GUIÃO PARA AS AULAS PRÁTICAS DE LI3 - JAVA: PARTE I

O projecto de Java, **GereVendas**, consiste fundamentalmente da leitura dos ficheiros que contêm os dados a serem tratados, cf. Clientes.txt, Produtos.txt e Vendas_3M.txt e da criação de uma classe **Hipermercado** que, usando as correctas colecções de JCF, irá estruturar todos os dados lidos de forma a responder da forma mais eficaz às consultas pretendidas.

Porém, nesta primeira semana do projecto de Java temos por objectivos pedagógicos solicitar aos alunos a realização de certas tarefas mais básicas, que não impedem a concepção global, e, antes pelo contrário, irão permitir solidificar procedimentos, criar mecanismos e introduzir alguns conceitos novos de Java que em muito irão ajudar no projecto final.

Tarefa 1: Definir a classe Venda de forma completa:

- Construtores das partes e de cópia;
- Programar todos os gets();
- Não programar sets() porque as instâncias de Venda são imutáveis;
- Programar os métodos toString(), clone() e equals().

Tarefa 2: Descarregar a classe Crono do BB e analisar, testar eventualmente, como funciona.

Tarefa 3: Usando BufferedReader e Scanner ler as linhas do ficheiro Vendas_3M.txt para um ArrayList<String> e comparar tempos de leitura.

Nunca se pode assumir que um arbitrário ficheiro de texto organizado em linhas usa como delimitador das suas linhas o mesmo delimitador de linha da nossa plataforma. Em Java, o delimitador de linha da plataforma, seja Windows, Unix, Linux ou OS, pode ser determinado usando `System.getProperty("line.separator")`. Em Windows será "\r\n", será "\n" em Unix e Linux, e "\r" em OS.

Mas tal é irrelevante na leitura. Em especial o método nextLine() da classe Scanner lida bem com todos estes problemas de compatibilidade.

```
public static ArrayList<String>
      readLinesArrayWithScanner(String ficheiro) {
  ArrayList<String> linhas = new ArrayList<>();
  Scanner scanFile = null;
  try {
      scanFile = new Scanner(new FileReader(ficheiro));
      scanFile.useDelimiter("\n\r");
      while(scanFile.hasNext())
          linhas.add(scanFile.nextLine());
  }
  catch(IOException ioExc)
       { out.println(ioExc.getMessage()); return null; }
  finally { if(scanFile != null) scanFile.close(); }
  return linhas;
}
```

Em seguida vamos codificar a leitura das linhas do ficheiro usando uma BufferedReader sobre uma FileReader usando igualmente o método readLine() da BufferedReader.

Teremos agora o seguinte código:

```
public static ArrayList<String> readLinesWithBuff(String fich) {
   ArrayList<String> linhas = new ArrayList<>();
   BufferedReader inStream = null;
   String linha = null;
   try {
         inStream = new BufferedReader(new FileReader(fich));
         while( (linha = inStream.readLine()) != null )
                 linhas.add(linha);
   }
   catch(IOException e)
       { out.println(e.getMessage()); return null; };
   return linhas;
}
```

Tarefa 4: Codificar um programa LerFichsTexto_Teste.java cujo método main() faça a leitura do ficheiro Vendas_3M.txt usando as duas funções auxiliares anteriores, conte o número total de linhas lidas e compare os tempos das duas leituras usando Crono. No final escolher a implementação a usar.

```
Ficheiro Vendas_3M.txt lido usando Scanner e FileReader !
Lidas e guardadas 3000000 linhas.
Tempo: 5.381097964
-------------------------------
Ficheiro Vendas_3M.txt lido usando BufferedReader !
Lidas e guardadas 3000000 linhas.
Tempo: 0.696382139

Numero de caracteres: 82259797
```

A diferença de tempos é BRUTAL em favor da BufferedReader. Os alunos têm que ter consciência disto de forma clara. E terão, depois da execução deste código.

Realizar o mesmo teste para Vendas_5M.txt.

Tarefa 5: Usando os métodos split() e trim() da classe String, escrever um método para ser usado no main() anterior, que transforme um ArrayList<String>, resultado das leituras, num ArrayList<Venda> vendas, ou seja, contendo instâncias da classe Venda que irão em seguida ser testadas.

Escrever o código dos métodos auxiliares:

```
public static Venda parseLinhaVenda(String linha) { ... }
public static ArrayList<Venda> parseAllLinhas(ArrayList<String> linhas) { ... }
```

Testar convenientemente o código destes dois métodos.

Escrever um método semelhante ao parseAllLinhas() mas que devolva um HashSet<Venda> e verificar se existem linhas duplicadas (cf. parseAllLinhasToSet();).

Tarefa 6: Determinar os tempos de **leitura com parsing** das linhas usando ambas as streams de leitura anteriorese ambos os dicheiros de vendas (3M e 5M).

Tarefa 6.1: Usando o código já desenvolvido, criar um método de assinatura

```
public static ArrayList<Venda> readVendasWithBuff(String fich)
```

que realize a leitura das linhas de vendas, faça no seu interior o *parsing* e devolva o *arraylist* ArrayList<Venda> que se pretende.

Tarefa 7: Usando o ArrayList<Venda> realizar alguns testes que correspondem a possíveis consultas (cada um deve ser codificado num método **static**), por exemplo:

- Determinar o número total de compras realizadas na filial dada como parâmetro;
- Determinar o número total de compras de preço 0.0;
- Contar o número de vendas duplicadas, ou seja, exactamente iguais;
- Determinar o total de produtos com código começado pela letra parâmetro;
- Criar um conjunto com todos os códigos de clientes que compraram na filial dada como parâmetro. Numa primeira versão usar HashSet<String>.

  Numa segunda implementação, criar um Comparator<String> que permita que os códigos sejam inseridos num TreeSet<String> por ordem alfabética crescente. Numa terceira implementação usar uma **expressão lambda** para definir o Comparator<String> e testar.

  Definir Comparator<String> ordemDecStrings = (s1, s2) -> s1.compareTo(s2); e usar este comparator na criação do treeset, cf.

  ```
  TreeSet<String> codsClientes = new TreeSet<>(ordemDecStrings);
  ```

- ...

Tarefa 8: Criar uma classe ParStringDouble que é um par String-Double, portanto tem duas variáveis de instância, uma String e um Double.

Criar um ArrayList<ParStringDouble> usando a seguinte construção auxiliar de Java:

```
ArrayList<ParStringDouble> paresSD =
   Arrays.asList( new ParStringDouble("X500", 20.75),
                  new ParStringDouble("Z398", 11.45),
                  new ParStringDouble("A11", 2.5),
                  new ParStringDouble("W455", 12.5)
                );
```

Em seguida criar dois Comparator<ParStringDouble> um que compare estes pares usando a ordem natural das strings e outro que os compare por ordem crescente do campo Double (equivalente a double).
Usando estes comparadores, copiar os elementos do ArrayList<ParStringDouble> paresSD para um TreeSet<ParStringDouble> usando um e depois o outro comparador.

Tarefa 9: Considerando as consultas realizadas na Tarefa 7, vamos usar as mais simples para realizar algumas comparações de tempos ao reprogramá-las usando Streams de Java, e usando o ArrayList<Venda> vendas; anteriormente criado.

```
/* Total de compras de preco unitario 0.0; codigo para inserir no main()  */
long totalZeros =
   vendas.stream()
         .filter(v -> v.getPreco() == 0.0)
         . peek(v -> System.out.println(v));  // apenas para inspecionar
```

```
                .count();
        System.out.println("Total de vendas de Preco 0.0 = " + totalZeros);
```

Continuaremos esta Tarefa 9 na Parte II deste Guião onde introduziremos testes com outras coleções e outras operações simples com Streams que poderão facilitar a codificação final do projecto.