Laboratórios de Informática III (LI3)

LEI – 2º ano – 2º semestre – Universidade do Minho – 2014/2015

F. Mário Martins, António Luís de Sousa

Maio de 2015

Projecto de Java: GESTHIPER-OO

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.

Não será solicitada aos alunos a aprendizagem de novas construções da linguagem Java, procurando-se ainda capitalizar o mais possível do trabalho e dos resultados obtidos no projecto de C, e na utilização de parte desses conhecimentos para a criação de uma estruturação de dados em Java baseada na utilização das colecções e interfaces de JCF ("Java Collections Framework"), que permita a inserção e a realização de consultas interactivas de informações relativas à **gestão básica de um hipermercado** (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 do programa a desenvolver.

Pretende-se desenvolver um programa em Java que seja capaz de, antes de mais, ler e armazenar em estruturas de dados (coleccções 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 Hipermercado. Um hipermercado deverá conter um Catálogo de Clientes (todos os códigos destes), um Catálogo de Produtos (códigos dos produtos disponíveis), uma Contabilidade (relacionando produtos e suas vendas mensais) e um registo global de todas as compras realizadas, por quem e quando, que designaremos por Compras.

Cada uma destas subestruturas de informação de um hipermercado mantêm os requisitos que foram apresentados no projecto de C.

O desenho e a implementação do código do programa deverão ter em atenção que o mesmo deverá ter dois grandes grupos de funcionalidades (correspondentes a 80% do valor do trabalho):

a) **Leitura de dados de memória secundária** e população das estruturas de dados em memória central; Gravação da estrutura de dados em memória (instância da classe Hipermercado) em ficheiro de objectos;
b) **Queries**: operações de consulta sobre as estruturas de dados;

Complementarmente, equivalendo a 20% 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 **GESTHIPER-OO**, 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 a qualquer momento e carregar os respectivos dados para memória. Na primeira execução do programa a realização desta operação é obrigatória (ver também penúltimo parágrafo desta secção).
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 hipermercado.obj 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.
O arranque da aplicação deve também poder fazer-se inicializando a estrutura de dados a partir de uma ObjectStream sobre o ficheiro de objectos indicado pelo utilizador.
O programa deverá também poder ler a qualquer momento um outro qualquer ficheiro de texto contendo as informações referentes às compras registadas (cf. Compras1.txt, Compras3.txt).

3.2.- Estatística e queries interactivas.

Sendo inúmeras as possíveis informações 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 compras lido, designadamente, nome dos ficheiros, 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 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) e total global;
- Número de distintos clientes que compraram em cada mês (não interessa quantas vezes o cliente comprou mas apenas quem de facto comprou);
- Total de registos de compras inválidos (os registos completos, com os nomes e valores dos respectivos campos, ou seja, não é apenas a linha lida, deverão ser também guardados em ficheiro de texto dado pelo utilizador).

3.2.2.- Consultas interactivas.

1. Lista ordenada com os códigos dos produtos nunca comprados e respectivo total;
2. Lista ordenada com os códigos dos clientes que nunca compraram e seu total;
3. Dado um mês válido, determinar o número total de **compras** e o total de clientes distintos que as realizaram;
4. Dado um código de cliente, determinar, para cada mês, quantas compras fez, quantos produtos distintos comprou e quanto gastou. Apresentar também o total anual facturado ao cliente;
5. Dado o código de um produto existente, determinar, mês a mês, quantas vezes foi comprado, por quantos clientes diferentes e o total facturado;
6. Dado o código de um produto existente, determinar, mês a mês, **quantas vezes foi comprado** em modo N e em modo P e respectivas facturações;
7. 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;
8. 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);

9. Determinar os X clientes que compraram um maior número de diferentes produtos, indicando quantos, sendo o critério de ordenação igual a 7;
10. Dado o código de um produto, determinar o conjunto dos X clientes que mais o compraram e qual o valor gasto (ordenação cf. 7);

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 seus tempos de execução. A classe **Crono**, cujo código fonte foi colocado no BB, usa o método `long System.nanoTime()` e os métodos start(), stop() e print() para realizar tais tarefas de medição de tempos.

3.3.- Medidas de performance (20%).

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* e *profiling*).

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 20% 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, Compras.txt, usando as classes Scanner() e BufferedReader(); Usar em seguida os ficheiros Compras1.txt e Compras3.txt e fazer novas medições de tempos; Realizar os mesmos testes mas, agora, incluindo o tempo de *parsing*, ou seja, de separação dos campos da linha e criação da instância da classe que representará uma compra. Apresente os resultados sob a forma de uma tabela e também de forma gráfica.

2.- 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>, comparar as "performances" de alguns dos queries mais

complexos que foram pedidos (o critério de escolha deve basear-se nos tempos originais obtidos usando **Crono**).
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<>.

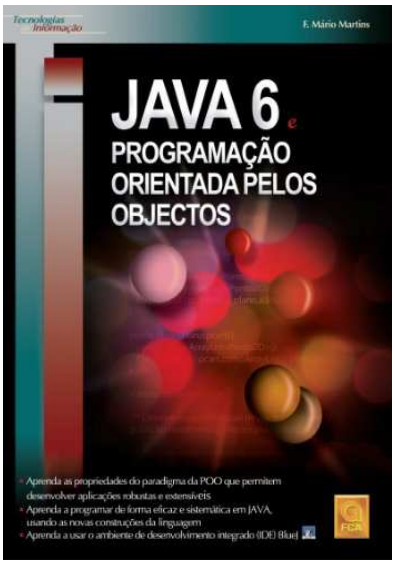
Nota: *Estes testes aparentemente complexos e trabalhosos não o são de facto dado que a substituição de uma colecção por outra do mesmo tipo preserva a linguagem, ou seja, por terem a mesma API os métodos são os mesmos. Assim, apenas há que realizar find/replace nas declarações.*
Por outro lado, ***se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

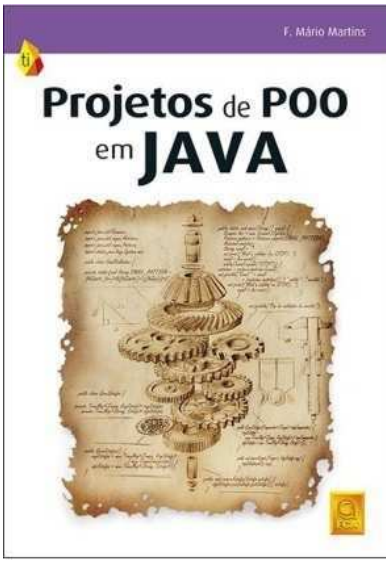
4.- Apresentação do projecto e Relatório.

O projecto será submetido por via electrónica sob a forma de uma pasta em formato zip/rar do projecto em BlueJ. Serão igualmente aceites projectos de outros IDEs como NetBeans e Eclipse. 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





ANEXO

QUESTÕES E RECOMENDAÇÕES GERAIS

✍ Este projecto de Java não tem polimorfismo ☹.

✍ 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 importantes para as consultas). Por exemplo, a classe **ParProdNumClis** poderá representar os pares anteriores.

✍ Instâncias de classes como **ParProdNumClis** poderão ter que ser guardadas em colecções como **HashSet<>.** 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<>**.
Tal implicará de imediato a criação de uma ou mais classes que implementem a interface **Comparator<>** respectiva, uma por cada algoritmo especial de ordenação dos pares, cf:

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

✍ 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 *warning*s 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 que estamos a usar (em geral as API das colecções). Muitos alunos acham que para verificar se um ficheiro tem linhas duplicadas é necessário ler o ficheiro duas vezes. Embora pouco relevante para o projecto, sendo apenas um exemplo didáctico (usa Crono, Scanner, etc.), apresenta-se uma solução simples e que poderá ser seguida em muitos outros casos (é um simples método auxiliar de main(), daí ser **static**):

```
public static void showQuery31() {
    out.println("--- LINHAS DUPLICADAS NUM FICHEIRO ----");
    out.print("Nome do ficheiro a ler : "); String ficheiro = Input.lerString();
    ArrayList<String> linhas = new ArrayList<String>();
    Crono.start();
    Scanner fichScan = null;
    try {
          fichScan = new Scanner(new FileReader(ficheiro));
          fichScan.useDelimiter(System.getProperty("line.separator"));
          while (fichScan.hasNext()) linhas.add(fichScan.next());
    }
    catch(IOException e) {out.println(e.getMessage()); }
    HashSet<String> noDupRecs = new HashSet<String>(linhas);
    out.println("Numero de Linhas lidas : " + linhas.size());
    out.println("Numero de Linhas em duplicado : " + (linhas.size() - noDupRecs.size()) );
    Crono.stop();
    out.println("Tempo Total de Teste de Duplicados = " + Crono.print() + " segundos.");
}
```

✍ 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 "wraper" não necessitam de clone.

A expressão acima, **HashSet<String> noDupRecs = new HashSet<String>(linhas);** que é um construtor de HashSet<> que recebe como parâmetro um ArrayList<>, desta forma eliminado-se automaticamente duplicados, apenas está correcta porque estamos a trabalhar com strings !

✍ 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:

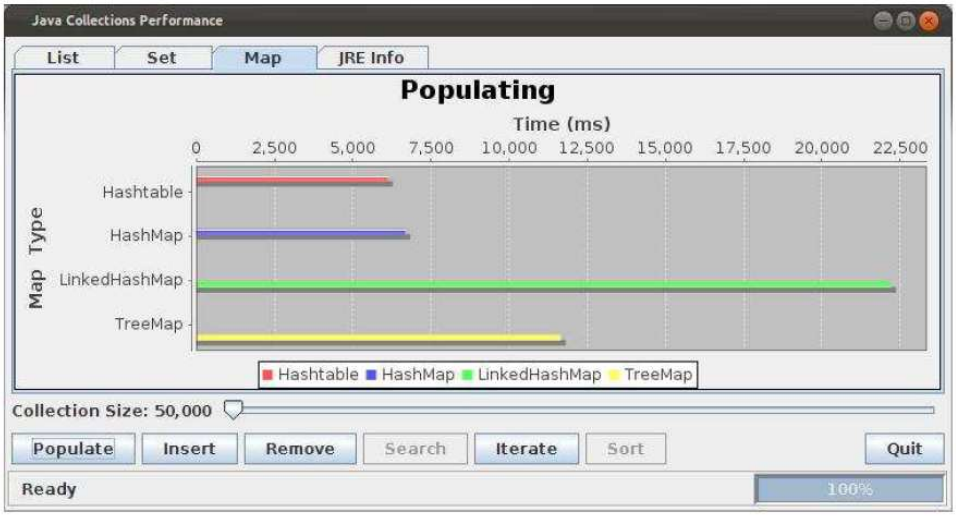
```
public int hashCode() {
      return Arrays.hashCode( new Object[] { vari1, vari2, vari3, … } );
}
```

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 !!

Universidade do Minho
Departamento de Informática

Performance Test: Exemplo

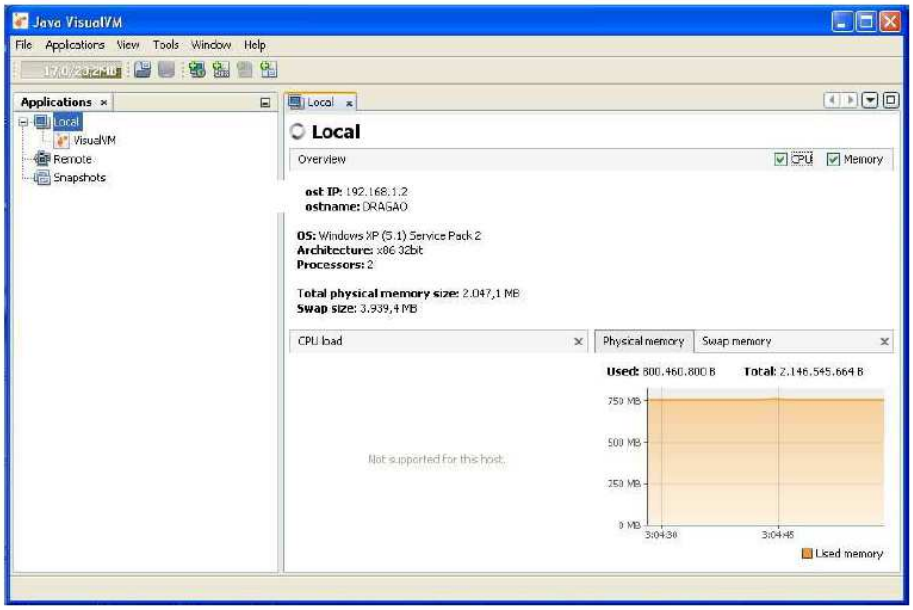


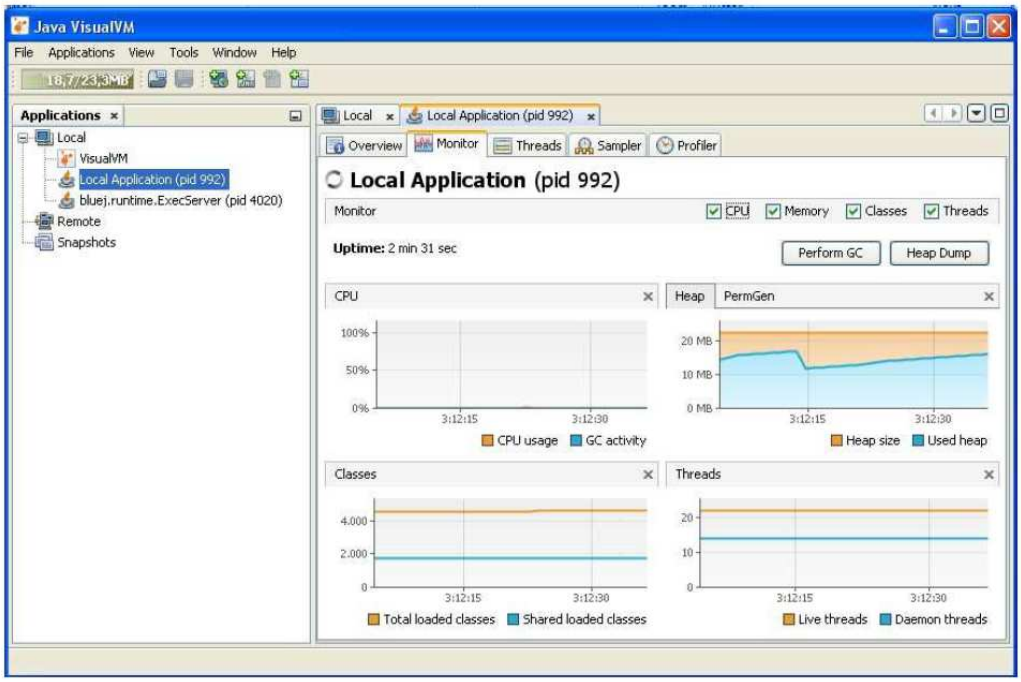
Nota: Caso surjam **OutOfMemoryException** usar as flags de java, **-Xms** e **-Xmx** para definir valores mín e máx de "heap size" (cf. –Xms1024m – Xmx2048m);

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/





Este documento corresponde ao enunciado do projecto de Java e aos conteúdos que seriam apresentados na aula Comum.

F. Mário Martins