

ENCAPSULAMENTO

PACOTES

- A pacotes são **módulos que agrupam classes**, forçosamente separadas ao nível do sistema de ficheiros
- De modo a definir que uma classe pertence a determinado pacote, é necessário incluir a respetiva declaração no início do ficheiro

```
package aguiaj.iscte;  
  
public class BinaryImage {  
    ...  
}
```

PACOTES NO SISTEMA DE FICHEIROS

\MyClass.java

MyClass

\aguiaj

\iscte

\BinaryImage.java

\GrayscaleImage.java

\ColorImage.java

\Color.java

aguiaj.iscte

BinaryImage

ColorImage

Color

GrayscaleImage

\mypackage

\MyBinaryImage.java

\MyColor.java

\util

\ImageUtils.java

mypackage.util

ImageUtils

mypackage

MyBinaryImage


MyColor

MODIFICADORES DE ACESSO


- Em Java, o conceito de encapsulamento pode ser aplicado utilizando modificadores de acesso, em atributos, métodos, e construtores
 - **public**: permite acesso do exterior do pacote e da classe
 - **package-private**: permite acesso dentro do pacote (utilizado até agora, não tem palavra-chave na sintaxe do JAVA)
 - **private**: não permite acesso do exterior, sendo possível apenas o acesso interno (no contexto dos métodos da classe)
 - Existe também **protected**, que será abordado noutra UC

ENCAPSULAMENTO AO NÍVEL DO PACOTE

- O pacote funciona como um **módulo composto por classes relacionadas** (p.e. funções sobre imagens, jogo, manipulação de matrizes)
- As classes e seus membros que não sejam relevantes para o exterior, ou até que não seja desejável que estejam disponíveis, podem ficar com visibilidade ***package-private***



```
package mypackage;  
  
class HiddenClass {  
    ...  
}
```



```
package mypackage;  
  
public class MyImage {  
    ...  
    void hiddenMethod() {  
        ...  
    }  
}
```

ENCAPSULAMENTO AO NÍVEL DO CLASSE

- Encapsulamento é um conceito que consiste em "esconder" atributos e métodos de um objeto do exterior, havendo uma separação entre **interface** e **implementação**, permitindo
 - **Maior flexibilidade** na evolução da implementação de uma classe
 - **Maior controlo** sobre a correcta utilização dos objetos de uma classe

ENCAPSULAMENTO DE ATRIBUTOS

É boa prática **encapsular os atributos dos objetos**. Desta forma é possível ter controlo sobre a sua correta utilização.

```
public class Rectangle {  
  
    private int width;  
    private int height;  
  
    ...  
  
}
```

```
Rectangle r = new Rectangle(20, 40);  
r.width = -5;
```

Instrução inválida se usada noutro contexto que não o da classe Rectangle!

ENCAPSULAMENTO DE MÉTODOS

Um método deverá ser encapsulado caso não faça sentido ser executado diretamente do exterior.

```
public class Rectangle {  
  
    private int width;  
    private int height;  
  
    public Rectangle(int width, int height) {  
        validateDimension(width, height);  
        this.width = width;  
        this.height = height;  
    }  
  
    private static void validateDimension(int w, int h) {  
        if(w < 0 || h < 0)  
            throw new IllegalArgumentException("...");  
    }  
  
}
```


PROPRIEDADES DE UM OBJETO

- O estado de um objeto é guardado no valor dos seus atributos. Contudo, determinada **informação** que se pretende saber sobre um objeto pode não ser obtida diretamente do valor de um atributo, mas sim **calculada com base nos valores dos atributos**.
- Do ponto vista externo, o **acesso a uma propriedade** de um objeto deverá ser **transparente** no que diz respeito à forma como o valor é obtido (diretamente ou calculado).


EXEMPLO:

PROPRIEDADES DE UM RETÂNGULO

```
public class Rectangle {  
  
    private int width;  
    private int height;  
  
    ...  
  
    public int getWidth() {  
        return width;  
    }  
  
    public int getArea() {  
        return width * height;  
    }  
  
    public boolean isSquare() {  
        return width == height;  
    }  
  
}
```

A utilização dos prefixos "get" e "is" seguidos do nome da propriedade começado por maiúscula é apenas uma convenção.


SEPARAÇÃO ENTRE INTERFACE E IMPLEMENTAÇÃO: Point



```
public class Point {  
    private double abscissa;  
    private double ordinate;  
    ...  
}
```

Implementação
baseada em
coordenadas
cartesianas


Implementação
baseada em
coordenadas
polares



```
public class Point {  
    private double radius;  
    private double angle;  
    ...  
}
```

SEPARAÇÃO ENTRE INTERFACE E IMPLEMENTAÇÃO: Point

```
public class Point {  
  
    private double abscissa;  
    private double ordinate;  
  
    public Point(double abscissa, double ordinate) {  
        this.abscissa = abscissa;  
        this.ordinate = ordinate;  
    }  
  
    public double getAbscissa() {  
        return abscissa;  
    }  
  
    public double getOrdinate() {  
        return ordinate;  
    }  
  
    public double getRadius() {  
        return Math.sqrt(abscissa * abscissa + ordinate * ordinate);  
    }  
  
    public double getAngle() {  
        return Math.atan2(ordinate, abscissa);  
    }  
}
```



```
Point p = new Point(1.2, 2.7);  
double abs = p.getAbscissa();  
double ord = p.getOrdinate();  
double rad = p.getRadius();  
double ang = p.getAngle();
```

SEPARAÇÃO ENTRE INTERFACE E IMPLEMENTAÇÃO: Point

```
public class Point {
```

```
    private double radius;  
    private double angle;
```

```
    public Point(double abscissa, double ordinate) {  
        this.radius = Math.sqrt(abscissa*abscissa + ordinate*ordinate);  
        this.angle = Math.atan2(ordinate, abscissa);  
    }
```


```
    public double getAbscissa() {  
        return Math.cos(angle)*radius;  
    }
```

```
    public double getOrdinate() {  
        return Math.sin(angle)*radius;  
    }
```

```
    public double getRadius() {  
        return radius;  
    }
```

```
    public double getAngle() {  
        return angle;  
    }
```


```
}
```




```
Point p = new Point(1.2, 2.7);  
double abs = p.getAbscissa();  
double ord = p.getOrdinate();  
double rad = p.getRadius();  
double ang = p.getAngle();
```

SEPARAÇÃO ENTRE INTERFACE E IMPLEMENTAÇÃO

Color1 e Color2 constituem implementações alternativas de uma classe para representar cores...



```
public class Color1 {  
    private int r;  
    private int g;  
    private int b;  
  
    public Color1(int r, int g, int b) {  
        this.r = r;  
        this.g = g;  
        this.b = b;  
    }  
  
    public int getR() {  
        return r;  
    }  
  
    public int getG() {...}  
  
    public int getB() {...}  
}
```



```
public class Color2 {  
    private int[] rgb;  
  
    public Color2(int r, int g, int b) {  
        rgb = new int[]{r, g, b};  
    }  
  
    public int getR() {  
        return rgb[0];  
    }  
  
    public int getG() {...}  
  
    public int getB() {...}  
}
```

INTERFACES

Uma interface em JAVA representa um conceito abstratamente, definindo um nome e declarando um conjunto de operações.

```
public interface IColor {  
  
    /**  
     * Devolve o valor de vermelho (Red) [0, 255]  
     */  
    int getR();  
  
    /**  
     * Devolve o valor de verde (Green) [0, 255]  
     */  
    int getG();  
  
    /**  
     * Devolve o valor de azul (Blue) [0, 255]  
     */  
    int getB();  
  
}
```

IMPLEMENTAÇÃO DE INTERFACES

Uma classe implementa uma interface caso defina um método público para cada operação da mesma.

```
public class Graytone implements IColor {  
  
    private int value;  
  
    public Graytone(int value) {  
        if(value < 0 || value > 255)  
            throw new IllegalArgumentException("...");  
  
        this.value = value;  
    }  
  
    public int getR() {  
        return value;  
    }  
  
    public int getG() {  
        return value;  
    }  
  
    public int getB() {  
        return value;  
    }  
  
}
```

Obrigatório implementar porque a classe Graytone implementa a interface IColor!

Obrigatório implementar porque a classe Graytone implementa a interface IColor!

Obrigatório implementar porque a classe Graytone implementa a interface IColor!

A RETER

- Pacotes
- Modificadores de acesso
- Encapsulamento
 - Ao nível do pacote
 - Ao nível da classe
 - Atributos
 - Métodos
 - Propriedades de um objeto
- Separação entre interface e implementação
- Interfaces

