma classe estática:

```
Conversoes c = new Conversoes();
```

Erro: não é possível criar uma instância desta classe

Classes e membros estáticos

- Numa classe podem conviver membros estáticos e não estáticos:

```
class MyClass
{
    public int A { get; set; }

    public static int B;

    public void FazerQualquerCoisa1()
    {
        // fazer qualquer coisa...
    }

    public static void FazerQualquerCoisa2()
    {
        // fazer qualquer coisa...
    }
}
```

Propriedade estática

Método estático

Classes e membros estáticos

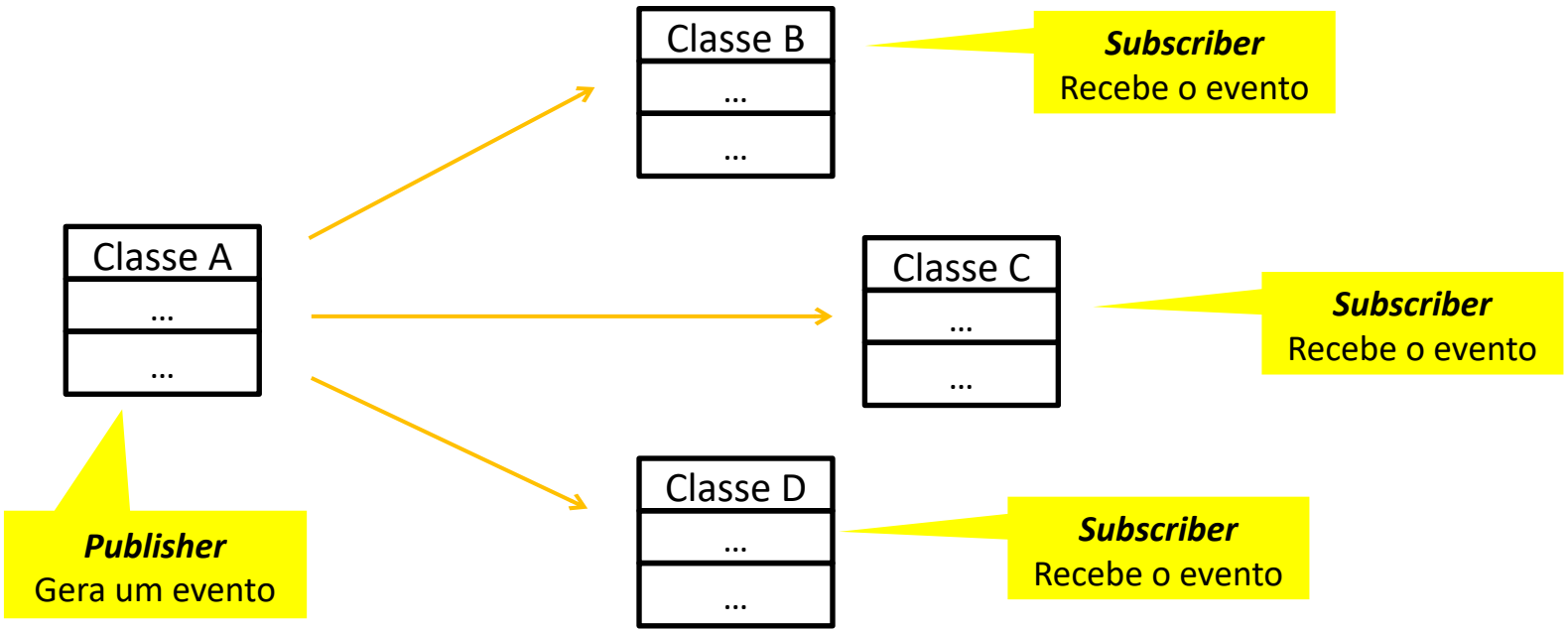
- A forma de ter acesso a estes membros é normal:
  - através de uma instância para os membros não estáticos
  - de forma direta com o nome da classe para os membros estáticos

```
MyClass m1 = new MyClass();

// Utilizar os membros não estáticos
m1.A = 33;
m1.FazerQualquerCoisa1();

// Utilizar os membros estáticos
MyClass.B = 97;
MyClass.FazerQualquerCoisa2();
```

EVENTOS

Eventos

- Os eventos permitem que uma classe notifique outras classes sempre que um determinado acontecimento ocorre
- A classe que gera o evento é chamada de *publisher*
- As classes que recebem o evento são chamadas de *subscribers*
- Uma classe pode receber eventos de várias classes



NAMING GUIDELINES

Naming guidelines

- Na .NET Framework existe um conjunto de *naming guidelines* (recomendações) para os identificadores dos vários componentes de um projeto
- Estas regras não são obrigatórias mas recomenda-se fortemente que os programadores as adotem nos seus programas

Capitalização

- Existem dois métodos principais para a capitalização:
  - PascalCasing: o primeiro caracter de cada palavra em maiúscula
  - camelCasing:
    - o primeiro caracter da primeira palavra em minúscula
    - o primeiro caracter das restantes palavras em maiúscula
- Exemplos:

`HtmlColor`	→	PascalCasing
`CartaoCidadao`	→	PascalCasing
`nomeCliente`	→	camelCasing
`numeroFaturaCancelada`	→	camelCasing

Naming guidelines

- Eis algumas das *naming guidelines* para a .NET Framework:

Componente	Casing	Exemplo
Namespaces	Pascal casing	`namespace` **`Veiculos`**
Classes	Pascal casing	`public class` **`Automovel`**
Campos	Pascal casing	`private int` **`Velocidade`**
Propriedades	Pascal casing	`public int` **`Quilometragem`** `{ get; set; }`
Métodos	Pascal casing	`public int` **`Acelerar`**`()`
Parâmetros	Camel casing	`public Automovel(string` **`marca`**`, string` **`modelo`**`, int` **`quilometragem`**`)`

- Para saber mais sobre as *naming guidelines*:
  https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines

Naming guidelines

```
namespace Veiculos
{
    public class Automovel
    {
        private int Velocidade;

        public string Marca { get; set; }
        public string Modelo { get; set; }
        public int Quilometragem { get; set; }

        public Automovel(string marca, string modelo, int quilometragem)
        {
            Velocidade = 0;
            Marca = marca;
            Modelo = modelo;
            Quilometragem = quilometragem;
        }

        public int Acelerar()
        {
            Velocidade += 10;
            return Velocidade;
        }
    }
    }
```

Namespace: Pascal casing

Classe: Pascal casing

Campo: Pascal casing

Propriedades: Pascal casing

Parâmetros: Camel casing

Método: Pascal casing