Programação em Java - Fundamentos

4 - Packages, interfaces e qualificadores de visualização

Citeforma

Jose Aser Lorenzo, Pedro Nunes, Paulo Jorge Martins

jose.l.aser@sapo.pt, pedro.g.nunes@gmail.com, paulojsm@gmail.com

Fevereiro de 2012

Sumário

Packages, interfaces e qualificadores de visualização

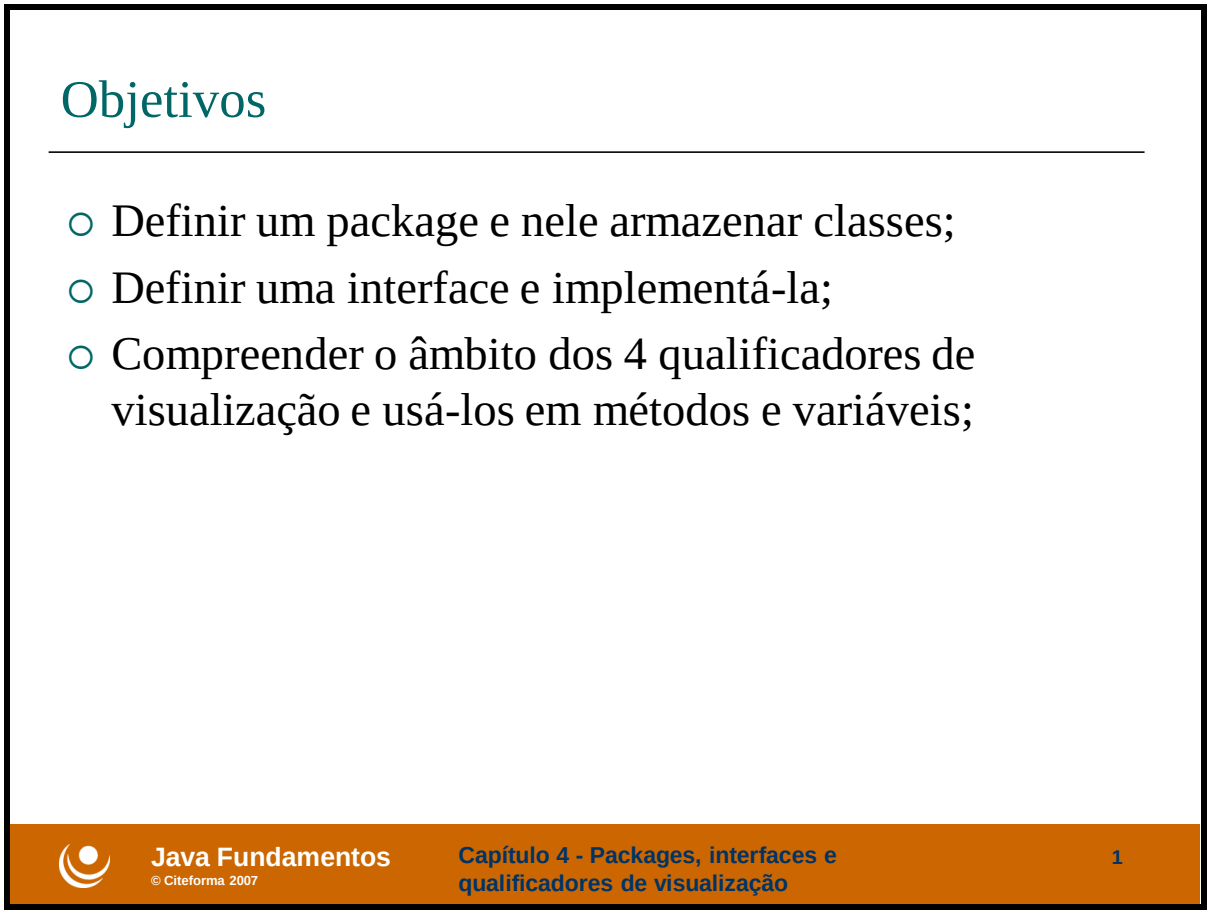
citeforma

Programação em Java Fundamentos

Capítulo 4 - Packages, interfaces e qualificadores de visualização

José Aser Lorenzo
Pedro Nunes
Paulo Jorge Martins

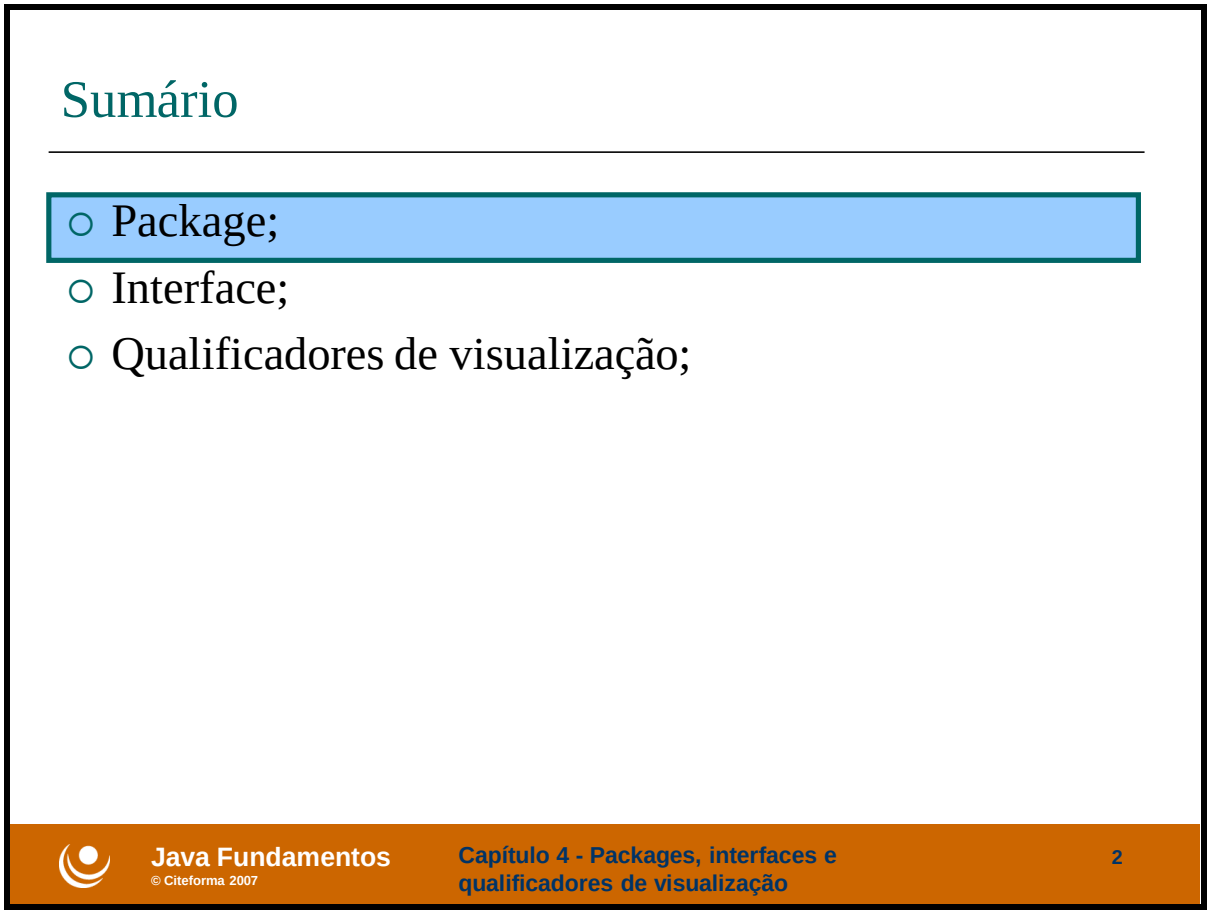
Objetivos



No fim deste capítulo saberá criar um package e nele armazenar as suas classes, compreenderá a noção de interface e será capaz de os definir e implementar, compreenderá o âmbito dos 4 qualificadores de visualização e poderá usá-los em métodos e variáveis das suas classes Java.

As classes desenvolvidas nos exercícios deste capítulo deverão ficar dentro do projeto **JavaFundamentos** e dentro do package **capitulo4**.

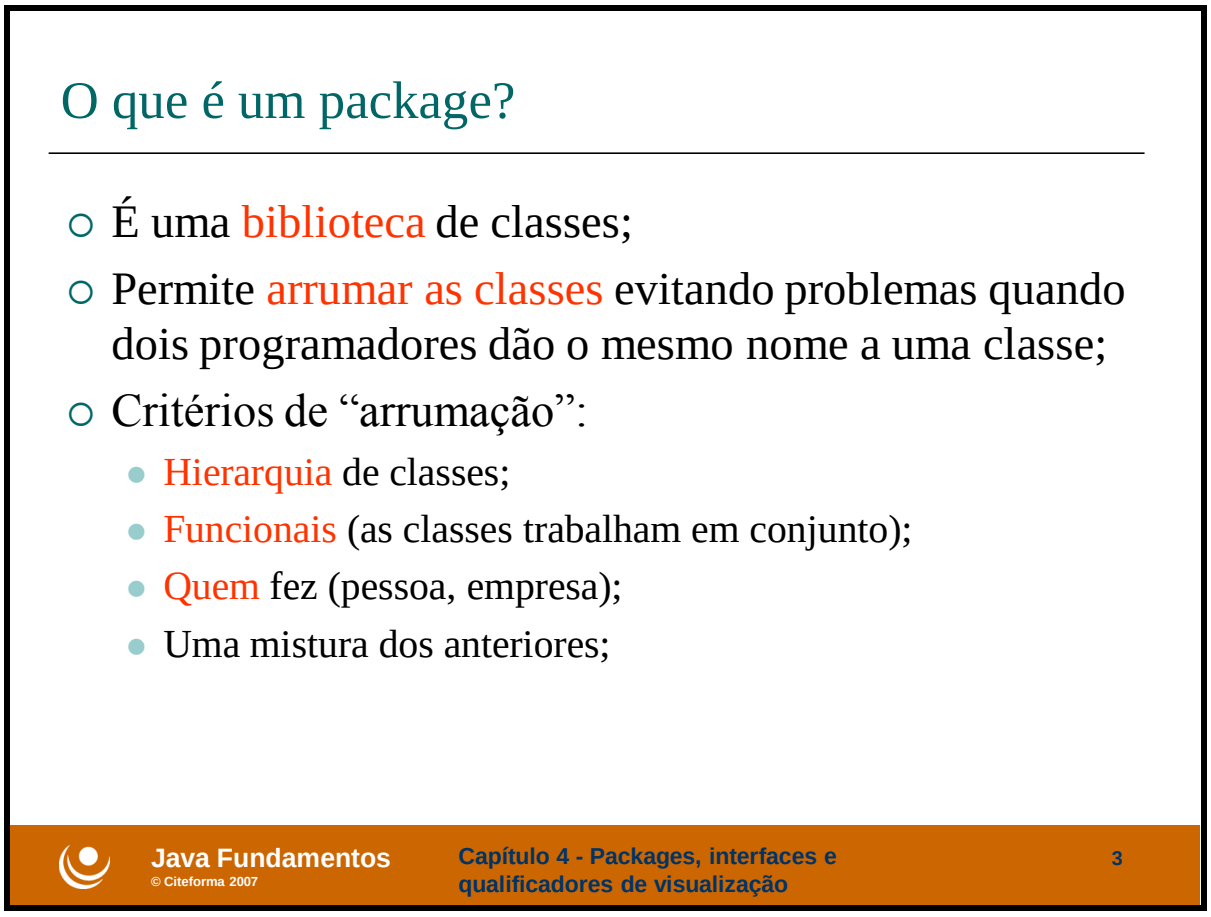
Package



Os packages estão para as classes Java como as diretorias (ou "*folders*") estão para os ficheiros nos nossos discos rígidos. Como há uma probabilidade elevada de duas pessoas darem o mesmo nome a dois ficheiros diferentes, quando se criam repositórios centrais de ficheiros é recomendado que cada pessoa agrupe os seus trabalhos (ficheiros) dentro da sua diretoria de trabalho. Desta forma os ficheiros são identificados não só pelo respetivo nome, mas também pelo nome do computador e pelo caminho de diretorias que é necessário percorrer para o encontrar.

Os packages Java funcionam da mesma forma; por um lado são um mecanismo para arrumar as classes e por outro permitem controlar a sua visualização e a dos seus componentes (métodos e variáveis)

O que é um package?



Um package funciona como uma biblioteca de classes, armazenadas segundo um critério.

Esse critério pode ser:

- Um ramo hierárquico de classes;
- Classes que trabalham em conjunto no desempenho de uma função comum, mas que não possuem uma relação hierárquica entre si;
- Classes desenvolvidas por uma pessoa / empresa e que irão ser usadas numa aplicação que inclui outros packages de outras pessoas / empresas;
- Uma mistura dos critérios anteriores;

Criar um package

Criar um package

- Criar uma directoria com o nome p1 e criar a classe A1 dentro dela;
- Editar a classe A1 e adicionar package p1; no início;

Java Fundamentos
© Citeforma 2007

Capítulo 4 - Packages, interfaces e qualificadores de visualização

4

O Java utiliza as diretorias do sistema operativo para armazenar os packages. Todas as classes pertencentes ao package p1 terão que ser armazenadas na diretoria p1. Atenção que **o Java é sensível a letras maiúsculas e minúsculas, pelo que o nome da diretoria tem que ser exatamente igual ao do package**, mesmo em ambiente Microsoft Windows.

Para criar um package:

1. Criar uma diretoria com o mesmo nome do package para armazenar as classes (desse package) dentro dela;
2. Na definição da classe incluir na primeira linha a palavra reservada **package**, seguida do respetivo nome. No exemplo seguinte a classe **A1** pertence ao package **p1**.

```
package p1;
class A1 {
    …
}
```

A classe A1 passou a ser conhecida por p1.A1 (package.classe). Este mecanismo estende o universo de nomes de classes, pois permite outra classe A1, desde que definida num package diferente, por exemplo p2. Neste caso a segunda classe seria identificada por p2.A1.

Os arquitetos Java recomendam que cada empresa ou programador crie as suas classes num package com o seu nome, usando apenas letras minúsculas. Isto evita possíveis conflitos de nomes quando no mesmo projeto são usados packages desenvolvidos por empresas/pessoas diferentes.

Quando na definição da classe se omite o package significa que a classe pertence ao package por omissão, que está na diretoria atual. Nessa diretoria não poderão existir duas classes com o mesmo nome, assim como não podem existir dois ficheiros com o mesmo nome.

Compilar e executar a uma classe definida dentro de um package

Compilar e executar classe definida em package

- Compilar a classe dentro da diretoria:
  - **javac A1.java;**
- Executar a classe fora da diretoria:
  - **cd ..**
  - **java p1.A1**

A compilação das classes Java é feita executando o comando **javac** seguido do caminho para a o ficheiro (.java) que contém o código fonte:

```
c:\> javac   c:\java\p1\A1.java
```

Ou, estando posicionado na diretoria p1, executar apenas o comando:

```
c:\java\p1> javac A1.java
```

Executar a classe

Só podemos executar classes que possuem método main definido com a assinatura correta. Estas classes têm ser executadas na diretoria fora do package (cd **..**) com o comando **java** seguido do seu nome completo (“full qualified name”):

```
c:\java> java p1.A1
```

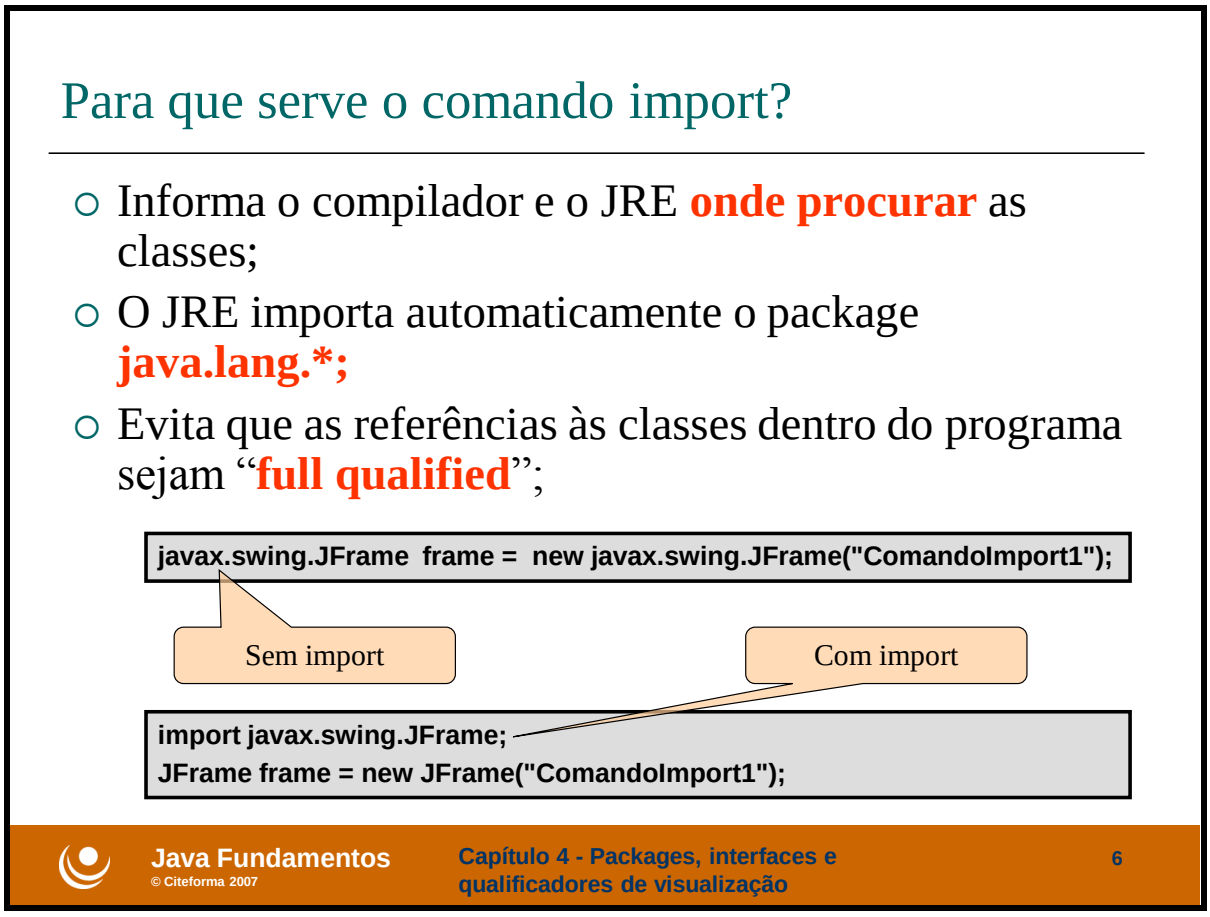
Hierarquia de packages

Assim como uma diretoria pode conter subdiretorias, um package pode conter outros packages.

```
package p1.p2.p3;
class A1 {
   …
}
```

No exemplo acima o package p3 está incluído em p2, que por sua vez pertence a p1. No sistema operativo existirá uma estrutura de diretorias equivalente (p1 contem p2 que por sua vez contem p3).

Para que serve o comando import?



Como vimos no ponto anterior o nome de uma classe deve incluir o nome do respetivo package. No entanto, nos programas desenvolvidos até aqui, referenciámos classes sem indicar o respetivo package. Isto porque todas elas pertencem ao package **java.lang** que é importado automaticamente pelo compilado. Se a classe que pretendemos usar pertencer a outro package temos que a referenciar com o nome completo ("full qualified").

O primeiro exemplo no slide utiliza a classe JFrame que está fora do package **java.lang** e por isso tem que ser referenciada com o respetivo **package** (javax.swing) usando o nome completo.

No segundo exemplo o comando **import javax.swing.JFrame;** indica o package onde o compilador deve procurar a classe JFrame, o que permite dispensar a indicação do package quando se referencia a classe.

Referência genérica

É frequente necessitarmos várias classes do mesmo package, por exemplo JFrame, JLabel, e JButton. Para evitar as referências individuais podemos utilizar o comando **import javax.swing.*** para referenciar de uma só vez todas as classes pertencentes ao package javax.swing.

O código gerado não será mais pesado que no caso em que referenciamos as classes individualmente, pois o comando **import** não implica o carregamento das classes para dentro do ficheiro byte-code, pois funciona como um apontador para as diretorias onde devem ser procuradas as classes.

Import e referências ambíguas

Import e referências ambíguas

- O comando **java.util.*** indica que devem ser procuradas todas as classes no package **util** que está dentro do package **java**;

```
import java.util.*;
import java.sql.*;
      Date d1 = new Date();
      Date d2 = new Date(System.currentTimeMillis());
```

Ambos packages têm uma classe Date, o que gera ambiguidade

Ambiguidade resolvida não usando import e recorrendo ao Full Qualified Name

```
java.util.Date d1 = new java.util.Date();
java.sql.Date d2 = new java.sql.Date(System.currentTimeMillis());
```

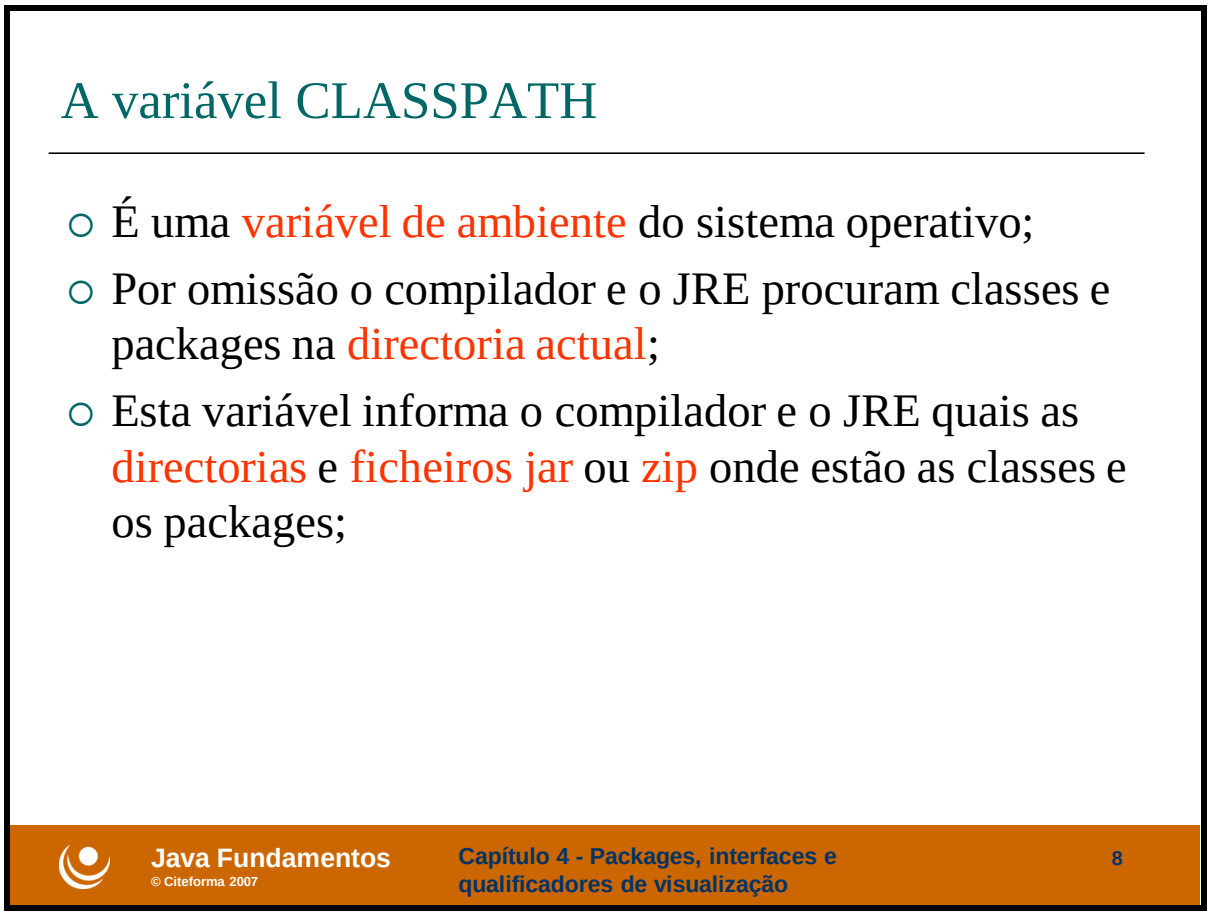
Java Fundamentos

Capítulo 4 - Packages, interfaces e qualificadores de visualização 7

A utilização de referências genéricas como as utilizadas no exemplo anterior pode gerar situações ambíguas, como ilustra o slide acima.

Este erro deve-se ao facto de ambos packages possuírem uma classe com o nome Date. A única forma de ultrapassar este erro é **não** usar o comando **import** e referenciar as classes pelo seu nome completo.

A variável CLASSPATH



O uso de packages alarga o universo de nomes para as classes e facilita a sua arrumação, mas levanta problemas quando se pretende compilar e correr uma classe pertencente a um package. Os problemas resultam de dois factos:

- O compilador e a máquina virtual Java procuram as classes nas diretorias definidas na variável de ambiente CLASSPATH;
- O nome da classe passa a incluir a estrutura de packages dentro da qual a classe está definida;

Considere a classe A1 definida dentro do package p1, ao qual corresponde a diretoria C:\progJava\testes\p1. A classe passou a ser conhecida por p1.A1. A classe pode ser compilada dentro da diretoria p1, dando origem ao ficheiro C:\progJava\testes\p1\A1.class.

Se ao tentar executar a classe for utilizado o comando abaixo, obtemos um erro cuja mensagem diz que não foi possível encontrar a classe A1:

```
C:\> java p1.A1
```

A diretoria que contem o package (C:\progJava\testes) tem que ser adicionada à variável de ambiente **CLASSPATH.** Quando o compilador recebe a diretoria raiz do package vai à procura de uma subdiretoria de nome p1 dentro da qual estará a classe A1.

```
c:\> set CLASSPATH=c:\progJava\testes
c:\> java p1.A1
```

Alternativamente podemos indicar à JVM a CLASSPATH, usando a opção classpath ou –cp dos comandos java e javac:

```
c:\> java –cp "c:\progJava\testes"  p1.A1
```

A variável CLASSPATH pode também receber nomes de ficheiros com extensão zip e jar. A estrutura de diretorias que inclui um package, ou conjunto de packages, onde estão guardados os ficheiros bytecode (class) pode ser compactada para formato ZIP (standard). O ficheiro ZIP resultante desta operação pode ser aberto pelo JVM como se as classes estivessem numa estrutura de diretorias em disco.

Com o JDK é fornecido o utilitário **jar.exe** que funciona de modo semelhante ao comando tar do Unix/Linux. Este utilitário permite compactar os packages e seus sub packages para dentro de um ficheiro, com extensão jar. Normalmente o conteúdo deste ficheiro é comprimido, sendo utilizado o algoritmo ZIP, embora o resultado final tenha extensão **jar**.

Interface

Sumário

- Package;
- Interface;
- Qualificadores de visualização;

O que é uma interface em Java?

O que é uma interface em Java?

- Pode ser vista sob três perspetivas:

1. Um conjunto de métodos públicos que uma classe oferece às outras;
2. Ao implementar uma interface a classe assume o compromisso de respeitar a "norma" definida pela interface;
3. A linguagem Java suporta um mecanismo de herança simples. As interfaces permitem "simular" a herança múltipla;

Java Fundamentos
© Citeforma 2007
Capítulo 4 - Packages, interfaces e qualificadores de visualização
10

Com as interfaces o programador pode definir o que a classe deve fazer, sem especificar como o vai fazer. Sintaticamente a definição de uma interface é semelhante à definição de uma classe, mas os métodos são apenas declarados e não implementados.

Na maioria das linguagens orientadas a objetos o conceito de interface está relacionado com o conjunto de métodos públicos que uma classe deve oferecer às outras, i.e. o conjunto de mensagens que as outras classes podem usar para interagir com ela.

A implementação de uma interface não depende da hierarquia de classes, permitindo que duas classes não relacionadas hierarquicamente entre si implementem a mesma interface e tenham assim comportamentos semelhantes. As interfaces permitem:

- Simular o mecanismo de herança múltipla que existe em C ++ (uma classe pode herdar de várias classes);
- Obrigar à implementação dos métodos definidos na interface, o que implica que ela respeita a "norma" definida pela interface;
- Definir num único sítio (a interface) um conjunto de constantes relacionadas com um determinado package ou aplicação.

Como definir uma interface?

Como definir uma interface?

- Indicando a assinatura dos seus métodos;
- Não definir variáveis;
- Podemos definir constantes com public final e static;
- Se a interface for public todos os seus métodos serão public;

```
package capitulo4;
public interface EmpregadoCalculoIRS {
   public final float[ ][ ] escaloesIRS = {
      {5600f, 5.0f}, {14000f, 7.0f}, {40000f, 10.0f}, {0f, 15.0f}};
   public double calculoImposto(double salarioBruto);
   public double calculoSalarioLiquido(double salarioBruto);
}
```

Java Fundamentos
© Citeforma 2007
Capítulo 4 - Packages, interfaces e qualificadores de visualização
11

Uma interface define-se indicando a assinatura dos métodos que a vão constituir. Não é permitido declarar métodos **static**.

Numa interface podemos também definir constantes que receberão obrigatoriamente os qualificadores: **public, final** e **static**. Desta forma tornam-se constantes públicas

Mesmo que uma interface não seja declarada como **public**, as constantes e métodos declarados nessa interface, quer o explicitemos ou não, são sempre **public**.

Este código será usado num exemplo mais à frente.

Como implementar uma interface?

Como implementar uma interface? #1/2

- Uma classe pode implementar várias interfaces;
- Uma interface pode ser implementada por várias classes;
- Para implementar uma interface a classe tem que definir todos os métodos da interface, respeitando na integra as suas assinaturas;

Java Fundamentos
© Citeforma 2007

Capítulo 4 - Packages, interfaces e qualificadores de visualização 12

Uma interface pode ser implementada por muitas classes, assim como uma classe pode implementar várias interfaces.

A implementação de uma ou várias interfaces por uma classe não impede que essa classe possa ainda estender outra, i.e. não inibe o mecanismo de herança, que continua a poder ser usado. Aliás, esta técnica é um dos mecanismos usados para “simular” a herança múltipla.

Uma classe não abstrata que implemente uma interface deve obrigatoriamente implementar os métodos definidos nessa interface, respeitando as suas assinaturas.

Como implementar uma interface? #2/2

```
package capitulo4;
public abstract class Empregado implements SalarioDeEmpregado {
   ... Definição normal da classe…

   public double calculoImposto(double salarioBruto) {
      //à partida fica no maior escalão
      float percentagemIRS = escaloesIRS[escaloesIRS.length - 1][1];
      for (int i = 0; i < escaloesIRS.length - 1; i++) {
         if (salarioBruto <= escaloesIRS[i][0]) {
            percentagemIRS = escaloesIRS[i][1];
            break;
         }
      }
      return salarioBruto*(percentagemIRS/100);
   }
   public double calculoSalarioLiquido(double salarioBruto) {
      return salarioBruto - calculoImposto(salarioBruto);
   };
}
```

Java Fundamentos
© Citeforma 2007

Capítulo 4 - Packages, interfaces e qualificadores de visualização

13

Como se mostra no exemplo do slide, para implementar uma interface usa-se na definição da classe a palavra reservada "***implements***", obrigando à implementação de todos os métodos declarados na interface.

Na implementação dos métodos de uma interface somos obrigados a usar o qualificador **public**, pois esses métodos são sempre **public** e o Java proíbe a redução da sua visibilidade.

Exercício sobre implementação de interface

Exercício sobre implementação de interface

- Crie a interface EmpregadoCalculoIRS definida anteriormente;
- Considere a classe Empregado definida no capítulo anterior. Altere essa classe de forma a implementar a interface, o que permite o calculo do IRS e salário liquido de um empregado;

Java Fundamentos
© Citeforma 2007
Capítulo 4 - Packages, interfaces e qualificadores de visualização
14

Definição da interface

O código da interface vai incluir um array onde a primeira coluna é o salário anual bruto e a segunda coluna é a respetiva taxa de IRS. Este array ficará disponível para todas as classes que implementem a interface.

```
package capitulo4;
public interface EmpregadoCalculoIRS {
    public final float[][] escaloesIRS = {
        {5600f, 5.0f}, {14000f, 7.0f}, {40000f, 10.0f}, {0f, 15.0f}};
    public double calculoImposto(double salarioBruto);
    public double calculoSalarioLiquido(double salarioBruto);
}
```

Implementação da interface

Vamos copiar as classes Empregado, EmpregadoAssalariado, EmpregadoVendedor, EmpregadoConsultor e EmpregadoEmpresa definidas no exemplo final do capítulo anterior para o package do capítulo 4.

Vamos alterar a classe Empregado de forma que implemente a interface definida no ponto anterior. Desta forma todas as subclasses de Empregado terão a capacidade de calcular o valor do IRS e o salário liquido.

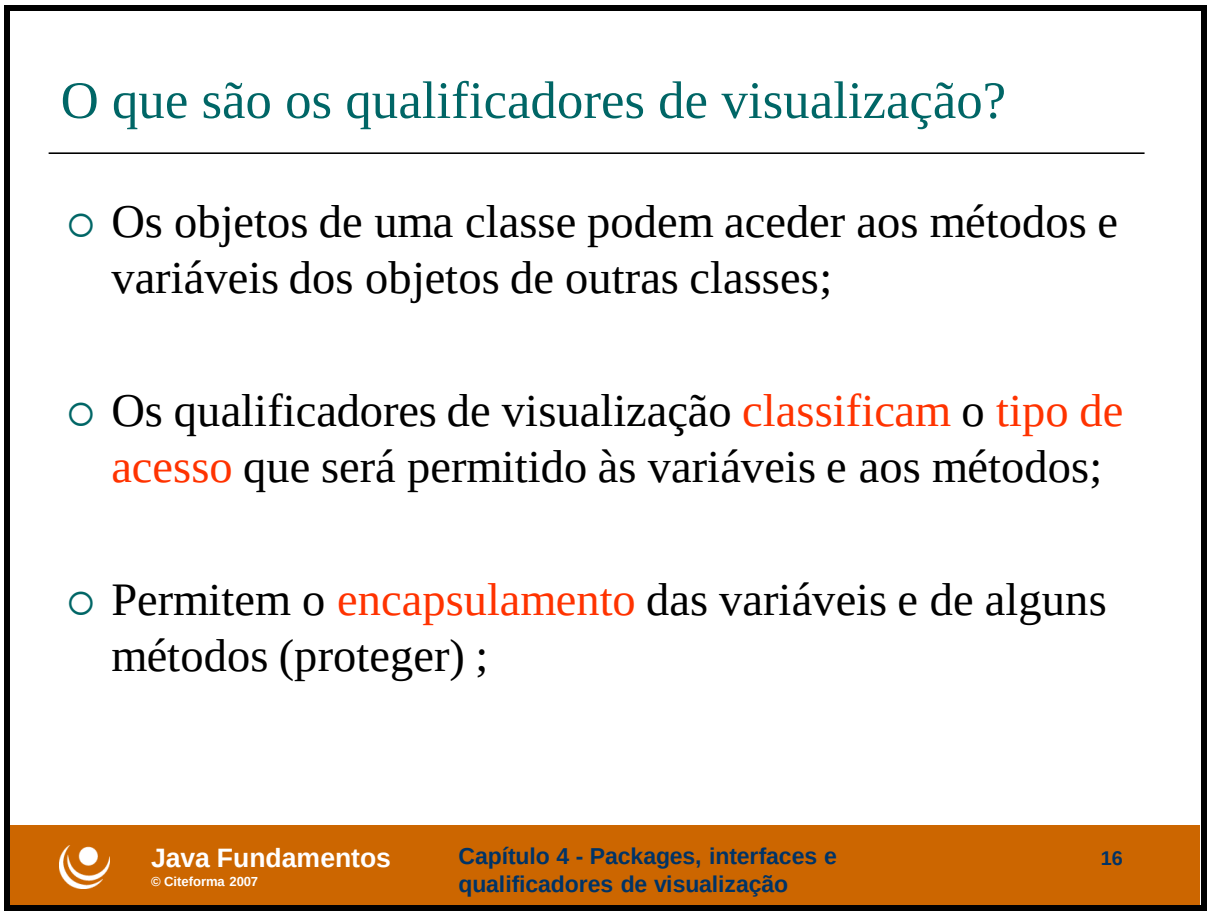
```
package capitulo4;
public abstract class Empregado implements EmpregadoCalculoIRS {

    public static long ultimoNumEmp=0;

    public String nome;
    public long numero;

    public Empregado(String s) {
        super();
        this.nome = s;
        this.numero = ++ultimoNumEmp;
    }

    public final long obterNumero() {
        return numero;
    }

    public final String obterNome() {
        return nome;
    }

    public final void alterarNome(String s) {
        this.nome = s;
    }

    //declaro método abstrat
    public abstract float calcularSalarioBruto();

    //método da interface
    @Override
    public double calculoImposto(double salarioBruto) {
        //à partida fica no maior escalão
        float percentagemIRS = escaloesIRS[escaloesIRS.length - 1][1];
        for (int i = 0; i < escaloesIRS.length - 1; i++) {
            if (salarioBruto <= escaloesIRS[i][0]) {
                percentagemIRS = escaloesIRS[i][1];
                break;
            }
        }
        return salarioBruto*(percentagemIRS/100);
    }
    //método da interface
    @Override
    public double calculoSalarioLiquido(double salarioBruto) {
        return salarioBruto - calculoImposto(salarioBruto);
    };
}
```

Qualificadores de visualização

Sumário

- Package;
- Interface;
- Qualificadores de visualização;

O que são os qualificadores de visualização?



Em programação orientada por objetos os métodos de umas classes invocam os métodos e alteraram as variáveis de outras classes. Por razões de segurança e integridade do estado dos objetos é normal proteger as variáveis e alguns métodos das chamadas externas, o que se consegue com os qualificadores de visualização.

Encapsulamento é a técnica de proteger as variáveis de um objeto permitindo que o seu acesso seja apenas feito através de métodos designados de “accessors” (*getters* e *setters*). Ao restringir a alteração de um atributo de um objeto por um método podemos validar e rejeitar o conteúdo a atribuir.

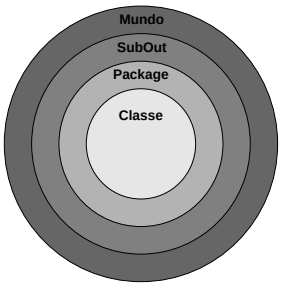
Os qualificadores de visualização são também usados para diminuir o impacto de futuras alterações ao código. Se um método não estiver acessível a outras classes sabemos que apenas é consumido internamente e portanto podemos reescrevê-lo com outra assinatura, fazendo as devidas alterações dentro da classe, tendo a garantia que as outras classes não serão afetadas.

Os qualificadores de visualização permitem ainda diminuir a complexidade aparente de uma classe, pois apresentando ao exterior apenas os métodos necessários, será mais fácil entender como esta deve ser usada.

Quais são os qualificadores de visualização?

Quais são os qualificadores de visualização?

Visualização / Qualificador	Classe	Package	SubOut	Mundo
private	SIM			
package	SIM	SIM		
protected	SIM	SIM	SIM	
public	SIM	SIM	SIM	SIM



Java Fundamentos
© Citeforma 2007
Capítulo 4 - Packages, interfaces e qualificadores de visualização
17

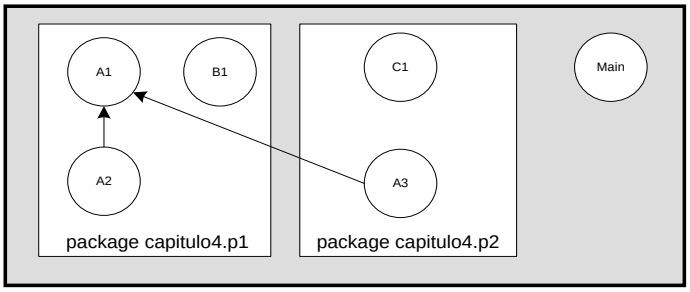
A tabela acima apresenta os qualificadores: **private**, "**package"**, **protected** e **public** usados na linguagem Java e a visibilidade que lhes está associada:

- A coluna **Classe** especifica se os métodos e variáveis definidos numa classe são visíveis por outros métodos definidos dentro da mesma classe. Como se verifica são sempre visíveis, qualquer que seja o qualificador.
- A coluna **Package** especifica se os métodos e variáveis definidos numa classe são visíveis pelas classes definidas no mesmo package, quer exista ou não uma relação hierárquica entre essas classes. O importante é estarem definidas no mesmo package e não a relação hierárquica.
- A coluna **SubOut** especifica se os métodos e variáveis definidos numa classe são visíveis pelas suas subclasses quando estas são definidas noutro package.
- A coluna **Mundo** indica se os métodos e variáveis definidos numa classe são visíveis por outras classes que não pertencem ao mesmo package e não têm relação hierárquica entre si.

O diagrama mostra como se relacionam os conjuntos associados a cada qualificador.

Exercício com qualificadores de visualização

Exercício com qualificadores de visualização



- Desenvolver o seguinte conjunto de classes:
  - A classe A1 tem quatro variáveis e quatro métodos, cada um definido com um qualificador (private, package, protected e public);
  - A2 e A3 estendem A1, A2 está no mesmo package e A3 no outro;
  - B1 não estende A1, mas está no mesmo package. C1 também não tem relação com A1 e está definida no outro package;
  - A classe Main está fora dos packages e cria um objecto de cada classe para testar o acesso às respectivas variáveis e métodos;

Para demonstrar as funcionalidades dos qualificadores de visualização apresentamos um exemplo que cria todas as situações possíveis. Para isso são definidas várias classes, dentro de 3 packages, com as interligações ilustradas no diagrama do slide:

- A classe A1 tem quatro variáveis e quatro métodos, cada um definido com um qualificador (private, package, protected e public).
- As classes A2 e A3 estendem as definições de A1, estando A2 no mesmo package de A1 e A3 no outro.
- A classe B1 não tem relação hierárquica com A1, mas está definida no mesmo package. A classe C1 também não tem relação com A1 e está definida num package diferente.

A classe Main está fora dos outros dois packages e cria um objeto de cada uma das outras classes para testar o acesso às respetivas variáveis e métodos.

```
package capitulo4.p1;
public class A1 {
    static int ultimo=1;

    private     int nPri;
    /*package*/ int nPac; //friendly=package
    protected   int nPro;
    public      int nPub;
```

```
    //construtor
    public A1() {
        nPri=ultimo;
        nPac=ultimo;
        nPro=ultimo;
        nPub=ultimo;
        ultimo+=1;
    }

    private     void mPri() {System.out.println("mPri()");}
    /*package*/ void mPac() {System.out.println("mPac()");}
    protected   void mPro() {System.out.println("mPro()");}
    public      void mPub() {System.out.println("mPub()");}

    public void ver() {
        System.out.println("ver() em A1");
        System.out.println("nPri="+nPri);
        System.out.println("nPac="+nPac);
        System.out.println("nPro="+nPro);
        System.out.println("nPub="+nPub);
        mPri();
        mPac();
        mPro();
        mPub();
    }
}
```

```
package capitulo4.p1;
import capitulo4.p1.A1;
public class A2 extends A1 {
    public void ver() {
        super.ver();
        System.out.println("ver() em A2");
        //System.out.println("nPri="+nPri);
        System.out.println("nPac="+nPac);
        System.out.println("nPro="+nPro);
        System.out.println("nPub="+nPub);
        //mPri();
        mPac();
        mPro();
        mPub();
    }
}
```

```
package capitulo4.p2;
import capitulo4.p1.A1;
public class A3 extends A1 {
    public void ver() {
        super.ver();
        System.out.println("ver() em A3");
        //System.out.println("nPri="+nPri);
        //System.out.println("nPac="+nPac);
        System.out.println("nPro="+nPro);
        System.out.println("nPub="+nPub);
        //mPri();
        //mPac();
```

```
        mPro();
        mPub();
    }
}
```

```
package capitulo4.p1;
import capitulo4.p1.A1;
public class B1 {
    public static void ver() {
        A1 o4 = new A1();
        o4.ver();
        //System.out.println("o4.nPri="+o4.nPri);
        System.out.println("o4.nPac="+o4.nPac);
        System.out.println("o4.nPro="+o4.nPro);
        System.out.println("o4.nPub="+o4.nPub);
        //o4.mPri();
        o4.mPac();
        o4.mPro();
        o4.mPub();
    }
}
```

```
package capitulo4.p2;
import capitulo4.p1.A1;
public class C1 {
    public static void ver() {
        A1 o5 = new A1();
        o5.ver();
        //System.out.println("o5.nPri="+o5.nPri);
        //System.out.println("o5.nPac="+o5.nPac);
        //System.out.println("o5.nPro="+o5.nPro);
        System.out.println("o5.nPub="+o5.nPub);
        //o5.mPri();
        //o5.mPac();
        //o5.mPro();
        o5.mPub();
    }
}
```

```
package capitulo4;
import capitulo4.p1.A1;
import capitulo4.p1.A2;
import capitulo4.p2.A3;
import capitulo4.p1.B1;
import capitulo4.p2.C1;

public class Main {
    public static void main(String[] args)  {
        System.out.println("Objecto de A1");
        A1 o1 = new A1();
        o1.ver();
        System.out.println("Objecto de A2");
```

```
        A2 o2 = new A2();
        o2.ver();
        System.out.println("Objecto de A3");
        A3 o3 = new A3();
        o3.ver();
        System.out.println("Objecto em B1");
        B1.ver();
        System.out.println("Objecto em C1");
        C1.ver();
    }
}
```

O resultado da execução da classe Main é apresentado a seguir:

```
>java   capitulo4.Main
Objecto de A1
ver() em A1
nPri=1
nPac=1
nPro=1
nPub=1
mPri()
mPac()
mPro()
mPub()
Objecto de A2
ver() em A1
nPri=2
nPac=2
nPro=2
nPub=2
mPri()
mPac()
mPro()
mPub()
ver() em A2
nPac=2
nPro=2
nPub=2
mPac()
mPro()
mPub()
Objecto de A3
ver() em A1
nPri=3
nPac=3
nPro=3
nPub=3
mPri()
mPac()
mPro()
mPub()
ver() em A3
nPro=3
nPub=3
mPro()
mPub()
Objecto em B1
ver() em A1
nPri=4
```

```
nPac=4
nPro=4
nPub=4
mPri()
mPac()
mPro()
mPub()
o4.nPac=4
o4.nPro=4
o4.nPub=4
mPac()
mPro()
mPub()
Objecto em C1
ver() em A1
nPri=5
nPac=5
nPro=5
nPub=5
mPri()
mPac()
mPro()
mPub()
o5.nPub=5
mPub()
```

Os métodos ver() das subclasses A2 e A3 chamam o método ver() da super classe. Este último método consegue ver todas as variáveis e métodos definidos na mesma classe, pelo que é usado para provar que as variáveis e métodos definidos em A1 são herdados para os objetos de A2 e A3. A não visualização de algumas variáveis e métodos em objetos de A2, A3, B1 e C1 deve-se à acção dos qualificadores.

public

As variáveis e métodos definidos com qualificador **public** são visíveis por todas as classes. No exemplo dos qualificadores, a variável nPub e o método mPub(), definidos em A1, são sempre visíveis, quer sejam invocados dentro da classe A1, pelas subclasses de A1, pela classe B1 ou C1, quer estejam no mesmo package ou em packages diferentes.

package

A palavra **package** é usada em dois contextos: por um lado representa um conjunto de classes, por outro lado representa um qualificador de visualização para variáveis e métodos. O seu uso como conjunto de classes foi descrito no início do capítulo.

Para usar **package** como qualificador de visualização basta omiti-lo, ou seja, o qualificador usado por omissão representa visualização no mesmo package. Isto significa que, se uma variável ou método não tem qualificador, então é visível no package. Este tipo equivale ao **Friendly** do C++.

As variáveis e métodos definidos com o qualificador por omissão numa classe, são visíveis a todas as classes definidas dentro do mesmo package, quer exista ou não uma relação hierárquica entre elas. O exemplo dos qualificadores ilustra como a variável nPac e o método mPac() são visíveis em A2 e B1 que pertencem ao package p1. Já não são visíveis a A3 e C1 que pertencem ao package p2.

protected

O qualificador **protected** estende o **package**, pois alarga a visualização de variáveis e métodos a subclasses definidas fora do mesmo package. No exemplo dos qualificadores repara-se que a variável nPro e o método mPro(), além de visíveis em A2 e B1 (mesmo

package), são também visíveis em A3, que está num package diferente de A1, mas que é uma subclasse de A1. A classe C1 não as consegue ver, pois não está no mesmo package e não tem relação hierárquica com A1.

private

O qualificador **private** é o mais restritivo pois os métodos e as variáveis só são visíveis dentro da própria classe. No exemplo dos qualificadores a variável nPri e o método mPri() só são vistos pelos métodos definidos dentro de A1. Os métodos definidos noutras classes não lhes acedem diretamente, sendo obrigatório usar o método ver() de A1.

O private é utilizado para proteger as variáveis que, se manipuladas do exterior, podem deixar o objeto num estado inconsistente. Também serve para proteger os métodos que, se usados incorretamente, corrompem o objeto.

private e herança

Os métodos e variáveis definidos como **private** são herdados pelas subclasses, mas não são visíveis por estas. Será necessário usar outro método definido na super classe (pai), para manipular essas variáveis ou chamar esses métodos. Veja-se no exemplo dos qualificadores o que sucede com a variável nPri e o método mPri(), que nos objetos das classes A2 e A3 só são consultados pelo método ver(), definido em A1. Usá-los diretamente é como se não existissem.

private e encapsulamento

Ao definir as variáveis como **private** e os métodos que as consultam e alteram como **public**, conseguimos encapsular o estado do objeto ou da classe. Nesta situação as variáveis não poderão ser alteradas diretamente por métodos de outras classes, ou mesmo por métodos redefinidos nas subclasses. Para o fazer, as outras classes terão que chamar os métodos public da nossa classe que consultam e alteram essas variáveis. Isto garante que apenas os nossos métodos alteram o estado do objeto, o que maximiza a probabilidades de manter o objeto num estado coerente e integro. Com esta técnica consegue-se o **encapsulamento** das variáveis, aumentando o controlo sobre os objetos e a facilidade em alterar o código da classe.

private e objetos da mesma classe

Os objetos da mesma classe podem aceder às variáveis uns dos outros, mesmo que definidas como private. O exemplo abaixo define a classe Private1, com um método que permite a comparação de variáveis entre objetos da mesma classe:

```
package capitulo4;
public class Private1 {
    private int o=2;
    public void saoIguais(Private1 ob2) {
        if (this.o==ob2.o) {
            System.out.println("Sao iguais");
        } else {
            System.out.println("Sao diferentes");
        }
    }
}
```

A classe Private2 permite fazer o teste:

```
package capitulo4;
public class Private2 {
  public static void main (String args [])   {
    Private1 obj1= new Private1();
    Private1 obj2= new Private1();
    obj1.saoIguais(obj2);
  }
}
```

A execução desta classe produz o seguinte resultado:

```
Sao iguais
```

Sumário

- Package;
- Interface;
- Qualificadores de visualização;