FICHA 04 – PROGRAMAÇÃO ORIENTADA A OBJECTOS

As unidades fundamentais em Java são os objectos. Os objectos normalmente possuem um conjunto de variáveis (ou estado interno) e um conjunto de métodos (ou comportamentos).

1. Classes

As classes servem para definir tipos de objectos. Objectos de uma mesma classe possuem os mesmos atributos e os mesmos métodos.

```
class Lampada {
  /** A lâmpada está acesa? */
  boolean acesa = false;
  /** Em watts*/
  int potencia;

  Lampada(){
    potencia = 100;
  }
  Lampada(int p) {
    potencia = p;
  }

  /**
   * Liga a lâmpada
   */
  void liga () {
    acesa = true;
  }
  /**
   * Desliga a lâmpada
   */
  void desliga () {
    acesa = false;
  }
}
```

Todos os objectos do tipo **Lampada** (também chamados instâncias da classe **Lampada**) possuem dois atributos ou campos: **acesa** e **potencia**; por omissão as lâmpadas estão apagadas (acesa é inicializada a ***false***). Estes objectos possuem dois

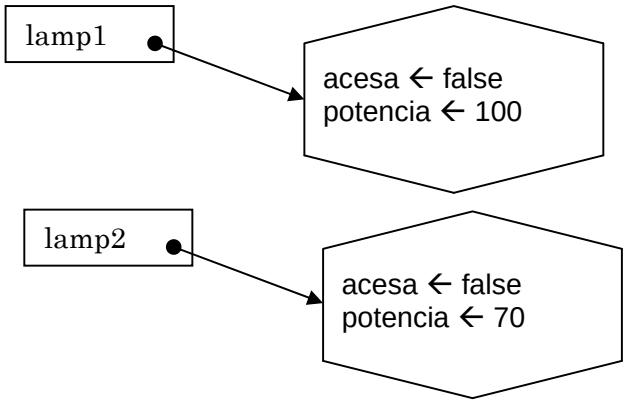
construtores: um construtor sem parâmetros, **Lampada()**, que cria uma lâmpada de 1000 watts, e um construtor em que é possível configurar a potência da lâmpada pretendida **Lampada(int)**. Objectos desta classe respondem ainda a dois métodos: **void liga()** e **void desliga()**.

Para se criarem novas instâncias desta classe basta fazer:

```
(…)
 Lampada lamp1 = new Lampada();
 Lampada lamp2 = new Lampada(70);
(…)
```

Cada uma das instâncias passará então a possuir a sua versão das variáveis **acesa** e **potencia**:

- O objecto **lamp1** vai possuir os campos **acesa** com o valor *false*, e **potencia** com o valor 100;
- O objecto **lamp2** vai ficar com os campos **acesa** também com o valor *false*, e **potencia** com o valor 70;



Para aceder ao valor das variáveis (ou campos) de uma classe basta usar a notação **nomeInstância.nomeCampo**. Para chamar um método usa-se a notação **nomeInstancia.nomeMetodo(listaDeArgumentos)**. Esse método poderá aceder aos campos da instância que está a executar o método:

```
(…)
Lampada lamp1 = new Lampada();
(…)
if (!lamp1.acesa) {
   System.out.println("Vou acender a lamp1");
   lamp1.liga();
   if (lamp1.acesa)
      System.out.println("A lamp1 já está acesa");
   else
      System.out.println("A lamp1 não acendeu");
}
```

O valor do campo **acesa** da **lamp1** é verificado, assumindo que o seu valor é ***false***, então vamos "acender" a lâmpada recorrendo ao método **liga()**; quando verificamos de novo, o valor do campo **acesa** foi alterado.

O resultado da execução do código acima será qualquer coisa como:

```
Vou acender a lamp1

A lamp1 já está acesa
```

A expressão "enviar uma mensagem ao objecto" por vezes é usada com o significado de chamar um método executado pelo objecto.

2. Keyword static

Normalmente quando criamos uma classe é para descrever o aspecto dos objectos dessa classe e o modo como se comportam. Na realidade, só a partir do momento da criação de uma instância é que os seus campos passam a existir, e os seus métodos se tornam disponíveis.

Mas por vezes esta abordagem não é suficiente. Por vezes queremos simplesmente guardar um dado, independentemente do número de objectos que tenham sido criados, ou então necessitamos de um método que não esteja associado a nenhuma instância em particular.

Podemos obter esse funcionamento através da *keyword* ***static***. Quando dizemos que algo (campo ou método) é ***static***, queremos dizer que não deve estar ligado a nenhuma instância em particular. Na realidade, podemos usar esses campos ou métodos mesmo sem criar nenhuma instância da classe respectiva.

Aos campos e métodos prefaciados pela *keyword* ***static*** chama-se, por vezes, "campos da classe" e "métodos da classe", uma vez que estão associados à classe propriamente dita, e não às instâncias.

Para aceder a um campo **static** podemos usar um de dois modos:

NomeClasse.nomeCampo ou

nomeInstancia.nomeCampo

De igual modo, para chamar um método **static** podemos fazer:

NomeClasse.nomeMétodo(argumentos) ou
nomeInstancia.nomeMétodo(argumentos)

Tomemos como exemplo o seguinte código:

```
class Ovelha {
  /**
   * Usado para atribuir um número único a cada uma das ovelhas.
   * Guarda  o número a ser atribuído à próxima ovelha que seja registada.
   */
  static int proxID = 1;

  /** Nome da ovelha */
  String nome;
  /**N.º único que identifica a ovelha na base de dados */
  int id;

  Ovelha(){
    id = proxID;
    proxID++;
  }
  Ovelha(String n) {
    id = proxID++;
    nome = n;
  }

  public String toString() {
    return "nome = " + nome + " número  = " + id;
  }
}
```

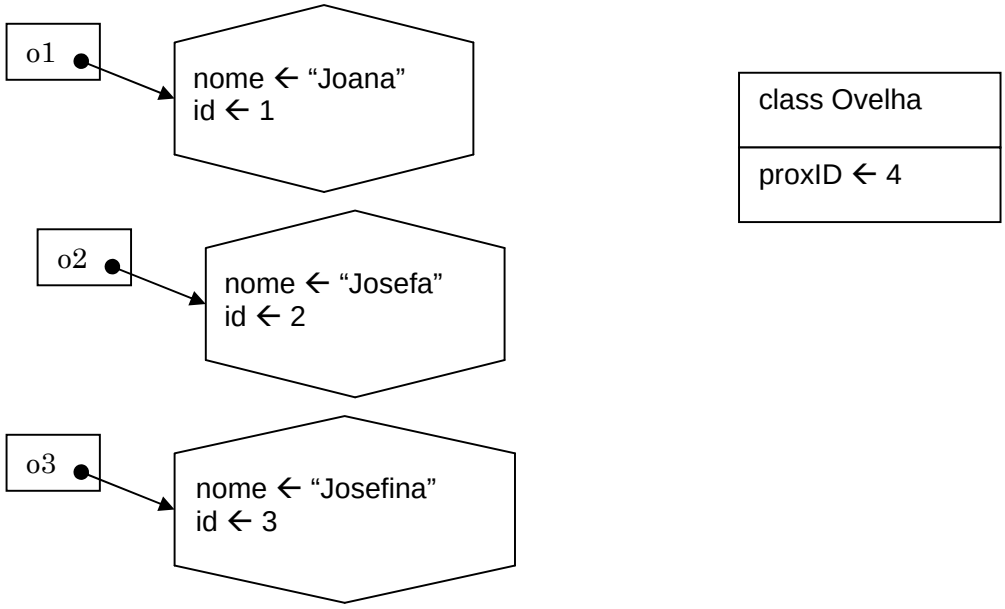
No código acima estamos a criar uma classe Ovelha. Cada ovelha pode possuir um nome e é identificada por um número único. Para assegurar que não existem números repetidos, mantemos um contador na própria classe. Cada vez que uma ovelha é criada o contador é incrementado.

De notar que apenas existe 1 cópia de cada variável de classe (enquanto existe uma cópia por instância de cada variável de instância). Ao executarmos o código seguinte,

```
  //Inicializa o numerador de ovelhas
  Ovelha.proxID = 1;

  Ovelha o1 = new Ovelha("Joana");
  Ovelha o2 = new Ovelha("Josefa");
  Ovelha o3 = new Ovelha("Josefina");
```

ficaríamos na seguinte situação:



O campo da classe **proxID** existe unicamente na classe e não nas instâncias.

A execução do código seguinte:

```
(…)
//Inicializa o numerador de ovelhas
Ovelha.proxID = 1;

Ovelha[] rebanho = {
  new Ovelha("Joana"), new Ovelha("Josefa"), new Ovelha("Josefina")};

for (int i = 0; i < rebanho.length; i++)
  System.out.println("Ovelha " + (i + 1) + ": " + rebanho[i]);
(…)
```

Teria como resultado:

```
Ovelha 1: nome = Joana número  = 1

Ovelha 2: nome = Josefa número  = 2

Ovelha 3: nome = Josefina número  = 3
```

Note que dentro de um método **static**, chamado na forma **NomeClasse.nomeMétodo(argumentos)**, não se pode referir a variáveis ou métodos

de instância, uma vez que o compilador não saberia determinar a que instância pertenceriam as variáveis em questão, ou qual a instância que deveria executar o método. A não ser é claro que tenha sido passada alguma instância ao método, e nesse caso este poderia fazer as tais chamadas, prefaciando-as com o nome da instância passada.

3. Construtores

Para garantir que os vários campos de cada classe sejam inicializados correctamente, o Javatm dispõe de construtores. Assim que um objecto é definido o primeiro código a ser executado (usando o novo objecto) é o código do construtor, garantindo que o objecto nunca seja utilizado sem estar devidamente inicializado.

Um construtor é semelhante a um método cujo nome é o mesmo que o nome da classe. Podem ser definidos um ou mais construtores para a mesma classe, desde que esses construtores tenham parâmetros diferentes (ver exemplo da classe **Lampada**, por exemplo). O Java saberá então qual dos construtores chamar de acordo com os parâmetros que são passados quando se faz a chamada **new**.

Existe um tipo de construtor especial, o construtor por omissão, que consiste num construtor sem argumentos (Por exemplo, **Lampada()**). Caso se crie uma classe e não se definam construtores para a mesma, então o compilador criará automaticamente um construtor por omissão para a classe.

```
 (…)
class Passaro {
   int num;
}
     (…)
     Passaro p = new Passaro();
     System.out.println(p);
     (…)
```

Se definirmos um ou mais construtores para a classe, com ou sem argumentos, então o compilador não criará nenhum construtor por omissão (mesmo que este não tenha sido definido).

```
 (…)
class Passaro {
  int num;

  Passaro(int n) {
    num = n;
  }
}
    (…)
    //Vai dar origem a um erro de compilação
    Passaro p = new Passaro();
    (…)
```

A *keyword* this

Suponha que dentro do código de um método, necessita de ser utilizar o objecto que está a executar o método. Dito de outro modo, necessita de obter um *handle* para o objecto. Pode obter o dito através da *keyword* **this**.

A *keyword* **this** só pode ser usada dentro de métodos e construtores (se bem que nestes assuma um significado ligeiramente diferente), e é um *handle* para o objecto que chamou o método. Este *handle* pode ser usado do mesmo modo que qualquer outro. Por exemplo:

```
class Folha {
  private int i;

  /**
   * Incrementa a variável e devolve o proprio objecto depois de incrementado.
   * Prático para poder fazer chamada em cadeia!
   */
  Folha incrementa () {
    i++;
    return this;
  }

  public String toString() {
    return "i = " + i;
  }

  public static void main(String[] args)
  {
    Folha x = new Folha();
    System.out.println(x.incrementa().incrementa().incrementa());
  }
}
```

Neste caso a própria instância é devolvida para que se possa abreviadamente executar várias operações sobre o mesmo objecto.

Por uma questão de simplicidade e legibilidade, a *keyword* **this** é omitida, a menos que seja inevitável.

Chamar construtores de dentro de construtores

Quando se escrevem vários construtores para a mesma classe torna-se maçador ter de repetir sempre o mesmo código. Para facilitar é possível utilizar a *keyword* **this** para chamar um construtor dentro de outro construtor. Dentro de um construtor a *keyword* **this** pode ser usada para referir um construtor. Os argumentos passados ao **this** são os mesmos que seriam passados ao construtor que queremos evocar. Por exemplo **this()** quer dizer o construtor por omissão da classe em causa.

```
/**
 * Utilização da keyword this para chamar construtores dentro de construtores
 */
class Flor {
  private int numeroPetalas = 0;
  private String s = new String("null");

  //construtores
  Flor(){
    this("olá", 47);
    System.out.println("construtor por omissão, sem argumentos");
  }
  Flor(int petalas){
    numeroPetalas = petalas;
    System.out.println(
      "construtor com um único argumento de entrada inteiro, numeroPetalas = "
       + numeroPetalas);
  }
  Flor(String ss){
    System.out.println(
      "construtor com um único argumento de entrada string, s = " + ss);
    s = ss;
  }
  Flor(String ss, int petalas){
    this(petalas);
    //! this(ss); não pode chamar dois construtores!!
    this.s = ss;  //outro uso para o this
    System.out.println("construtor com dois argumentos de entrada");
    s = ss;
  }

  //utilitários
  public String toString () {
    // Incorrecto! Este tipo de chamada só é válida dentro de construtores
    //! this(11);
    return "numeroPetalas = " + numeroPetalas + " s = " + s;
  }

}
```

O significado da keyword static

Uma vez sabendo o significado da *keyword* **this**, talvez se torne mais aparente o significado de um método de classe (ou **static**).

Pode-se dizer que dentro de um método **static,** chamado do modo usual (**nomeClasse.nomeMétodo(argumentos)**) não se tem acesso ao **this.** A excepção é caso uma instância tenha sido associada ao método, isto é se tenha feito, **nomeInstancia.nomeMétodo(argumentos)**).

Exercícios

1. Ângulos

Implemente uma classe Angulo que represente um angulo, com valores entre 0 e 360 graus i.e 0 <= graus < 360. Cada objecto da classe referida deve possuir um atributo do tipo double: graus. (Use os metodos da classe java.lang.Math).

a) Defina os construtores da classe, nomeadamente:

- Angulo ( )

- Angulo (double).

b) Defina as seguintes operações sobre angulos:

- Angulo adicao (Angulo)

- Angulo subtraccao (Angulo)

c) Escreva um método double radianos(), que transforme um ângulo de graus em radianos.

d) Defina o método boolean equals (Angulo), que determine se dois angulos são iguais.

e) Defina os métodos double sin(), double cos() e double tg() que retorne o número real correspondente ao seno, coseno e tangente do angulo, respectivamente.

f) Defina o método String toString ( ) que permita apresentar o angulo no formato “angulo de n graus”.

g) Faça um pequeno programa que peça ao utilizador dois angulos e os apresente, assim como à sua soma, o seno, coseno e tangente da soma.

2. Fracções

Faça um programa em Java, usando a abordagem de programação orientada a objectos, que lhe permita realizar operações com fracções (somar, subtrair, multiplicar, dividir). Para isso deve definir uma classe Fraccao com os atributos e métodos que considerar convenientes e uma classe Principal onde dadas duas fracções e uma operação proceda à sua realização.

3. Conversor monetário

Crie um programa que faça a conversão de moedas entre cinco unidades monetárias (euros, dólares, libras, francos suíços, yenes). Para isso defina uma classe moeda com informação sobre o tipo de moeda (string) e a quantidade a converter. Utilize-a numa classe Principal que faz repetidamente e enquanto o utilizador desejar conversões entre moedas diferentes.

4. Polígonos (problema para avaliação)

Usando a classe Angulo desenvolvida no problema **1.** escreva um programa que defina uma classe Poligono com n lados (máximo 10) e que verifique para um conjunto de polígonos dado quantos e quais são regulares (um polígono é regular se tiver todos os lados e todos os ângulos iguais). Admita que um polígono é especificado através das coordenadas dos seus vértices.