Projeto 6

HashMap

Unidade Curricular de
Laboratório de Programação

2020/2021

Objetivos

- Praticar o uso e implementação de classes
- Praticar o uso de tabelas de dispersão (HashMap)

Antes de Começar

De modo a poder realizar este projeto deverá recordar a estrutura de dados Tabela de dispersão e a classe HashMap.

Deve descarregar o ficheiro `alunosProjeto6.zip` disponível na página de LabP e, em seguida, no Eclipse escolher `File → Import → General → Existing Projects into Workspace`, para importar esse ficheiro.

Deverá passar a ter um projeto chamado **Projeto6** contendo:

- Na pasta `src`, o ficheiro `RunPurchasePlanner.java` e a interface `Shop.java`
- Na pasta `tests`, o ficheiro `TestPurchasePlanner.java`, com vários testes `JUnit`

Purchase Planner

O *Purchase Planner* permite analisar e escolher as *lojas* adequadas para comprar certas *encomendas*.

Cada encomenda – **ItemOrder** – é composta por um artigo – **Item** – e respectiva quantidade. Os artigos têm uma designação e uma necessidade de refrigeração. O seu preço não é uma característica dos artigos visto que podem ter preços diferentes em lojas diferentes.

Vamos considerar um tipo de lojas em particular – os supermercados – que vendem todos os artigos possíveis e imaginários e que o fazem a um preço por omissão (imagine loja tudo a 1€), a não ser que se defina explicitamente o preço de um artigo específico.

O interface **Shop**, incluído no zip `alunosProjeto6`, define os métodos que qualquer loja deve implementar para poder ser usada pelo *Purchase Planner*. São eles:

- `int priceOf(Item item);` que obtém o preço de um item na loja, assumindo que item não é null.
- `Item cheapestItem();` que devolve o Item mais barato à venda nesta loja. Caso haja empates, deve devolver null pois não existe um único item que é o mais barato.
- `String toString();` que devolve uma representação textual da loja

O que fazer

Deve implementar as seguintes classes:

- **Item** - que representa items de supermercado. Os items têm duas características: descrição e necessidade de refrigeração (precisa ou não precisa). Dois items dizem-se iguais quando tanto a sua descrição como a necessidade de refrigeração coincidem. É assumido que a descrição de um item não é null. Além de um construtor e dos métodos para observar items, deve ainda implementar um método `public static Item getRandomItem(Random rand)` que cria um item com uma designação e necessidade de refrigeração aleatórias. Observe que os objetos desta classe são imutáveis, em particular porque não têm nenhum *setter.*

- **Supermarket** - que representa supermercados (como descritos na página anterior) e que implementa a interface **Shop.** Permite criar supermercados com um certo nome e preço de produtos por omissão e tem ainda o método que permite definir o preço de um Item particular:
  - `void setPriceOf(Item item, int price);` que define o preço de um item assumindo que item não é `null` e `price` > 0

- **ItemOrder** - que agrega um Item e uma quantidade (valor inteiro). Deve-se assumir que `item != null` e quantidade > 0. Deve implementar um construtor e ainda um método `toString()` e os observadores `getItem()` e `getQuantity()`.

- **PurchasePlanner** - que representa *planners* capazes de analisar e selecionar as lojas mais adequadas para a compra de um dado *plano* de encomendas. Implementa os seguintes métodos:

  para adicionar items e quantidades ao *Planner*:

  - `void addToOrder(List<ItemOrder> order);`
  - `void addToOrder(ItemOrder itemOrder);`
  - `void addToOrder(Item item, int qtty);`

para remover items e quantidades ao *Planner,* (se a quantidade a remover de um item for superior à existente já no plano, o efeito deve ser o mesmo do que remover todas as unidades existentes):

- `void removeFromOrder(List<ItemOrder> order);`
- `void removeFromOrder(ItemOrder itemOrder);`
- `void removeFromOrder(Item item, int qtty);`

E ainda:

- `String toString();` que devolve uma representação textual do plano. Em particular, deve indicar, para cada item no plano, quantos exemplares desse item são pedidos. Deve também indicar quantos artigos são encomendados na totalidade do plano.
- `int priceInMarket(Shop market);` que permite ver o preço do plano atual na loja `market`, assumindo que `market != null`
- `Shop cheapestMarket(List<Shop> markets);` que devolve a loja mais barata para o plano atual, assumindo que `markets != null && markets.size() > 0` e cada loja em `markets` não é `null`. Em caso de empate, deve devolver a primeira loja dos empatados.
- `Item mostlyCheaper(List<Shop> markets);` que determina se há, e qual é, o Item do plano atual que é o mais barato na **maioria** das lojas (isto é, em mais de metade das lojas), assumindo que `markets != null && markets.size() > 0` e que cada loja em `markets` não é `null`. Por exemplo, suponha que o plano contém os items *Banana* e *Morango* e `markets` contém 3 lojas. Se o item *Banana* for o item mais barato em 2 das 3 lojas, o método deverá devolver o Item que corresponde à *Banana*. No caso de não existir nenhum item que seja o mais barato na maioria das lojas, deverá devolver null. Como cada loja apenas pode ter um Item que é o mais barato (ver método `cheapestItem()` de **Shop**), não existem empates.

NOTA: repare que os parâmetros destes 3 últimos métodos são do tipo **Shop**, ou seja, só poderá invocar sobre eles os métodos definidos na *interface* **Shop**.

O que entregar

Deve entregar as classes java que desenvolveu, juntamente com a interface fornecida, devidamente comentadas.