

Aula 02

Classes, Objectos e Pacotes

Como funcionam estes mecanismos em Java

Programação II, 2016-2017

v1.12, 20-02-2017

DETI, Universidade de Aveiro

02.1

Objectivos:

- Noção sobre a sintaxe e a construção (sintática) de classes;
- Saber distinguir atributos e métodos de classes;
- Saber o significado prático dos membros estáticos e não estáticos de classes;
- Saber distinguir e implementar invocações internas e externas de membros de classes.
- Compreender o significado sintáctico da visibilidade dos membros de classes.
- Compreender a forma como se inicializam objectos.
- Noções básicas sobre pacotes em Java.

Conteúdo

1	Classes	1
1.1	Novos Contextos de Existência	2
1.2	Objectos	2
1.3	Encapsulamento	4
1.4	Sobreposição (<i>Overloading</i>)	5
1.5	Construtores	6
1.6	Resumo	7
2	Pacotes (<i>Packages</i>)	7

02.2

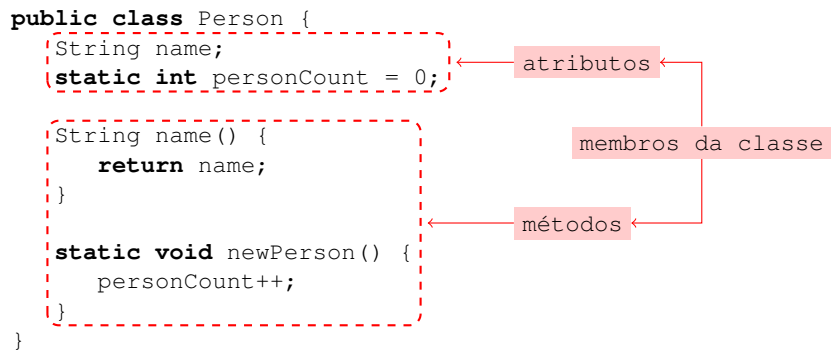
1 Classes

As linguagens orientadas por objectos, como é o caso do Java, introduzem uma nova entidade de linguagem, designada por *classe*.

Classe: Uma *classe* é uma entidade da linguagem que contém *métodos* e *atributos*, podendo também servir para definir novos tipos de dados (com os quais se podem instanciar *objectos*).

Embora seja menos frequente, as classes podem também conter outras classes.

- Dentro da *classe* podemos definir *atributos* (ou *campos*) e *métodos*.
- Os atributos permitem o registo de informação.
- Os métodos permitem a implementação de algoritmos.



1.1 Novos Contextos de Existência

A classe define dois novos contextos de existência:

1. Contexto de classe (ou estático);
2. Contexto de objecto (ou de instância).

O primeiro – contexto de classe – é composto por todos os membros da classe (atributos e métodos) que são estáticos (ou seja, em cuja declaração existe o modificador `static`). Os membros definidos neste contexto têm a sua existência ligada à existência da própria classe.

<pre> public class C { static int a; static void p() { a++; // equivalente a: // C.a++; } static boolean f() { ... } } </pre>	<pre> public class Test { public static void main(String[] args) { C.a = 10; C.p(); if (C.f()) { ... } } } </pre>
---	---

Neste exemplo vemos que, a partir do exterior, a utilização dos membros de classe requer o uso do nome da própria classe. Já a utilização interna à própria classe não necessita dessa qualificação.

É possível também em Java associar à classe um procedimento de inicialização dos seus atributos estáticos. Este procedimento é executado uma única vez assim que a classe passa a ter existência no programa.

1.2 Objectos

No contexto de objecto (ou de instância) têm existência todos os membros da classe (sejam estáticos ou não), mas com a particularidade de que todos os membros não estáticos operarem sobre o estado próprio do objecto (definido como sendo o conjunto dos seus atributos não estáticos). Diferentes objectos da mesma classe operam sobre diferentes estados.

Um objecto necessita de ser explicitamente criado e deixa de existir quando já não pode ser referenciado de dentro do programa (ou seja, quando a ele já não é possível chegar, directa ou indirectamente, fazendo uso de uma qualquer variável existente no programa).

```

public class C {
    int a;

    void p() {
        a++; // equivalente a:
            // this.a++;
    }

    boolean f() {
        ...
    }
}

```

```

public class Test
{
    public static
    void main(String[] args)
    {
        // criar um objecto:
        C o = new C();
        o.a = 10;
        o.p();
        if (o.f())
        {
            ...
        }
        o = null;
        // objecto o deixa de
        // ser referenciável
    }
}

```

02.5

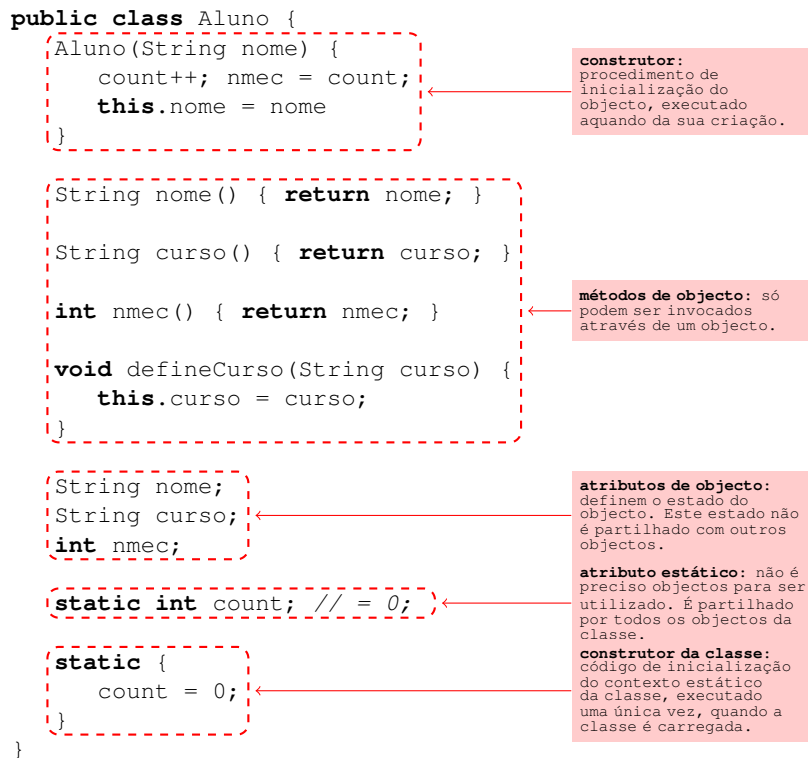
Para aceder aos membros do objecto, vemos que do exterior é necessário usar uma referência para o objecto (existente na variável `o`, no exemplo). Já do interior de um método do objeto, podemos usar a palavra reservada `this` para referir o próprio objeto ou usar apenas o identificador do membro, se não for ambíguo.

Novos Contextos de Existência: Classes e Objectos

- Contexto de classe (*static*):
 - Atributos e métodos de classe existem sempre, haja objectos ou não;
 - Todos os objectos da classe partilham os mesmos atributos de classe;
 - O contexto de execução dos métodos é também sempre o mesmo;
- Contexto de objecto (*non static*):
 - Atributos e métodos só existem enquanto o respectivo objecto existir;
 - Atributos são diferentes para cada objecto;
 - O contexto de execução dos métodos é o contexto do respectivo objecto.
- Se necessário, na mesma classe podem coexistir (e cooperar) membros com diferentes contextos de existência.

02.6

Exemplo de classe



02.7

Invocação de Métodos (Mensagens)

- A invocação de um método pode ser *interna* ou *externa*;
- A invocação externa é sempre efectuada através da *notação de ponto*:

```
myObj.add(25);
deti.abrePorta();
```

- A invocação de um método de um objecto pode ser vista como o envio de uma mensagem (pedido de um serviço) ao objecto: “*DETI, abre a tua porta!*”
 - O receptor da mensagem é o indicado à esquerda do ponto;
 - O tipo de mensagem é o nome do método;
 - Quaisquer outros detalhes que possam ser necessários serão parâmetros;
 - Dentro do método, o objecto receptor funciona como um parâmetro implícito (`this`);
 - `this` é um identificador especial, existente no contexto de objecto, que referencia sempre o próprio objecto.
- O receptor é uma referência para um objecto. No caso de métodos de classe (`static`), o receptor pode ser o nome da classe, e.g.: `String.valueOf()` ou `Math.sqrt()`.
- O acesso a atributos segue regras idênticas.

02.8

1.3 Encapsulamento

- Permite que a classe defina a política de acesso exterior aos seus membros autorizando, ou proibindo, esse acesso;
- Em Java, os modificadores de controlo de acesso que podemos usar são os seguintes:

public - indica que o membro pode ser usado em qualquer classe;

protected - o membro só pode ser usado por *classes derivadas* (conceito estudado noutra disciplina) ou do mesmo *package*;

modificador omisso - o membro só pode ser usado em classes do mesmo *package*;

private - o membro só pode ser usado na própria classe.

```
class X {
    public void publ( ) { /* . . . */ }
    public void pub2( ) { /* . . . */ }
    private void priv1( ) { /* . . . */ }
    private void priv2( ) { /* . . . */ }
    private int i;
    ...
}

public class XUser {
    private X myX = new X();
    public void teste() {
        myX.publ(); // OK!
        // myX.priv1(); Errado!
    }
}
```

- Um método de uma classe tem sempre acesso a toda a informação e a todos os métodos dessa classe.

02.10

Métodos privados

- Uma classe pode dispor de diversos métodos privados que só são internamente utilizados por outros métodos da classe;

```
// exemplo de funções auxiliares numa classe:
class Screen {
    private int row();
    private int col();
    private int remainingSpace();
    ...
};
```

02.11

1.4 Sobreposição (*Overloading*)

- Muitas linguagens requerem que os nomes das funções sejam diferentes – mesmo que conceptualmente se pretenda que executem a mesma acção:

```
void sortArray (Array a);
void sortLista (Lista l);
void sortSet (Set s);
```

- Em Java isto pode ser feito com um único nome:

```
void sort (Array a);
void sort (Lista l);
void sort (Set s);
```

- A distinção faz-se pela assinatura completa da função (assinatura = nome + argumentos);
- Não é possível distinguir funções pelo tipo de valor devolvido (porque poderia gerar situações ambíguas).

02.12

1.5 Construtores

- A inicialização de um objecto pode implicar a inicialização simultânea de diversos atributos;
- Um método especial, o *construtor*, é invocado automaticamente sempre que um novo objecto é criado;
- Os objectos são criados por instanciação através do operador `new`:

```
Carro c1 = new Carro();
```

- O construtor é identificado por ter o nome da classe, e por não ter resultado (nem sequer `void`);
- O construtor pode ser sobreposto (com várias assinaturas) de modo a permitir diferentes formas de inicialização:

```
Carro c2 = new Carro("Ferrari", "430");
```

02.13

- O construtor é chamado apenas uma vez: na criação do objecto;
- É usado para inicializar os atributos do novo objecto de forma a deixá-lo num estado coerente;
- Pode ter parâmetros de entrada;
- Não devolve qualquer resultado;
- Tem sempre o nome da classe.

```
public class Livro {  
    public Livro() {  
        titulo = "Sem titulo";  
    }  
    public Livro(String umTitulo) {  
        titulo = umTitulo;  
    }  
    private String titulo;  
}
```

02.14

Construtor “por omissão”

- Um *construtor por omissão* (*default constructor*) é automaticamente criado pelo compilador caso a classe não especifique nenhum construtor;
- O construtor por omissão não tem parâmetros;

```
class Machine {  
    int i;  
}  
Machine m = new Machine(); // ok
```

- No entanto, se houver pelo menos um construtor especificado na classe, o compilador já não cria o de omissão (nem este pode ser utilizado):

```
class Machine {  
    int i;  
    Machine(int ai) { i= ai; }  
}  
Machine m = new Machine(); // erro!
```

- Para além do construtor, a linguagem Java inicializa todos os atributos antes do construtor.

02.15

1.6 Resumo

O que uma classe pode conter

- A definição de uma classe pode incluir:
 - zero ou mais declarações de *atributos*;
 - zero ou mais definições de *métodos*;
 - zero ou mais *construtores*;
 - zero ou mais *blocos static* (raro);
 - zero ou mais declarações de *classes internas* (raro).
- Esses elementos só podem ocorrer dentro do bloco: `class NomeDaClasse { ... }`

02.16

```
class Point {  
  
    public Point() {...}  
    public Point(double x, double y) {...}  
  
    public void set(double newX, double newY) {...}  
    public void move(double deltaX, double deltaY) {...}  
  
    public double getX() {...}  
    public double getY() {...}  
    public double distanceTo(Point p) {...}  
    public void display() {...}  
  
    private double x;  
    private double y;  
  
}
```

02.17

2 Pacotes (*Packages*)

Espaço de Nomes: *Package*

- Em Java a gestão do espaço de nomes é efectuado através do conceito de *package*;
- Porquê gestão de espaço de nomes?
- Evita conflitos de nomes de classes!
 - Não temos geralmente problemas em distinguir os nomes das classes que implementamos.
 - Mas como garantimos que a nossa classe `Point` não colide com outra que eventualmente possa já existir?

02.18

Instrução *import*

- Utilização:
 - As classes são referenciadas através dos seus nomes absolutos ou utilizando a primitiva `import`;

```
import java.util.Scanner;  
import java.util.*;
```

- As cláusulas `import` devem aparecer sempre antes das declarações de classes;

- Quando escrevemos:

```
import java.util.*;
```

estamos a indicar um caminho para um pacote de classes permitindo usá-las através de nomes simples:

```
Scanner in = new Scanner(System.in);
```

- De outra forma teríamos de escrever:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

02.19

Criar um novo pacote

- Na primeira linha de código:

```
package pt.ua.prog;
```

- garante que a classe pública dessa unidade de compilação (WIO, por exemplo) fará parte do package `pt.ua.prog`.

- O espaço de nomes é baseado numa estrutura de sub-directórios;

- Este pacote vai corresponder a uma entrada de directório: `{ $CLASSPATH } pt/ua/prog/`
 - Boa prática usar uma espécie de *DNS* (endereço Internet) invertido.

- A sua utilização será na forma:

```
pt.ua.prog.WIO.println(...);
```

ou

```
import pt.ua.prog.*;  
WIO.println(...);
```

ou (neste caso em que o método é estático):

```
import static pt.ua.prog.WIO.*;  
println(...);
```

02.20