FICHA 4.1 - OBJETOS E CLASSES

Na **programação orientada a objetos** um **programa** é constituído por um conjunto de entidades fundamentais chamadas **objetos**. Um **objeto** consiste num conjunto de **dados** (**variáveis**) e **ações** (**métodos**) relacionados.

Qualquer objeto pertence a uma **classe**. Uma **classe** define um conjunto de objetos semelhantes ou um **objeto** do ponto de vista genérico, atuando como uma espécie de "molde". Desta definição consta a especificação dos seus **atributos** (**variáveis**) e **comportamento** (**métodos**).

A linguagem **Java** possui uma variedade grande de classes de objetos predefinidas. Estas classes estão organizadas em **packages** (pacotes ou grupos de classes relacionadas).

1. Exemplo

Considere que se pretendia um programa que leia os dados de um conjunto de estudantes (nome e um conjunto de notas), calcule a média obtida por cada estudante e os ordene por ordem decrescente das médias.

O primeiro passo será definir uma nova classe, a classe Estudante, para representar um aluno, que terá como atributos o nome e as notas, e como comportamentos a inicialização, o cálculo e obtenção da média e a escrita dos seus dados.

Terá de haver uma outra classe capaz de guardar o conjunto de estudantes e ordená-los de acordo com a sua média.

Criação da classe Estudante

```
class Estudante {
	//Atributos
	private String nome;
	private int[] notas;

	//Construtor da classe, promove a inicialização dos atributos
	public Estudante() {
		...
	}
	public Estudante(String nomeEst, int[] notasEst){
```

```
            ...
        }
        //Comportamentos
        private float calculaMedia() {
            ...
        }
        public float getMedia(){
            ...
        }
        public void imprimeEstudante(){
            ...
        }
}
```

Todos os objetos do tipo **Estudante** (também chamados instâncias da classe **Estudante**) possuem dois atributos ou campos: **nome** e **notas**. Estes objetos possuem dois construtores: um construtor sem parâmetros, **Estudante ()**, que cria um Estudante (pode, por exemplo, pedir dados ao utilizador para inicializar os atributos ou apenas criar o objeto e inicializar depois); um construtor **Estudante (String nomeEst, int[] notasEst)**, que recebe um parâmetro do tipo String e outro do tipo tabela de inteiros, tipicamente para inicializar os atributos. Objetos desta classe respondem ainda a três métodos: **float calculaMedia()**, **float getMedia()** e **void imprimeEstudante()**.

Para se criarem novas instâncias desta classe basta fazer:

```
Estudante est = new Estudante();
```

ou

```
int[] notas = {...};
Estudante est = new Estudante ("Paulo",notas);
```

Cada uma das instâncias passará então a possuir a sua versão das variáveis **nome** e **notas**.

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:

```
float media = est.getMedia();
```

Na classe Estudante, o método getMedia() poderá ser implementado da seguinte forma:

```
//Método de acesso externo à média
       public float getMedia() {
              return calculaMedia();
       }
```

Por sua vez, o método calculaMedia() poderá ser implementado como se segue:

```
       //Método para cálculo da média
       private float calculaMedia() {
              float media = -1;
              if (notas.length > 0) {
                  float soma = 0;
                  for (int i = 0; i < notas.length; i++) {
                            soma += notas[i];
                  }
                  media = soma/notas.length;
              }
              return media;
       }
```

Como se pode verificar, o método getMedia() chama o método calculaMedia() e devolve o valor fornecido por este. O método calculaMedia(), lê os valores do atributo notas (tabela) da instância (objeto), calcula a sua média e devolve o resultado.

2. Construtores

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

Um construtor é semelhante a um método cujo nome é o mesmo que o nome da classe e não devolve qualquer valor (nem *void*). Podem ser definidos um ou mais construtores para a mesma classe, desde que esses construtores tenham parâmetros diferentes (ver exemplo da classe **Estudante**, 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, **Estudante()**). Caso se crie uma classe e

não se definam construtores, então o compilador criará automaticamente um construtor por omissão para a classe (neste caso, nem a inicialização dos atributos é garantida).

3. *Keyword this*

Suponha que dentro do código de um método, necessita de utilizar o objeto que está a executar o método. Dito de outro modo, necessita de obter um *handle* para o objeto. 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 objeto que chamou o método. Este *handle* pode ser usado do mesmo modo que qualquer outro. Por exemplo:

```
public Estudante() {
	System.out.println("A criar um objeto Estudante!");
}

public Estudante(String nome, int[] notas) {
	// Chama o construtor por omissão
	this();
	// Cria um objeto nome com a string recebida
	this.nome = new String (nome);
	//Cria o vetor notas com a dimensão necessária
	this.notas = new int[notas.length];
	System.arraycopy(notas, 0, this.notas, 0, notas.length);
}
```

Neste exemplo, o termo **this** é utilizado em duas situações diferentes:

- **this()**: nesta situação, está a ser utilizado para chamar o construtor por omissão (sem parâmetros) do próprio objeto. A chamada de um construtor dentro de outro construtor só pode ser efetuada se for a primeira instrução do último construtor.

- **this.<atributo>:** serve para aceder a um atributo do objeto, quando o construtor ou método em causa tem um parâmetro de entrada ou variável local com o mesmo nome, distinguindo-os desta forma.

4. *Keyword static*

Normalmente quando criamos uma classe é para descrever as características dos objetos dessa classe (atributos e comportamentos). 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 objetos 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 respetiva.

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)

Já fizemos uso deste tipo de métodos (static), por exemplo quando utilizámos o método random() da classe Math – **Math.random()** – ou o método arrayCopy() da classe System – **System.arrayCopy()**.

Conforme referido, o termo **static** pode ser utilizado não apenas com métodos de classe mas também com atributos de classe.

Por exemplo:

```
class Estudante {
	static int nEstudantes = 0;	// Atributo de classe para contar
// a quantidade de objetos criados
	//Atributos
	private String nome;
	private int[] notas;

	//Construtor da classe, promove a inicialização dos atributos
	public Estudante() {
		nEstudantes++;
		...
	}
...
}
```

Desta forma, cada vez que é criado um objeto, o atributo de classe (static) nEstudantes é incrementado.

5. Encapsulamento de dados

Um objeto deve ser autocontido, ou seja, qualquer alteração do seu estado (das suas variáveis) deve ser provocada apenas pelos seus métodos. Desta forma, o acesso aos atributos deve ser realizado através de **métodos de acesso**, tanto na obtenção (***get****) dos valores como na sua alteração (***set****). De salientar que os atributos devem ser classificados como **private** de forma a garantir o encapsulamento dos dados. De seguida apresentam-se exemplos de métodos de acesso (*get** e *set**) relativos ao atributo *nome* da classe Estudante.

```
public String getNome (){
	return nome;
}
```

```
public void setNome (String nome){
       this.nome = new String (nome);
}
```

6. Método toString()

A escrita dos dados de um objeto no ecrã pode ser feita com recurso a um método que escreve os valores de todos os atributos.

No entanto, a apresentação desses dados pode diferir de programa para programa, e da interface utilizada para o efeito (linha de comandos, interface gráfica, etc.).

Por este motivo, é mais frequente recorrer à implementação do seguinte método:

```
public String toString ();
```

Este método é automaticamente herdado por todas as classes, devendo, no entanto, ser (re)implementado em cada classe, correspondendo a uma sobreposição ("override") do código herdado. Gera e devolve uma String correspondente ao descritivo completo do objeto.

Na classe Estudante, o método toString() poderia ser implementado da seguinte forma:

```
@Override
    public String toString (){
        String s = new String ("As notas de "+nome+" são: ");
        for (int i = 0; i < notas.length; i++) {
            s = s + notas[i] + " ";
        }
        System.out.println();
        s += "\n";
        s += "A média é "+calculaMedia();
        return s;
    }
```

Assim, a escrita dos dados de um objeto da classe Estudante:

```
Estudante est = new Estudante();
```

Pode ser efetuada através de:

```
System.out.println(est.toString());
```

Ou, simplesmente:

```
System.out.println(est);
```

7. A classe ArrayList

Um ArrayList representa o conceito de tabela dinâmica cuja dimensão pode variar durante a execução de uma aplicação. É a estrutura por excelência para armazenamento dinâmico de objetos.

Para criar um ArrayList é necessário definir qual o tipo dos seus elementos (têm que ser objetos). Por exemplo, para criar uma lista de Strings pode usar-se:

```
ArrayList<String> frases = new ArrayList<String>();
```

De seguida, apresentam-se alguns dos **métodos** desta classe:

Método	Funcionalidade
`void add(E o)`	Adiciona o objeto *o* (do tipo E) no fim do ArrayList
`add(int indice, E o)`	Insere o objeto *o* na posição do ArrayList indicada por *índice*
`void remove(int indice)`	Remove o objeto armazenado na posição dada por *índice*
`void clear()`	Elimina todos os elementos do ArrayList
`int indexOf(E o)`	Devolve o índice da primeira posição onde se encontra um objeto igual a *o* ou –1 se não o encontrar
`E get(int indice)`	Devolve o elemento que ocupa a posição definida por *índice*
`void set(int indice, E o)`	Substitui o elemento da posição dada por *índice* pelo objeto *o*
`List<E> subList(int a, int b)`	Devolve um ArrayList formado pelos elementos colocados entre as posições *a* e *b*-1
`boolean isEmpty()`	Indica se o ArrayList está vazio
`int size()`	Devolve o número de elementos do ArrayList

Exemplos de iteração sobre os elementos de um `ArrayList<String>`

```
for (int i = 0; i < frases.size(); ++i)
      System.out.println(frases.get(i));

for (String frase: frases)
      System.out.println(frase);

for (Iterator<String> iter =frases.iterator();iter.hasNext();)
      System.out.println(iter.next());
```

7. Javadoc

O Javadoc é uma ferramenta utilizada para gerar documentação, em formato HTML, para API produzidas em Java.

As páginas em HTML são produzidas a partir dos comentários introduzidos no código fonte, no formato Javadoc.

Os comentários em Javadoc podem conter várias linhas:

- A primeira linha inicia com `/**`
- Cada linha intermédia inicia com `*`
- A última linha termina com `*/`

Para iniciar um comentário no seu código, no formato Javadoc, basta escrever `/**` e clicar em `<Enter>`.

Além do texto simples, podem ser utilizadas "tags" Javadoc, que permitem o preenchimento automático de campos das páginas HTML produzidas por esta ferramenta.

Exemplos de "tags":

`@author` (apenas em classes e interfaces - obrigatório, por convenção)

`@version` (apenas em classes e interfaces - obrigatório, por convenção)

`@param` (apenas em métodos e construtores)

`@return` (apenas em métodos)

`@exception` (`@throws` sinónimo introduzido no Javadoc 1.2)

`@see`

`@since`

`@serial` (ou `@serialField` ou `@serialData`)

`@deprecated` (para ver como e quando descontinuar APIs)

Mais informação sobre Javadoc pode ser encontrada nos materiais da disciplina.

Exercícios

1. Data

Implemente uma classe `Data`, que deve reunir os valores do dia, mês e ano, constituintes de cada data a representar.

a) Defina um construtor com o seguinte formato: `Data (int, int, int).`
b) Defina o método `String toString()` que permita apresentar a data no formato "[dia] de [mês por extenso] de [ano]".
c) Faça um pequeno programa que inicialize variáveis adequadas com o dia, mês e ano correspondentes a uma data, construa o respetivo objeto Data e o apresente no ecrã, com recurso ao seu método `toString()`.

2. Ângulos

Implemente uma classe `Angulo` que represente um ângulo, com valores entre 0 e 360 graus, i.e 0 <= graus < 360. Cada objeto da classe referida deve possuir um atributo do tipo `double: graus` (use os métodos da classe java.lang.Math).

a) Defina os construtores da classe, nomeadamente:

- `Angulo ( )`
- `Angulo (double).`

b) Defina o método `String toString()` que permita apresentar o ângulo no formato "ângulo de n graus".
c) Faça um pequeno programa que inicialize uma variável com o valor de um ângulo (em graus), construa o respetivo objeto Angulo e o apresente no ecrã, com recurso ao seu método `toString()`. Teste o seu programa.
d) Defina as seguintes operações sobre ângulos:

- `Angulo adicao (Angulo)`

- `Angulo subtracao (Angulo)`

e) Altere o seu programa, de forma a criar dois ângulos e que os apresente no ecrã, assim como à sua soma e subtração. Recorra à classe Angulo definida nas alíneas anteriores.
f) Escreva um método `double radianos()`, que devolva o valor do respetivo ângulo em radianos.
g) Defina o método `boolean equals (Angulo)`, que determine se o respetivo ângulo é igual ao fornecido.
h) Defina os métodos `double sin()`, `double cos()` e `double tg()` que retorne o número real correspondente ao seno, coseno e tangente do ângulo, respetivamente.
i) Altere o seu programa de forma a criar dois ângulos e indicar se são iguais, assim como o seno, coseno e tangente da sua soma. Recorra à classe Angulo definida nas alíneas anteriores.
j) Adicione os elementos necessários e produza a documentação Javadoc completa do seu Projeto.

3. Frações

Faça um programa em Java, usando a abordagem de programação orientada a objetos, que lhe permita realizar operações com frações (somar, subtrair, multiplicar, dividir). Para isso deve definir uma classe `Fracao` com os atributos e métodos que considerar convenientes e uma classe principal `Fracoes` onde implementará o programa que, dadas duas frações e uma operação, proceda à sua realização e apresente o resultado.

4. Polígonos

Utilizando a classe `Angulo` desenvolvida no problema **2.**, escreva um programa que defina uma classe `Poligono` com n lados (máximo 10) e que verifique se é, ou não, regular (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, sendo necessário implementar uma classe `Vértice`.

5. Contactos

Defina uma classe `Contacto` que implemente:

a) o construtor e os campos necessários para conter a informação do contacto

```
Contacto(String nome, String email, String morada, String telefone)
```

Defina uma classe `ListaDeContactos` que implemente:

b) o construtor `ListaDeContactos(int size)`, em que `size` é o número máximo de contactos que podem existir na lista.
c) o método `Contacto pesquisa(String nomeAPesquisar)` que permita pesquisar um `Contacto` na `ListaDeContactos` por nome e devolva o objeto pesquisado, se este for encontrado, ou `null`, se não for; a pesquisa deve ignorar diferenças entre maiúsculas e minúsculas;
d) o método `Contacto remove(String nome)` que permita eliminar um `Contacto` associado ao nome passado por parámetro da `ListaDeContactos`; se o contacto existir devolve o Contacto eliminado e se não existir o método devolve `null`.
e) um método que adicione um novo `Contacto` à `ListaDeContactos` e devolva uma referência para o `Contacto` criado; se não existir espaço na lista o método devolve `null`

```
Contacto insere(String nome, String morada, String telefone)
```

f) um método quer permita redimensionar o tamanho da lista de contactos; o novo tamanho será igual à soma da dimensão atual com o valor indicado como parâmetro

```
void redimensiona(int incremento)
```

g) Por último, implemente um programa que permita testar as funcionalidades das classes criadas. Utilize valores previamente definidos, atribuídos a variáveis adequadas a cada um, de forma a não depender da utilização de dados inseridos pelo utilizador.

6. Tarefas

Desenvolva um programa em Java que permita gerir a lista de tarefas de um utilizador. Inclua os elementos necessários e produza a documentação Javadoc completa do seu Projeto.

a) Defina uma classe com os dados relativos a uma tarefa, sabendo que as tarefas podem ter três níveis de prioridade (*baixa*, *normal* – valor por omissão, *alta*), duas categorias (*pessoal*, *trabalho* – valor por omissão), e podem estar num de três estados (*não realizada* – valor por omissão, *em realização*, *realizada*). Além das características indicadas, cada tarefa tem um título e um campo opcional com a data limite para sua realização.

b) Defina uma classe para gerir as tarefas de um utilizador. Desenvolva os métodos que considere relevantes à gestão de tarefas como, por exemplo, os métodos para adicionar e remover uma tarefa.

c) Desenvolva um método que devolva todas as tarefas em atraso.

d) Desenvolva um método que apague todas as tarefas que já foram realizadas.

e) Desenvolva um programa que permita testar todas as funcionalidades das classes criadas.

7. Liga Portuguesa de Futebol

Desenvolva um programa em Java com informação sobre a primeira liga de futebol portuguesa. Inclua os elementos necessários e produza a documentação Javadoc completa do seu Projeto.

a) Defina uma classe com os dados relativos a um jogador, incluindo nome, data de nascimento, e posição em que joga.
b) Defina uma classe com os dados relativos a uma equipa, incluindo nome da equipa, jogadores, e nome do treinador.
c) Defina uma classe com os dados relativos a um jogo, incluindo as equipas que se encontraram, em que campo, o resultado final, e tempo de posse de bola por cada equipa. Para simplificar, assuma que todos os jogadores de uma equipa participam no jogo.
d) Defina uma classe com informação estatística de uma equipa num jogo: número de golos marcados, número de golos sofridos, número de passes total, número de passes corretos, outros que considere pertinentes. Adicione essa classe ao seu desenho da aplicação.
e) Defina uma classe com os dados relativos à liga, incluindo todos os jogos realizados e o quadro de pontuação da primeira liga. Considere que uma época tem 32 jogos.
f) Desenvolva um método que atualize os objetos relevantes na sequência de um jogo. Simule 32 jogos chamando esse método 32 vezes.
g) Desenvolva um método que permita mostrar estatísticas cumulativas para as equipas ao fim de todos os jogos do campeonato.
h) Desenvolva um programa que permita testar todas as funcionalidades das classes e métodos criados.

Nota: Tenha em atenção que cada um dos métodos descritos acima deve ser incluído na classe apropriada.

9. Estatística

A partir da classe `ArrayList<E>` desenvolva a classe `ListStats` para armazenar valores inteiros (classe `Integer`), contendo as seguintes operações:

a) Adicionar um valor à lista;
b) Retirar um elemento da lista: `boolean remove(int n);`
c) Valor máximo da lista;
d) Valor mínimo da lista;
e) Número de ocorrências de um valor;
f) Dimensão máxima da lista (desde a sua criação);
g) Somatório dos valores da lista;
h) Interseção de duas listas: `ListStats intersecao(ListStats s);`
i) Faça um programa que permita testar todas as funcionalidades.

10. Turma

Desenvolva uma aplicação que crie um ArrayList com os dados das avaliações dos alunos de uma turma. Cada elemento da lista deve conter a seguinte informação: Nome, N.º e Nota (0 a 20).

a) Crie uma sublista da inicial, correspondente aos alunos reprovados (nota inferior a 9.5);
b) Elimine da lista inicial todos os elementos da lista de reprovados;
c) Imprima o conteúdo de todas as listas obtidas, pela ordem em que os respetivos elementos se encontram.

11. Gestão de contactos

Pretende-se simular um sistema de gestão de contactos. Para isso, desenvolva uma aplicação seguindo os passos seguintes:

a) Implemente uma classe “Contacto” composta pelos seguintes atributos: Nome, Idade, Profissão, Telefone, Email. Esta classe deve, ainda, conter um construtor que recebe parâmetros de inicialização de todos os atributos, e um método String toString(), que devolve uma String com o seguinte formato:

"<Nome>, <Idade> anos de idade, <Profissão>, telefone n.º <Telefone>, e-mail: <Email>";

b) Implemente uma classe GestaoContactos, que deve conter três ArrayList: Familia, Amigos e Profissional. Estes ArrayList destinam-se, respetivamente, a contactos de familiares, amigos e de âmbito profissional;
c) Implemente um método adicionaContacto, que pede informações ao utilizador relativas a um contacto a criar, e o adiciona ao ArrayList respetivo;
d) Implemente um método eliminaContacto, que pede ao utilizador o nome do contacto e o elimina do ArrayList em que este se encontrar;
e) Implemente um método listaContactos, que recebe um argumento correspondente ao ArrayList pretendida e escreve no ecrã todos os contactos dessa lista;
f) Implemente um método String toString() que escreva no ecrã o conteúdo de todos os ArrayList;
g) Implemente o método maisVelho(...), que devolve o contacto mais velho da lista correspondente ao argumento que este recebe;
h) Implemente, o método maisNovo(...);
i) Implemente um método ArrayList profissionaisAmigos(), que devolve um ArrayList com todos os contactos profissionais que também fazem parte da lista de amigos;
j) Implemente um método ArrayList escalaoEtario(int idadeMin, int idadeMax) que devolva um ArrayList contendo os contactos das pessoas com idades compreendidas entre as idades mínima e máxima (em anos) correspondentes aos argumentos recebidos;
k) Crie uma aplicação que permita testar as funcionalidades descritas;
l) Introduza todos os elementos necessários no seu Código e produza a respetiva documentação com recurso à ferramenta Javadoc.

12. Clínica

A clínica DEIMED pretende desenvolver uma aplicação para marcação de consultas. Esta aplicação deverá possuir os dados dos pacientes (nome, idade, número do paciente), dados dos médicos (nome, especialidade, número do médico) e os tipos de

consultas que a clínica oferece. Uma consulta tem informação sobre: data, hora, especialidade, médico e paciente. A aplicação deverá ser capaz de:

a) Marcar uma consulta;
b) Remarcar uma consulta;
c) Eliminar uma consulta marcada;
d) Listar todos os pacientes com consultas marcadas num determinado dia por tipo de consulta (Por exemplo oftalmologia);
e) Listar pacientes com consultas feitas;
f) Listas consultas disponíveis na clínica.

13. Rotas

A rede de autocarros de uma cidade é constituída por um conjunto de rotas. Cada rota é definida por um conjunto de paragens. Em cada paragem pode parar mais do que um autocarro. Pretende-se que, usando ArrayList, desenvolva uma aplicação para:

a) Inserir novas rotas;
b) Inserir novas paragens, tipicamente em zonas novas da cidade;
c) Determinar se entre duas quaisquer paragens da cidade há uma rota de autocarro direta;
d) Determinar se entre duas quaisquer paragens da cidade há a possibilidade de, com uma mudança de rota apenas, fazer o trajeto. Deve indicar uma rota possível ou apresentar uma mensagem elucidativa, caso não seja possível;
e) Mostrar todas as rotas da rede de autocarros.