Aula Prática 3

Resumo:
- Programação modular.

Exercício 3.1
Implemente um módulo `Data` com os seguintes métodos públicos[1]:

- Construtor com três argumentos: dia, mês e ano;
- Um conjunto de métodos que indiquem o dia, mês e ano do objecto data;
- Três funções – `igualA`, `menorDoQue` e `maiorDoQue` – que estabelecem a relação de ordem entre datas (todas com um argumento do tipo `Data`).

Quando estiver pronto, este módulo deve tornar funcional o programa `p31.java` fornecido[2].

Exercício 3.2
Modifique o módulo `Data` passando a operação de escrita de uma data (com o mesmo formato da existente no programa `p31.java`, para dentro da classe[3].

Exercício 3.3
Crie um novo módulo – `Nota` – que representa um texto associado a um intervalo entre duas datas. Utilize o programa `p33.java`para inferir qual tem de ser a interface do módulo.

[1] Caso tenha resolvido o problema 2.5 adapte e reutilize esse módulo.
[2] Disponível dentro do arquivo `aula03.zip` existente no `moodle`.
[3] Utilize o programa `p32.java` para testar o módulo.

Exercício 3.4

Utilizando os módulos desenvolvido nos exercícios anteriores construa um novo módulo `Agenda` onde seja possível registar notas. Neste novo módulo deve ser possível realizar as seguintes tarefas:

- Adicionar uma nova nota (método `novaNota`);
- Devolver um *array* com os compromissos existentes num intervalo de datas (método `compromissos`);
- Escrever o conteúdo completo da agenda (método `escreve`).

Note que as notas na agenda devem estar sempre ordenadas por ordem crescente pelas suas datas iniciais.

A saída do programa fornecido `p34.java` tem de ser a seguinte:

```
Agenda:
22-03-2012 <-> 27-03-2012: Prog2: ACITP1
09-05-2012 <-> 09-05-2012: Prog2: AIP
06-06-2012 <-> 06-06-2012: Prog2: ACITP2
14-06-2012 <-> 27-06-2012: Prog2: APF
02-07-2012 <-> 13-07-2012: Prog2: Recurso

Compromissos de 27-03-2012 a 15-06-2012:
22-03-2012 <-> 27-03-2012: Prog2: ACITP1
09-05-2012 <-> 09-05-2012: Prog2: AIP
06-06-2012 <-> 06-06-2012: Prog2: ACITP2
14-06-2012 <-> 27-06-2012: Prog2: APF
```

Exercício 3.5

Pretende-se construir um módulo reutilizável que implemente a caixa de uma loja. Este módulo deve permitir: o armazenamento de diferentes valores de moedas (e notas), mostrar as moedas existentes, colocar moedas e retirar dinheiro. Para simplificar o problema, considere que todos os valores estão em cêntimos (utilize o tipo `long` para o representar).

Implemente este módulo (com o nome `Caixa`) e teste-o simulando com um programa tipo menu:

O aspecto do menu deve ser o seguinte:

```
1. Adicionar moedas
2. Retirar dinheiro
3. Ver moedas na carteira
4. Ver total da carteira
0. Termina

Opção:
```

As operações que deve ser possível realizar serão as seguintes:

- Adicionar um conjunto de moedas à carteira (método `void adicionaMoeda(long moeda)`);
- Retirar moedas da carteira por forma a perfazer um valor mínimo de dinheiro (método `long[] retiraDinheiro(long valorMin)`);
- Mostrar as moedas existentes na carteira (método `long[] moedas()`);
- Ver o total de dinheiro existente na carteira (método `long total()`).

Exercício 3.6

Pegando no exercício da aula anterior (problema 2.5), crie um novo módulo `Data`, exactamente com o mesmo Tipo de Dados Abstracto[4] que a classe `Data` já feita, mas em que a representação interna da data seja somente o número de dias desde 1 de Janeiro de 2000[5].

Faça com que o programa desenvolvido na aula anterior funcione igualmente com objectos deste novo módulo.

Para evitar conflitos entre as duas classes `Data`, pode fazer uso dos mecanismos de definição de pacotes do Java (coloque as classes em pacotes diferentes, e seleccione-as pela instrução `import`).

Acrescente também a essas duas classes os serviços de relação de ordem (igual, maior, menor), assim como um serviço que indique o número de dias entre duas datas (modifique o programa acrescentando as opções que entender ao menu por forma a poder testar esses novos serviços).

Exercício 3.7

Na área da construção de edifícios de habitação, os vários profissionais precisam de gerir a informação sobre cada unidade habitacional (casa ou apartamento), bem como extrair diversas propriedades. Pretende-se que desenvolva um conjunto de classes para esta aplicação. Fornece-se em anexo um programa de teste das funcionalidades pretendidas bem como uma classe para representação de pontos num espaço cartesiano.

a. Desenvolva uma classe `Room` para representar as divisões das habitações. Assume-se que cada divisão terá uma forma rectangular, estando alinhada com os eixos de um determinado sistema de coordenadas. Esta classe deverá ter os seguintes métodos:

   - Construtor com três argumentos, nomeadamente o tipo da divisão (uma cadeia de caracteres), e as coordenadas dos cantos inferior esquerdo e superior direito;
   - `roomType()` - devolve o tipo da divisão;

[4]Ou seja: exactamente com a mesma interface publica.

[5]Esta definição é idêntica à de *data Juliana*, usada em Astronomia, embora aí a data de referência seja outra.

- `bottomLeft()` - devolve o canto inferior esquerdo;
- `topRight()` - devolve o canto superior direito;
- `geomCenter()` - devolve o centro geométrico da divisão;
- `area()` - devolve a área da divisão;

b. Desenvolva uma classe `House` para representar as habitações, com os seguintes métodos:

- Construtor que recebe como argumento o tipo da habitação - além de registar o tipo da casa, este construtor deve reservar memória para 8 divisões e deve também registar que, caso seja preciso armazenar informação sobre mais divisões, a memória das divisões será expandida em blocos de 3 divisões adicionais (inicialmente 8, depois sucessivamente 11, 14, etc., conforme as necessidades);
- Construtor que recebe como argumentos o tipo da habitação, o número de divisões, para as quais se vai inicialmente reservar memória, bem como o número de divisões adicionais a reservar sempre que a memória esteja cheia;
- `addRoom(Room)` - adiciona uma nova divisão à habitação;
- `size()` - devolve o número de divisões da casa;
- `maxSize()` - devolve o número máximo de divisões que é possível armazenar num dado momento;
- `room(int)` - dado um índice de uma divisão (um inteiro entre `0` e `size()-1`), devolve a divisão correspondente;
- `area()` - devolve a área total da habitação, dada pela soma das áreas das divisões;
- `getRoomTypeCounts()` - devolve os tipos de divisões existentes com o número de divisões de cada tipo, na forma de um vector (array) de elementos da seguinte classe:

```
public class RoomTypeCount {
    String roomType;
    int count;
}
```

  Nota: este vector não deverá estar sobre-dimensionado.

- `averageRoomDistance()` - devolve a distância média entre as divisões da casa, tomando como referência os respectivos centros geométricos;