Aula Prática 4

Resumo:

- Programação por Contrato.

Exercício 4.1

Pretende-se implementar um programa para o jogo "Advinha o número". Neste jogo, o programa escolhe aleatoriamente um número secreto num determinado intervalo [min, max] e o objectivo é adivinhar esse número com o mínimo de tentativas. Por cada tentativa feita, o jogo deve indicar se acertou (e terminar), ou se é menor ou maior do que o número secreto.

Implemente o comportamento essencial do jogo no módulo `GuessGame` respeitando a seguinte interface pública. Sempre que possível, use instruções `assert` para tornar explícitos os contratos dos métodos, particularmente as suas pré-condições.

- Um construtor com dois argumentos (`min` e `max`) que inicializa o jogo com um número aleatório no intervalo `[min,max]` (que não poderá ser vazio). Pode utilizar o método `Math.random()` ou a classe `Random` para gerar o número aleatório.
- Dois métodos inteiros – `min()` e `max()` – que indiquem os limites do intervalo definido para o objecto.
- Um método booleano – `validAttempt` – que indique se um número inteiro está dentro do intervalo definido.
- Um método booleano – `finished()` – que indique se o segredo já foi descoberto.
- Um método – `play` – que registe uma nova tentativa para adivinhar o número secreto. Este método só deverá aceitar tentativas que estejam dentro do intervalo definido e só enquanto o jogo não tiver terminado ou seja, enquanto o segredo não tiver sido descoberto.
- Dois métodos booleanos – `attemptIsHigher()` e `attemptIsLower()` – que indiquem se a última tentativa foi, respectivamente, acima ou abaixo do número secreto.
- Um método inteiro – `numAttempts()` – que indique o número de tentativas (jogadas) já realizadas.

Note que terá de definir um conjunto de campos (que devem ser privados) adequados para a implementação do módulo. Ou seja, necessita de campos que permitam saber os valores do intervalo, o número secreto, o número de tentativas, ou determinar se a última tentativa é igual, maior ou menor que o secredo.

Para facilitar a depuração do módulo, é fornecido um programa (`main`) embutido no próprio módulo, que utiliza a instrução `assert` para fazer alguns testes ao funcionamento do módulo. Assim, para testar o módulo, deve compilá-lo e executá-lo com as asserções activadas.

```
javac GuessGame
java -ea GuessGame
```

Exercício 4.2

Complete o programa `p42` que usa o módulo do exercício anterior para implementar o jogo "Advinha o número". Sem argumentos, o programa escolhe um número secreto no intervalo $[0, 20]$.

Altere o programa para aceitar dois argumentos que indiquem os limites do intervalo. A interação com o utilizador deve ser análoga à da solução fornecida, que pode testar com o comando seguinte.

```
java -ea -jar p42.jar
```

Exercício 4.3

Modifique os módulos `Data` e `Tarefa` da aula anterior por forma a tentar assegurar a sua correcção. Com esse objectivo, aplique contratos ao módulo `Data` por forma a garantir que os seus objectos correspondem sempre a datas válidas. Para dar uma boa solução para este problema, deve recorrer aos métodos estáticos referidos no exercício 2.5. No caso do módulo `Tarefa`, aplique contratos de forma a garantir as seguintes propriedades em todos os objectos deste tipo:

- a data de fim da tarefa não pode ser anterior à data de início;
- todas as tarefas têm um texto não vazio.

Teste os módulos correndo o programa `p43` sempre com as asserções ativadas.

```
java -ea p43
```

Exercício 4.4

No problema 3.5 da aula anterior a noção de "caixa" de dinheiro não tinha em consideração algumas propriedades essenciais desta abstracção. Por um lado, nada era dito sobre os valores admissíveis das moedas. Por outro, nada impedia que se retirasse mais dinheiro do que o existente na caixa ou que a caixa tivesse, por exemplo, um montante negativo. Acrescente contratos adequados para corrigir estas situações. Defina um método

`moedaValida` que verifique se um valor corresponde a uma moeda válida (1 cêntimo, 2 cêntimos, 5 cêntimos, ou qualquer potência de 10 vezes 1, 2 ou 5).

Exercício 4.5

A classe `number.Fraction` é baseada na `v5.Fraction` dada na aula03.

- Altere o invariante da class `Fraction` para garantir que o denominador armazenado é sempre positivo. Este invariante mais forte implica alterar vários métodos. Por exemplo, `new Fraction(3, -5)` deve continuar a funcionar, mas o novo objeto deverá ter `num = -3` e `den = 5`. Por outro lado, a implementação de `toString()` pode ficar mais simples.
- Acrescente-lhe métodos para subtrair, dividir e para verificar a igualdade de frações.

Teste com o programa `p45`. Confirme que $3/4 = 6/8$ e teste as restantes operações.

Exercício 4.6

O programa `p46` usa a função `Arrays.sort` para ordenar um array de fracções. Para poder funcionar, altere a declaração da classe `Fraction` para

```
public class Fraction implements Comparable<Fraction>
```

e defina uma função `compareTo` com o funcionamento e propriedades prescritas na interface `Comparable`. Isto é, `a.compareTo(b)` deve devolver um inteiro negativo se $a < b$, positivo se $a > b$ e nulo se $a = b$.

Exercício 4.7

Altere a implementação da class Fraction para garantir que as frações são sempre armazenadas na forma reduzida (ou irredutível), com o menor denominador possível (e positivo). Dará jeito criar uma função para achar o máximo divisor comum de dois inteiros. Altere a definição do invariante interno e modifique os restantes métodos adequadamente.