



# Programação em Linguagem Java

## Projeto Intercalar - Gestão de Finanças Pessoais

Novembro/Dezembro 2020

### 1 Introdução

Os sistemas de finanças pessoais são uma aplicação comum, permitem importar dados do banco através de ficheiros com um formato simples (tipicamente *csv*, *comma separated values*) e ter uma visão personalizada do tipo de despesas feitas. Podem ser encontrados diversos exemplos online deste tipo de aplicações.

Este trabalho consiste em criar um sistema simples de gestão das contas de um utilizador. Deve ler ficheiros que contêm dados de uma conta bancária, num formato muito próximo do formato usado na realidade por alguns bancos. Deve permitir categorizar as despesas (por exemplo, dividir em despesas de CASA, SAÚDE, ESCOLA e EXTRAS) de acordo com as descrições dos movimentos e fazer algumas análises dos movimentos.

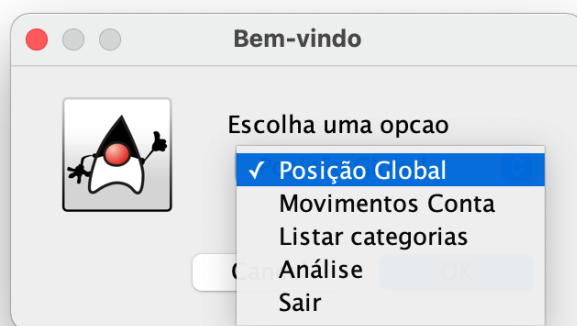
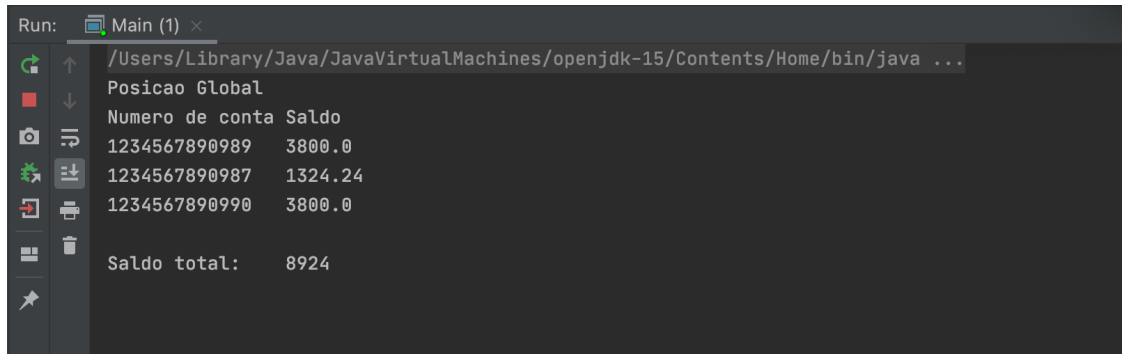


Figura 1: Exemplo da interface gráfica da aplicação (utilização da classe `Menu`, fornecida no pacote `base`)



```
Run: Main (1) x
/Users/Library/Java/JavaVirtualMachines/openjdk-15/Contents/Home/bin/java ...
Posicao Global
Numero de conta Saldo
1234567890989 3800.0
1234567890987 1324.24
1234567890990 3800.0

Saldo total: 8924
```

Figura 2: O resultado das consultas deverá ser impresso para a consola.

Resumidamente, este programa deve permitir visualizar de forma condensada a situação financeira do utilizador, ver detalhes de movimentos de uma conta, categorizar os movimentos de uma conta e calcular a divisão das despesas por categoria.

## 2 Requisitos funcionais

Deve implementar um sistema de gestão de finanças pessoal, com as seguintes funcionalidades:

1. Ao iniciar o programa deve ler toda a informação gravada em ficheiros na pasta `account_info` e também as descrições de movimentos recentes num formato semelhante na pasta `statements`. Deve ter o cuidado de impedir a inserção duplicada de movimentos, re-ordenar os movimentos se necessário e tratar adequadamente as exceções lançadas em caso de erros no formato dos ficheiros. Caso haja um ficheiro com movimentos de uma conta que não existe, deve criar essa conta.
2. Mostrar os vários menus e permitir escolher opções.
3. Categorizar automaticamente todos os movimentos cuja descrição esteja já incluída numa das categorias.
4. Categorizar manualmente (pelo utilizador) todos os movimentos cuja descrição não esteja incluída em nenhuma das categorias.
5. Mostrar a posição global: resumo de todas as contas do utilizador e respetivos saldos atuais.
6. Mostrar os movimentos de uma conta. Podem existir dois tipos de contas, as contas à ordem (“DRAFT”) e as contas de poupança (“SAVINGS”) tal como detalhado nos requisitos de

implementação.

7. Listar as categorias definidas pelo utilizador até ao momento.
8. Listar o resumo dos saldos por mês (totais de movimentos para todas as contas, por mês).
9. Cálculo dos débitos feitos numa conta por categoria, desde o início do mês corrente e previsão de débitos nessa conta, por categoria, até ao final do mês. A previsão é feita usando uma regra de três simples, o valor estimado no final do mês para uma categoria é igual ao valor atual sobre o número de dias que passaram este mês, vezes o número de dias do mês.
10. Previsão do total de juros a auferir no final do ano em cada conta de acordo com as regras do ponto da secção “Requisitos de implementação” relacionado com a hierarquia de tipos de contas-bancárias.
11. Gravar toda a informação (contas e categorias), no mesmo formato em que se encontra na pasta `account_info`.

### 3 Formato dos ficheiros

Registo de informação sobre as contas, lido e escrito pelo programa:

Account Info - <Data da última atualização pelo utilizador>

Account ;<nº da conta> ; EUR ;<nome da conta> ;<tipo da conta> ;<info adicional>

Start Date ;<Data do primeiro movimento registado>

End Date ;<Data do último movimento registado>

<Legenda dos movimentos>

<Movimentos>

Registo de informação sobre novos movimentos, lido pelo programa e enviado/escrito pelo banco:

Statement - <Data da consulta>

Account ;<nº da conta> ; EUR ;<nome da conta> ;<tipo da conta> ;<info adicional>

Starting Date ;<Data do primeiro movimento registado>

Ending Date ;<Data do último movimento registado>

<Legenda dos movimentos>

<Movimentos>

Formato dos movimentos:

```
<Data> ;<Data-valor> ;<Descrição> ;<Débito> ;<Crédito> ;<Saldo contabilístico> ;  
<Saldo disponível> ;
```

Linhas ou colunas vazias podem existir e devem ser ignoradas ou tratadas adequadamente. Em alguns ficheiros pode não existir informação adicional.

O formato do ficheiro de categorias é livre (o ficheiro no projeto exemplo está vazio), mas tem de conter o nome das categorias e, para cada categoria a lista das descrições que lhe estão associadas.

## 4 Pacote-base e detalhes de implementação

Para o desenvolvimento do projeto, é fornecido pelos formadores um pacote base que inclui um conjunto exaustivo de casos de teste. Este projeto deverá seguir a metodologia de Test Driven Development, servindo os testes como ponto de partida para o desenvolvimento da aplicação.

No pacote base fornecido, são incluídas também classes para escrever para o ecrã bem como pedir inputs do utilizador (classe `Menu`), uma classe para representar datas (classe `Data`), entre outras.

### 4.1 Test Driven Development

O Desenvolvimento Guiado por Testes (Test Driven Development) é uma técnica de desenvolvimento de software onde o desenvolvedor começa por criar testes automatizados que definam requisitos em código antes de escrever o código da aplicação.

Os testes contêm assertões que podem ser verdadeiras ou falsas. Após as mesmas serem consideradas verdadeiras, após sua execução, os testes confirmam o comportamento correto, permitindo os desenvolvedores evoluir e melhorar o código. Normalmente, usam frameworks de testes como `jUnit` para criar e executar automaticamente uma série de casos de teste.

Este projeto segue esta metodologia de desenvolvimento e o pacote base do projeto é constituído por um conjunto de casos de teste. Nestes, são validadas as funcionalidades esperadas da aplicação, ex:

```
@Test  
public void testFirstOfNextMonth() {  
    Date d = new Date(2, 2, 2014);
```

```

        assertEquals(new Date(1, 3, 2014), Date.firstOfNextMonth(d));
    }

```

Por exemplo, o teste acima permite verificar se a função `firstOfNextMonth(Date d)` devolve o valor correcto.

Nota: para correcta avaliação do trabalho, não podem ser alterados os casos de teste nem os ficheiros dentro da pasta `account_info_tests`; os testes podem ser corridos diretamente no IntelliJ.

## 4.2 Formataadores

Os formataadores são classes que definem os formatos de contas, movimentos e ficheiros, tornando estas classes independentes do modo como queremos ver os dados que contêm as suas instâncias.

A função `format()` de cada formataador (i.e. de cada implementação da interface genérica `Format<T>` fornecida no pacote-base) define que informação vai inspeccionar no objecto dado passado como argumento e como vai mostrar essa informação ao utilizador.

A utilização desta técnica para separar o modo como se veêm os dados de uma conta do seu funcionamento interno é altamente aconselhada e permite trocar rapidamente de interface (mudar a forma como os dados são formatados), sem mexer nas funcionalidades críticas do sistema. Tudo isto tendo por base o recurso ao polimorfismo.

Este tipo de organização pode permitir também uma combinação de formatos para mostrar a informação das contas de vários modos diferentes.

## 4.3 Filtros

Os filtros são uma maneira modular de fazer seleção de elementos de uma lista segundo um determinado critério. Os filtros contêm um seletor e o seu principal método (`apply()`) copia para uma nova lista todos os objetos que respeitem o critério definido no selector.

Os seletores são implementações da interface genérica `Selector<T>` fornecida no pacote-base e permitem seleccionar objectos que cumprem uma determinada condição. No seletor, a função `isSelected()`, devolverá o valor lógico de `true` se o objeto passado como argumento respeitar um determinado critério. Veja o exemplo no projecto da classe `AccountIdSelector`.

## 5 Requisitos de implementação

1. Têm de ser adequadamente usados os interfaces fornecidos no projeto exemplo.
2. Deve ser usada a classe `Menu` para os pedidos de informação ao utilizador.
3. As classes que definem as contas e movimentos devem ser independentes dos formatos de saída de dados, que devem ser codificados em classes que implementam os interfaces `StatementLineFormat` e `Format<Account>`.
4. Deve usar filtros (e respetivos seletores) de movimentos e contas derivados das classes genéricas `Filter` e `Selector` tal como o exemplo `AccountIdSelector`.
5. As JUnits de teste fornecidas deverão estar a funcionar.
6. As exceções verificadas (aquelas cujo tratamento é obrigatório) devem ser tratadas ao nível adequado à sua recuperação sempre que possível, ou de modo a que o utilizador seja avisado da causa do erro antes do programa terminar. O programa não deve terminar abruptamente sem aviso prévio.
7. Deve ser usada a consola (`System.out`) para dar informação ao utilizador.
8. Todas as classes podem ser alteradas desde que os testes se mantenham funcionais, exceto as classes de teste, as interfaces, a classe `Date` e o enumerado `Month`.
9. As classes implementadas devem estar adequadamente distribuídas por `packages`.
10. Deve existir uma classe `Main` onde estará o método `main()` para iniciar o programa.
11. Cada categoria deve guardar o seu nome e todas as descrições que foram associadas à categoria pelo utilizador.
12. Deve existir uma hierarquia de tipos de contas-bancárias para permitir calcular a previsão dos juros a receber de modo diferenciado:
  - Nas contas à ordem (“DRAFT”) a taxa de juros é fixa e igual para todas as contas do banco em questão. A previsão dos juros anuais é calculada multiplicando o saldo médio estimado desde o início do ano até ao dia de hoje pela taxa de juro. O saldo médio é uma média ponderada dos movimentos do tipo  $(a_1s_1 + a_2s_2 + \dots + a_ns_n)/(a_1 + a_2 +$

$\dots + a_n$ ), sendo  $a_n$  o número de dias em que  $s_n$  foi o saldo na conta bancária. Devem ser contabilizados os saldos desde o início do ano até ao dia de hoje.

- Nas contas de poupança (“SAVINGS”) a taxa de juros é fixa, embora mais alta que a das contas à ordem. A previsão dos juros é feita do mesmo modo, multiplicando o saldo médio estimado pela taxa de juros, mas neste caso a estimativa do saldo médio será o saldo corrente (presume-se que estas contas têm um valor crescente ao longo do ano, por isso o saldo atual deverá ser uma boa estimativa do saldo médio anual); As contas de poupança categorizam automaticamente todos os seus movimentos como SAVINGS.

## 6 Sequência de funcionamento e responsabilidades das classes

1. Ao iniciar a aplicação, no construtor da classe **PersonalFinanceManager** devem ser lidos todos os ficheiros nas pastas **account\_info** e **statements**. Na pasta **account\_info** estão os ficheiros que são lidos e escritos pela aplicação. Na pasta **statements** estão os ficheiros originários do banco e que deveriam ser apagados ou substituídos manualmente pelo utilizador após o fecho da aplicação e antes de reiniciar. A aplicação lê os ficheiros em **statements** e inclui a informação nas suas estruturas internas que serão (se o utilizador quiser) escritas no final da aplicação na pasta **account\_info**. Atenção ao seguinte:
  - Para a leitura de cada ficheiro deve ser chamada a função-fábrica **newAccount**. Para a leitura de cada linha de movimento deve ser chamada a função-fábrica **newStatementLine**.
  - Os ficheiros em **statements** podem não ser atuais, mas a listagem de movimentos tem de manter a ordem por data. Deve presumir (e verificar) que os movimentos em cada ficheiro vindo do banco estão corretamente ordenados por data, em caso de datas iguais, presume que está correta a ordenação).
  - Deve verificar se os movimentos estão coerentes (se o levantamento ou depósito e o saldo após o movimento estão de acordo com a informação anterior).
  - Deve verificar possíveis problemas de formato dos números de conta e dos movimentos (as restantes partes dos ficheiros não são críticas, podem não ser alvo de verificações e lançamento de exceções).
2. É iniciado o interface gráfico com o utilizador que irá pedir opções (usando o Menu) e devolver respostas em texto, para a consola (esta funcionalidade é responsabilidade da classe

`PersonalFinanceManagerInterface`). O cálculo das respostas deve ser feito (ou coordenado) pela classe que contém os dados (`PersonalFinanceManager`). A informação mostrada ao utilizador deve, sempre que possível, ser formatada pelas classes que implementam o interface `Format<T>`.

3. Ao terminar, se o utilizador desejar gravar, a classe `PersonalFinanceManager` deve guardar todos os seus dados em ficheiro (um ficheiro para cada conta e um ficheiro para a informação sobre categorias) na pasta `account_info`.
4. A classe `Account` deve ter tudo o que é comum a todas as contas (métodos e atributos).

**Bom trabalho!**