

SEMANAS:

- INICIO/APRESENTAÇÃO
- INTRODUÇÃO AOS ALGORITMOS E ESTRUTURAS DE DADOS
- ESTRUTURAS DE CONTROLO DE FLUXO EM C#
- CONCEITO DE CLASSE E OBJETO
- SEPARAÇÃO ENTRE INTERFACE PÚBLICA E IMPLEMENTAÇÃO PRIVADA
- ESTRUTURAS DE DADOS – INTRODUÇÃO
- ANÁLISE DE ALGORITMOS E COMPLEXIDADE
- ALGORITMOS DE ORDENAÇÃO
- ALGORITMOS DE ORDENAÇÃO (PARTE 2)
- PROVA DE AVALIAÇÃO
- LISTAS E FILAS
- TABELAS DE HASH
- TRATAMENTO DE COLISÕES
- ÁRVORES BINÁRIAS
- TIPOS GENÉRICOS E TRATAMENTO DE EXCEPÇÕES
- PROVA DE AVALIAÇÃO

Sumário:

- Separação entre interface pública e implementação privada
  - Encadeamento de construtores
  - *Encapsulated fields*
  - Métodos públicos e privados
- Exercício de aplicação
- Modificadores de visibilidade
- Exercício de exposição de *boas práticas de programação*. Separação:
  - GUI - *graphical user interface*
  - Lógica de negócio
  - Persistência dos dados

# Algoritmos e Estruturas de Dados

**CTeSP - Tecnologias e Programação de Sistemas de Informação 2020/21**



**Docente: [Ricardo Henriques](#).**

## Separação entre interface pública e implementação privada

### Encadeamento de construtores utilizando `this`

Outro uso da primitiva `this` é aplicada no desenho de classes segundo a técnica designada de *constructor chaining*. Este padrão de desenvolvimento é útil quando se tem uma classe onde se definem múltiplos construtores. Dado que é comum ser necessário validar os vários argumentos para impor o cumprimento das regras de negócio, pode evidenciar-se uma lógica de verificação repetitiva.

Vamos introduzir o seguinte exercício como motivação para a esta matéria.

- Defina no seu projeto da conta bancária uma nova classe designada por `Pessoa`.
- Caracterize `Pessoa` mediante o registo da sua `Idade`, `Nome`, `Género` e `Número Fiscal`.
- Defina os construtores:
  - `Pessoa(String nome, int idade, long nif)`
  - `Pessoa(String nome, int idade, Genero sexo, long nif)`
- Imponha nos dois construtores que se o valor do parâmetro `idade` for inferior a zero a `idade`, por omissão, deverá ser dezoito.

Nota: o tipo `Genero` deverá ser um enumerado!



```

class Pessoa {
    public enum Genero { masculino, feminino }

    private int idade;
    private string nome;
    private long nif;
    private Genero sexo;

    metodos e construtores
}

```

### Atributos de Pessoa

Numa primeira implementação dos construtores tem-se uma óbvia redundância no que respeita à verificação do valor de idade:

```

public Pessoa(int idade, long nif) {
    this.idade = idade < 0 ? 18 : idade;
    this.nif = nif;
}

public Pessoa(String nome, int idade, long nif) {
    this.nome = nome;
    this.idade = idade < 0 ? 18 : idade;
    this.nif = nif;
}

public Pessoa(String nome, int idade, Genero sexo, long nif) {
    this.nome = nome;
    this.idade = idade < 0 ? 18 : idade;
    this.sexo = sexo;
    this.nif = nif;
}

```

### Construtores da classe Pessoa

A abordagem correcta é designar um construtor como *master constructor* o qual deverá tratar o máximo de argumentos e contém toda a lógica de criação de um objeto desta classe. Os demais construtores podem fazer uso deste construtor através da primitiva `this` para redirecionarem para o *master constructor*. Desta forma apenas é necessário refletir na implementação de um construtor, pois os demais estão despojados de lógica.

```

public Pessoa(int idade, long nif) {
    this.idade = idade < 0 ? 18 : idade;
    this.nif = nif;
}

public Pessoa(String nome, int idade, long nif) : this(idade, nif) {
    this.nome = nome;
}

public Pessoa(String nome, int idade, Genero sexo, long nif) : this(nome, idade, nif) {
    this.sexo = sexo;
}

```

### Construtores encadeados da classe Pessoa

#### Fluxo dos construtores...



Uma nota adicional que é importante reter é que uma vez passados os argumentos ao *master constructor* (e após a execução integral deste) o construtor invocado originalmente irá executar o código presente no restante bloco de código. Para clarificar as ideias, altere a última definição de forma a incluir um `Console.WriteLine(...)` de forma a "sentir" a execução da ordem.

## Encapsulamento

Todas as linguagens baseadas em objetos devem respeitar os três principais pilares conceitos de suporte à programação baseada em objetos:

### Encapsulamento

*Traduz o mecanismo que a linguagem facilita para preservar a integridade dos dados e esconder os detalhes da implementação.*

### Herança

*Representa a capacidade de promover a reutilização de código.*

### Polimorfismo

*Carateriza a forma de como a linguagem permite tratar objetos relacionados de forma similar.*

Não faz parte desta disciplina aprofundar estes três conceitos, mas o encapsulamento é um elemento dentro do tema da separação entre interface pública e implementação privada, pelo que vamos aqui expô-la.

O primeiro pilar da POO é o **encapsulamento**, no qual se escondem os detalhes da implementação ao mesmo tempo que restringe o acesso a métodos e propriedades. Dessa forma a encapsulação permite que uma implementação mude sem que isso afete os outros componentes do sistema - desde que se respeite a interface previamente definida.

Desta forma uma classe exporta apenas os elementos e métodos que caracterizam o estado e a forma de interagir com o objeto. Exeriormente não é necessário (nem conveniente) a quem utiliza os métodos de um objeto saber os detalhes de implementação. Assim, apenas é exportada a assinatura *pública* da classe.

Pode o leitor questionar: qual a necessidade da encapsulação?

Uma vez que este conceito orbita à volta da política que restringe o acesso direto aos valores que caracterizam uma instância, é importante centrar a questão nas consequências práticas que o não cumprimento desta política acarretaria.

Suponha que a classe `Pessoa` estaria definida como:

```
class Pessoa
{
    public string nome;
    public int idade;
}
```

Sabendo que o limite superior em C# de um `int` é 2147483467 facilmente podemos deduzir que permitir aceder diretamente ao estado de um objeto significaria dar azo ao surgimento de instruções como `pessoal.idade = 5604`, ou mesmo a atribuição de um valor negativo.

Ora o que se passa neste ponto é que o tipo de dados não tem capacidade intrínseca de compreender o domínio do problema de modo avaliar determinados valores como

inválidos.

Nota: este tipo de restrições designam-se por *invariante de estado*, havendo linguagens que permitem a sua definição a par logo da definição do atributo, o que não é o caso da maioria das LP's, acontecendo o mesmo com o C#.

Ora, a definição de métodos de acesso/modificação (*getters/setters*), facultam o necessário mecanismo de preservação da integridade lógica do estado de dados do objeto. Por esta razão a definição pública de campos de dados de um objeto deve ser completamente abolida, devendo ser sempre privada (*private*). Há duas técnicas em C# (e Delphi, por exemplo) de indiretamente interagir com os atributos de um objeto:

1. Através da definição do par `get` e `set`, também designados de *accessor* e *mutator*.
2. Através da definição de um *type property*.

Qualquer que seja a técnica escolhida o importante é reter que uma classe bem encapsulada deve esconder todos os detalhes do seu funcionamento interno. Isto é vulgarmente designado por *black box programming*.

### **Encapsulação por accessor/mutator**

De acordo com a definição acima, da classe `Pessoa` bastaria acrescentar:

```
class Pessoa
{
    private string nome;
    private int idade;

    public int getIdade()
    {
        return idade;
    }

    public void setIdade(int i)
    {
        // se i for válido...
        idade = i;
    }

    /* idem para o nome, por exemplo para retirar
       caracteres inválidos */
}
```

### **Encapsulação por type properties**

Este é também designado por *encapsulated fields*. De acordo com a definição acima, da classe `Pessoa` bastaria acrescentar:

```
class Pessoa
{
    private string nomePessoa;
    private int idadePessoa;

    // Properties

    public string nome
    {
        get { return nomePessoa; }
        set { nomePessoa = value; }
    }
```



```
public int idade
{
    get { return idadePessoa; }
    set {
        // verificação da validade
        idadePessoa = value;
    }
}
```

## Propriedades *read-only* e *write-only*

Quando se encapsula os dados pode ser relevante permitir que uma propriedade seja apenas *read-only*. Para o conseguir basta omitir o bloco `set`. De forma dual para garantir que uma propriedade é *write-only* basta omitir o bloco `get`. Uma propriedade *read-only* habitualmente apenas é definida aquando da criação do objeto ou por processamento e não por mensagem dirigida ao objeto.

---

# Fábrica de objetos

Nota: esta matéria recorre à noção de herança e polimorfismo (*interface*), estando de momento fora do alcance da matéria lecionada!

---

## Exercício 6:

Complete e altere a definição das classes `Pessoa` e `ContaBancaria` de forma a conseguir modelar em PPO a seguinte realidade:

- Uma `Pessoa` é caracterizada pelos atributos já vistos: NIF, nome, idade (altere para data de nascimento) e o género.
- Uma `ContaBancaria` deverá poder ter um ou dois titulares, referenciados pelo seu NIF.
- Constitua uma classe `Banco` que ingloba o conhecimento do conjunto de pessoas (suas clientes) e das múltiplas contas bancárias, que é o objeto do seu negócio.
- Deverá num banco poder efectuar:
  - Abrir uma nova conta (com um titular obrigatório).
  - Adicionar um segundo titular
  - Listar os titulares (NIF + nome) de uma conta.
  - Depositar um valor numa conta.
  - Levantar o dinheiro de uma conta.
  - Transferir internamente (no mesmo banco) dinheiro entre duas contas.

Nota: utilize apenas vectores (*array*) como estrutura de dados de pessoas e contas bancárias.

## Modificadores de visibilidade

Os modificadores de visibilidade estão directamente associados à utilização do encapsulamento. Assim para saber utilizar convenientemente o encapsulamento é necessário compreender quais os aspectos de um tipo que ficam visíveis e para que partes da aplicação. Especificamente tipos como classes, interfaces, estruturas,



enumerações (e *delegates*), assim como os seus membros - propriedades, métodos, construtores, etc. - são sempre definidos utilizando uma chave específica de controlo da sua visibilidade. A seguinte tabela sintetiza a sua utilização e significado.

C# modificador de acesso	Pode ser aplicado a...	Descrição
public	types e type members	Itens com este modificador não impõem restrições de acesso. Os <i>public member</i> podem ser acedidos a partir de qualquer objecto ou classe derivada. Um <i>public type</i> pode ser acedido a partir de <i>assemblies</i> externos.
private	type members e nested types	Itens privados apenas podem ser acedidos pela classe (ou estrutura) na qual estão definidos.
protected	type members e nested types	Itens deste tipo não estão directamente acessíveis a partir de uma variável de instância, mas apenas a partir do tipo que a define, bem como das classes de si derivadas.
internal	types e type members	Itens internos estão acessíveis apenas dentro do <i>assembly</i> no qual está contido, dessa forma não estando disponíveis para quaisquer outros.
protected internal	types e type members	Quando combinados estes modificadores significa que o item está apenas disponível dentro do seu <i>assembly</i> , da sua classe e das classes do seu <i>assembly</i> que de si derivam.

MRH, 2020/21

