

# Laboratórios de Informática III (LI3)

LEI – 2º ano – 2º semestre – Universidade do Minho – 2018/2019

F. Mário Martins

Abril de 2019

## Projecto de Java

### GestVendas

## DESENVOLVIMENTO DE UMA APLICAÇÃO DESKTOP USANDO O MODELO MVC PARA A GESTÃO DAS VENDAS DE UMA CADEIA DE DISTRIBUIÇÃO

### 1.- Introdução.

O projecto de Java da disciplina de LI3 tem por objectivo fundamental ajudar à consolidação experimental dos conhecimentos teóricos e práticos adquiridos na disciplina de Programação Orientada aos Objectos, procurando-se ainda capitalizar o mais possível do trabalho e dos resultados obtidos no projecto de C.

Pretende-se agora a criação de uma aplicação de uma **aplicação Desktop** em Java, baseada na utilização das interfaces e das colecções de JCF (“Java Collections Framework”), cujo objectivo é a realização de consultas interactivas de informações relativas à **gestão básica de uma cadeia de distribuição** (tema do projecto de C já concluído).

### 2.- Pré-Requisitos.

O projecto de Java tem como fontes de dados os mesmos ficheiros de texto, usados no projecto C, pelo que a estrutura de cada linha de ficheiro de texto será a mesma.

### 3.- Requisitos da aplicação a desenvolver.

Pretende-se desenvolver uma aplicação desktop Java que seja capaz de, antes de mais, ler e armazenar em estruturas de dados (colecções de Java) adequadas as informações dos vários ficheiros, para que, posteriormente, possam ser realizadas diversas consultas (“queries”), algumas estatísticas e alguns testes de “performance”.

A classe agregadora de todo o projecto de JAVA será a classe **GestVendas** e a aplicação final, que seguirá o padrão MVC, designar-se-á por **GestVendasAppMVC**.

A concepção da aplicação deverá seguir os **princípios da programação com interfaces**, ou seja, criando-se antes das classes as APIs, assim visando tornar a aplicação mais genérica e flexível.

Deverá ser criado um **Catálogo de Clientes** (todos os códigos destes), um **Catálogo de Produtos** (códigos dos produtos disponíveis), uma **Facturacao** (relacionando produtos e suas vendas) e um registo de todas as compras realizadas em cada filial, por quem e quando, que designaremos por **Filial**. Devem ser consideradas inicialmente 3 filiais, identificadas por 1, 2 e 3.

Cada uma destas subestruturas de informação mantém as características e os requisitos que foram apresentados no projecto de C.

O desenho e a implementação do código da aplicação deverão ter em atenção que o mesmo deverá ter dois grandes grupos de funcionalidades (correspondentes a 70% do valor do trabalho):

- a) **Leitura e validação de dados de memória secundária, a partir de ficheiros de texto**, e população das estruturas de dados em memória central; Gravação da estrutura de dados (instância da classe **GestVendas**) em ficheiro de objectos;
- b) **Queries**: operações de consulta sobre as estruturas de dados;

Complementarmente, equivalendo a 30% do valor do trabalho, pretende-se que seja desenvolvido um conjunto de pequenos programas (ou um que realize a globalidade dos testes), independentes da aplicação **GestVendasAppMVC**, que realizem a funcionalidade designada por:

- c) **Medidas de performance** do código e das estruturas de dados.

### **3.1.- Leitura, população das estruturas e persistência de dados.**

O programa deverá poder ler os ficheiros de texto a qualquer momento e carregar os respectivos dados para memória. Na primeira execução do programa a realização desta operação é obrigatória. O programa deverá cf. indicação do utilizador, ler um dos ficheiros de texto contendo as informações referentes às vendas registadas (cf. **Vendas\_1M.txt**, **Vendas\_3M.txt** ou **Vendas\_5M.txt**) bem como os ficheiros **Clientes.txt** e **Produtos.txt**.

A qualquer momento deverá estar disponível uma opção que permita ao utilizador gravar toda a estrutura de dados de forma persistente usando **ObjectStreams**, criando por omissão o ficheiro **gestVendas.dat** ou um outro se indicado pelo utilizador.

A qualquer momento também deverá o utilizador poder carregar os dados a partir de uma **ObjectStream** de nome dado, repopulando assim toda a informação da estrutura de dados até então existente em memória. Tal poderá ser útil para evitar múltiplas validações.

### 3.2.- Estatística e queries interactivas.

Sendo inúmeras as possíveis informações úteis a extrair da estrutura de dados, elas deverão ser agrupadas para o utilizador da seguinte forma:

#### 3.2.1.- Consultas estatísticas.

1.1.- Apresenta ao utilizador **os dados referentes ao último ficheiro de vendas lido**, designadamente, nome do ficheiro, número total de registos de venda errados, número total de produtos, total de diferentes produtos comprados, total de não comprados, número total de clientes e total dos que realizaram compras, total de clientes que nada compraram, total de compras de valor total igual a 0.0 e facturação total.

1.2.- Apresenta em ecrã ao utilizador os números gerais respeitantes aos dados actuais **já registados nas estruturas**, designadamente:

- Número total de **compras** por mês (não é a facturação);
- **Facturação** total por mês (valor total das compras/vendas) para cada filial e o valor total global;
- Número de distintos clientes que compraram em cada mês (não interessa quantas vezes o cliente comprou) filial a filial;

#### 3.2.2.- Consultas interactivas.

1. Lista ordenada alfabeticamente com os códigos dos produtos nunca comprados e o seu respectivo total;
2. Dado um mês válido, determinar o número total global de **vendas** realizadas e o número total de clientes distintos que as fizeram; Fazer o mesmo mas para cada uma das filiais;
3. Dado um código de cliente, determinar, para cada mês, quantas compras fez, quantos produtos distintos comprou e quanto gastou no total. ;
4. Dado o código de um produto existente, determinar, mês a mês, quantas vezes foi comprado, por quantos clientes diferentes e o total facturado;

5. Dado o código de um cliente determinar a lista de códigos de produtos que mais comprou (e quantos), ordenada por ordem decrescente de quantidade e, para quantidades iguais, por ordem alfabética dos códigos;
6. Determinar o conjunto dos X produtos mais vendidos em todo o ano (em número de unidades vendidas) indicando o número total de distintos clientes que o compraram (X é um inteiro dado pelo utilizador);
7. Determinar, para cada filial, a lista dos três maiores compradores em termos de dinheiro facturado;
8. Determinar os códigos dos X clientes (sendo X dado pelo utilizador) que compraram mais produtos diferentes (não interessa a quantidade nem o valor), indicando quantos, sendo o critério de ordenação a ordem decrescente do número de produtos;
9. Dado o código de um produto que deve existir, determinar o conjunto dos X clientes que mais o compraram e, para cada um, qual o valor gasto (ordenação cf. 5);
10. Determinar mês a mês, e para cada mês filial a filial, a facturação total com cada produto.

A criação das consultas deve ser realizada de forma suficientemente estruturada de modo a que se torne simples alterar o identificador e texto da mesma no menu de consultas e invocar o método associado respectivo.

Todas as *queries* realizadas devem, antes mesmo da apresentação dos resultados, e qualquer que seja a forma como os resultados finais são apresentados, indicar ao utilizador os tempos de execução usando o método `long System.nanoTime()`; que mede diferenças de tempo em nano segundos e que devem ser convertidas para segundos e milisegundos para apresentação (usar a classe `Crono`).

### 3.3.- Medidas de performance (30 %).

Pretende-se realizar alguns testes de “performance” meramente experimentalistas e apenas com o intuito de introduzir a ideia, ainda não muito interiorizada por razões óbvias, de que os programas são entidades que possuem alguns atributos que são mensuráveis e cuja análise pode ser útil (cf. *testing*, *profiling* e *benchmarking*).

Estes testes podem e devem ser realizados fora do contexto do programa anteriormente desenvolvido (não farão parte da funcionalidade do programa) usando as partes do código e as estruturas de dados necessárias para cada teste.

Assim, como requisito complementar do projecto, e valendo 30% da nota final do mesmo, pede-se aos alunos que realizem os seguintes testes e apresentem os resultados dos mesmos da forma mais simples e ilustrativa possível (ver anexo):

- 1.- Tempos de leitura (sem *parsing*) do ficheiro de base, **Vendas\_1M.txt**, usando as classes **BufferedReader()** e **Files**; Usar em seguida os ficheiros **Vendas\_3M.txt** e **Vendas\_5M.txt** e fazer novas medições de tempos;
- 2.- Realizar os mesmos testes mas, agora, incluindo o tempo de *parsing*, ou seja, de separação dos campos da linha e criação das instâncias da classe que representará uma **Venda**. Apresente os resultados sob a forma de uma tabela e também de forma gráfica.
- 3.- Realizar os mesmos testes mas com parsing e validação;
- 4.- Sendo para todos evidente que a estrutura de dados do projecto se vai basear na utilização de colecções do tipo **Map<K, V>**, bem como na utilização de colecções que são do tipo **Set<E>** e **List<E>**, torna-se importante comparar as performances de algumas das queries mais complexas que foram pedidas (de 5 a 9).

Para tal, deve usar-se **HashMap<>** onde se usou **TreeMap<>** ou vice-versa, usar **HashSet<>** ou **LinkedHashSet<>** onde se usou **TreeSet<>** e vice-versa, e usar **Vector<>** onde se usou **ArrayList<>**.

**Nota:** *Estes testes aparentemente complexos e trabalhosos não o são de facto dado que a substituição de uma colecção por outra do mesmo tipo preserva a linguagem, ou seja, por terem a mesma API os métodos são os mesmos. Assim, apenas há que realizar find/replace nas declarações.*

*Por outro lado, se programou os seus métodos usando interfaces e não classes concretas para definir os tipos dos parâmetros de entrada e dos resultados nada será alterado no código dos métodos (porquê?) !!*

- 4.- Caso se verifique que os alunos possuem algum conhecimento sedimentado sobre as **expressões lambda** e as **streams de Java 8**, será proposto um exercício adicional de medição de performance comparativa para 3 operações relevantes a indicar, usando uma **Stream<Venda>** e o ficheiro **Vendas\_5M.txt**, usando mesmo o paralelismo intrínseco da JVM8.

#### 4.- Apresentação do projecto e Relatório.

O projecto será submetido por via electrónica sob a forma de uma pasta em formato zip/rar do projecto em **BlueJ** ou outros IDEs como **NetBeans**, **Eclipse** ou **IntelliJ**. O código Java avaliado será o código que foi submetido.

O relatório do projecto de Java deverá ter a estrutura já conhecida, sendo de salientar agora a importância do **diagrama de classes**, do **desenho da estrutura de dados** usada e da **apresentação dos resultados dos testes**. Ainda que não seja para incluir no relatório, gere o **javadoc** do seu projecto e inclua-o na pasta final do projecto.

O relatório será entregue aquando da apresentação presencial do projecto e servirá de guião para a avaliação do mesmo.

F. Mário Martins

---

#### Bibliografia JAVA



#### ANEXO

#### QUESTÕES E RECOMENDAÇÕES GERAIS

✂ Este projecto de Java quase não tem polimorfismo, o que é muito mau. Mas é apenas a continuação de um projecto em C.

✎ Algumas das consultas devolvem colecções de “pares de coisas” ou mesmo triplos. Por exemplo, pares de (**código produto, nº de clientes**). É portanto aconselhável criar classes auxiliares que representem tais pares (que são muito importantes para as consultas). Por exemplo, a classe **ParProdNumClis** poderá representar os pares anteriores.

Como alternativa, para pares temporários que não “mereçam” uma classe própria, a classe de JAVA **SimpleEntry<F,S>** (melhor que **Entry<>**) e os seus métodos simples resolvem facilmente tal problema.

✎ Instâncias de classes como **ParProdNumClis** poderão ter que ser guardadas em colecções como **HashSet<T>**. Porém, dado que pretendemos na maioria dos casos ter os resultados ordenados por um dado critério, o mais provável é que tais pares devam ser guardados em **TreeSet<T>**.

Tal implicará de imediato a criação de uma ou mais classes que implementem a interface **Comparator<T>** respectiva, uma por cada algoritmo especial de ordenação dos pares, cf:

```
public class CompParesProdClientes implements Comparator<ParProdClis>,
                                           Serializable {
```

Expressões lambda podem aqui ser muito úteis e simplificar muito a criação destes **Comparator<T>** porque nos permitem escrever código muito simples que de imediato pode ser usado no construtor do **TreeSet<T>**, cf. o exemplo:

```
Comparator<ParClienteTotalComprado> compCompras =
    (p1, p2) -> {
        if(p1.getTotalComprado() > p2.getTotalComprado()) return -1;
        if(p1.getTotalComprado() < p2.getTotalComprado()) return 1;
        else return 0;
    };

TreeSet<ParClienteTotalComprado> pares = new TreeSet<>(compCompras);
```

✎ No exemplo anterior procura-se também chamar a atenção para o facto de que, por omissão devemos declarar todas as nossas classes como **Serializable** já que, a qualquer momento, podemos pretender guardar instâncias suas em **ObjectStreams**.

✎ Todos os alunos que já usam Java7 podem beneficiar da designada **notação do diamante** (*diamond notation*) que se baseia no **operador diamante** (cf. **<>**).

Desde Java7, as declarações de colecções podem ser simplificadas na sua inicialização pois o compilador de Java7 faz inferência de tipos. Assim, usando exemplos,

Em vez de `ArrayList<String> nomes = new ArrayList<String>();`  
pode escrever-se apenas `ArrayList<String> nomes = new ArrayList<>();`

Em vez de `TreeMap<Ponto2D, String> pmap =  
new TreeMap<Ponto2D, String>();`  
pode escrever-se apenas `TreeMap<Ponto2D, String> pmap = new TreeMap<>();`  
NOTA FINAL: Nunca esquecer o `<>` !!

A declaração ERRADA `ArrayList<String> nomes = new ArrayList();` gera apenas um *warning* mas grandes problemas durante a execução. Se tiver *warnings* falando de declarações *unsafe* em colecções é isto !

O tipo `ArrayList()` é de Java1 a Java4 mas ainda existe em Java7 (por razões de retro-compatibilidade) mas não é "type safe", ou seja, verificado em tempo de compilação.

✎ As soluções "rabuscadas" surgem em geral por falta de domínio das construções da linguagem (em especial do pouco conhecimento das API das colecções). Por exemplo, muitos alunos acham que para verificar se um ficheiro tem linhas duplicadas é necessário ler o ficheiro duas vezes. Não! Basta ler as linhas para um `ArrayList<String> linhas;` e depois copiá-lo para, por exemplo, uma `HashSet<String>` usando o construtor, cf. `HashSet<String> noDupRecs = new HashSet<>(linhas);`. Calculando a diferença de `size()` saberemos quantas estavam em duplicado.

✎ É absolutamente obrigatório o uso de `clone()` em todos os métodos interrogadores, ou seja, que consultam o estado interno de uma qualquer instância. Claro que strings e as instância das classes "wrapper" não necessitam de `clone()`.

A expressão `HashSet<String> noDupRecs = new HashSet<>(linhas);`, que é um construtor de `HashSet<T>` que recebe como parâmetro um `ArrayList<T>`, elimina automaticamente duplicados, mas apenas está correcta porque estamos a trabalhar com strings e strings são imutáveis e não necessitam de `clone()` !

✎ Classes deverão ter também bons métodos `hashCode()` dado que tal torna muito mais eficiente a sua inserção ou remoção na maioria das colecções que necessitam de ordem e/ou indexação, por exemplo, `TreeSet<Ponto2D>`.

A classe `Arrays` oferece um método `hashCode()` geral que poderemos usar de forma muito simples em qualquer classe, programando-o da seguinte forma:



```
public int hashCode() {  
    return Arrays.hashCode( new Object[] { vari1, vari2, vari3, ... } );  
}
```

Passamos como parâmetro do método **Arrays.hashCode()** um *array* de objectos inicializado com as **variáveis de instância** dessa classe (tudo é compatível pois **Object** é superclasse de todas).

É simples e elimina a necessidade de usarmos números primos !!

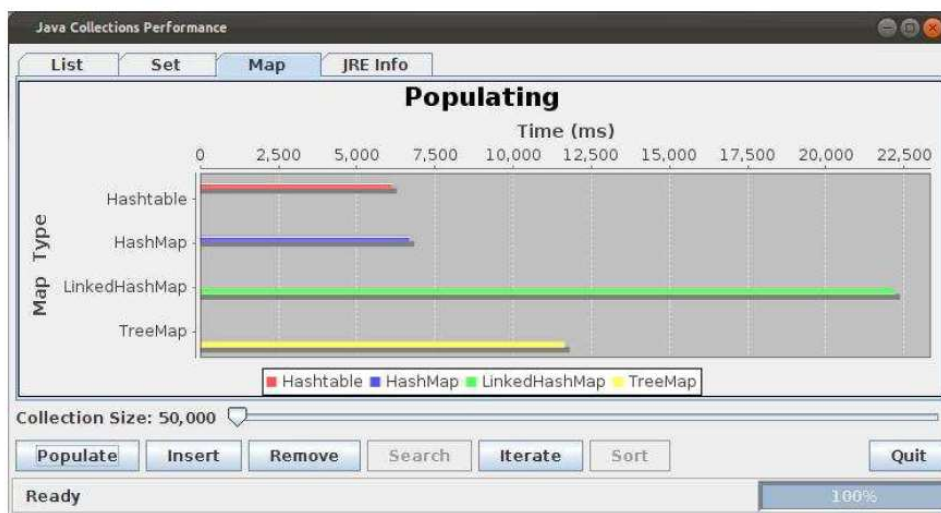
🔗 Informações semanais detalhadas serão dadas através dos obrigatórios BRPJAVA.

## MEDIDAS DE PERFORMANCE: Gráficos e Ferramentas



Universidade do Minho  
Departamento de Informática

### Performance Test: Exemplo



Nota: Caso surjam **OutOfMemoryException** usar as flags de java, **-Xms** e **-Xmx** para definir valores mín e máx de “heap size” (cf. **-Xms1024m -Xmx2048m**);

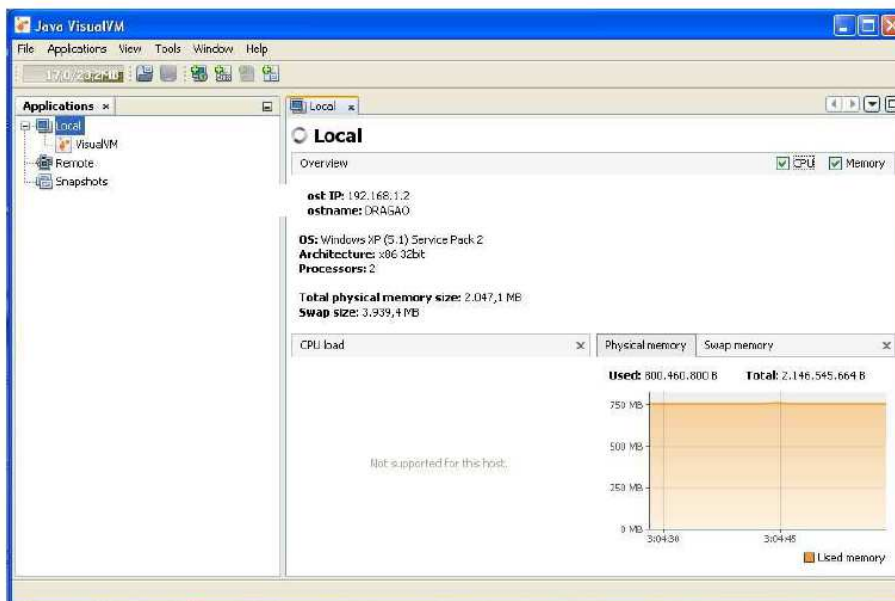
Atenção ao BlueJ que pode ser limitador.

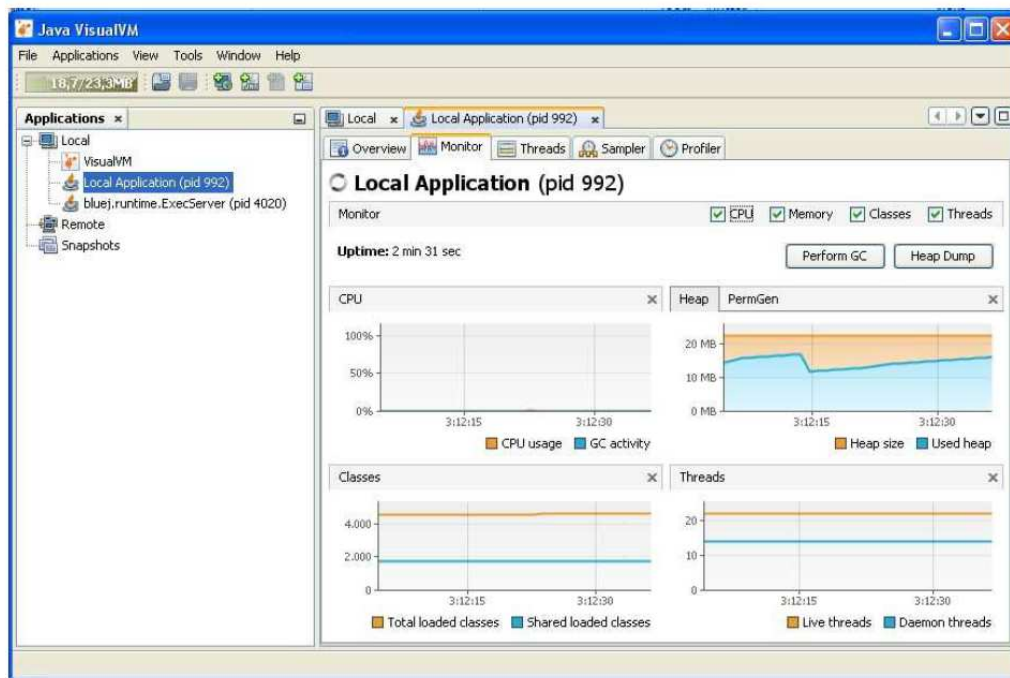
☐ Valores gerais de execução podem ser medidos e analisados recorrendo à utilização da própria **Java Visual Virtual Machine (J VisualVM)** em **jdk\bin**.

<http://docs.oracle.com/javase/6/docs/technotes/guides/visualvm/>



jvisualvm





---

F. Mário Martins