Aula Prática 4

Resumo:
- Programação por Contrato.

Exercício 4.1

Modifique a classe `Data` da aula anterior por forma a garantir que os seus objectos correspondem sempre a datas válidas. Para isso, acrescente instruções `assert` para testar pré-condições adequadas aos métodos `Data(d, m, a)` e `diasDoMes(m, a)` e para testar o invariante desejado no final do método `seguinte()`. Recorra ao método estático `dataValida(d, m, a)` que já criou. Por conveniência, pode também criar e usar um método de objeto `valida()` que verifique a validade do próprio objeto.

Teste esta classe correndo o programa `TestData3` com as asserções ativadas.

```
java -ea TestData3
```

Nesse programa, descomente a linha

```
Data.diasDoMes(0, 1900); // should fail!
```

compile e volte a testar. O programa deveria falhar com um erro semelhante ao seguinte:

```
java.lang.AssertionError: Mês inválido;
```

indicando que o teste à pré-condição falhou.

Volte a comentar a linha, descomente a próxima e volte a testar. Para cada uma dessas linhas, o programa deverá detetar uma falha de asserção. Devem ser asserções no início dos métodos, correspondentes a pré-condições, o que indica que a responsabilidade do erro é do programa cliente.

Introduza erros propositados na implementação dos métodos `diasDoMes(...)` e de `seguinte()` e volte a testar. As asserções que falham agora devem estar no fim dos métodos, correspondendo a pós-condições, o que indica que a responsabilidade do erro é do próprio método.

Exercício 4.2

Modifique também a classe `Tarefa`, acrescentando asserções que garantam as seguintes propriedades em todos os objectos deste tipo:

- a data de fim da tarefa não pode ser anterior à data de início;
- o texto de uma tarefa não pode ser vazio.

Teste esta classe correndo o programa `TestTarefa2` sempre com as asserções ativadas.

```
java -ea TestTarefa2
```

Exercício 4.3

A classe `number.Fraction` é inspirada num exemplo desenvolvido numa aula teórico-prática (ver `aula03-print.pdf`). A classe já tem um construtor e métodos para adicionar e multiplicar frações. Note que esta classe garante que os seus objetos nunca têm denominador nulo. O construtor já tem asserções que definem e testam o seu contrato.

- 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 ficar com `num = -3` e `den = 5`. Por outro lado, a implementação de `toString()` pode ser simplificada. Precisa de alterar o método `parseFraction`? E os restantes?
- Complete o método `equals` que serve para verificar se duas frações são iguais.
- Complete os métodos para dividir e subtrair. Qual a pré-condição do método `divide`? Inclua uma asserção para o testar. Inclua também asserções para testar a pós-condição de cada operação. Note que o quociente $q = a/b$ tem de satisfazer $q \cdot b = a$. Analogamente, a diferença $d = a - b$ tem de satisfazer $d + b = a$. Garanta ainda que o resultado satisfaz o invariante das frações.

Use o programa `FractionCalcB` para experimentar a classe. Calcule o resultado das expressões abaixo e outras que se lembre.

```
3/4 + 4/3
25/12 - -3/-4
64/48 == 4/3
3/2 x 1/3
2/-3 : -1/3
```

Exercício 4.4

As asserções, além de serem uma excelente ferramenta para verificar os contratos das classes, também podem ser usados para especificar testes *externos*. O programa `TestFractions` é um exemplo disso: faz uma série de testes à classe `Fraction`. Experimente compilá-lo e executá-lo. Corrija quaisquer erros que detete e volte a tentar.

Este programa tem vários outros testes em comentário. Deverá descomentar os testes progressivamente à medida que for implementando as funcionalidades abaixo.

- Complete o método de comparação na classe `Fraction`. Pretende-se que `a.compareTo(b)` devolva um inteiro negativo se $a < b$, positivo se $a > b$ e nulo se $a = b$. Descomente os testes a essa função no `TestFractions` e volte a correr.

- A biblioteca *standard* de Java inclui a função `Arrays.sort` que permite ordenar arrays genéricos com elementos de qualquer tipo de dados que tenha uma relação de ordem definida. Por exemplo, é possível ordenar um array de inteiros, ou um array de strings. Descomente os testes para ver se consegue ordenar um array de frações. Funcionou?

- Para o `Arrays.sort` funcionar com arrays de frações, além de ter o método `compareTo`, também é preciso que a classe `Fraction` declare que implementa essa operação com o funcionamento e propriedades prescritas na interface `Comparable`.

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

  Faça essa alteração e volte a testar o programa.

- Acrescente dois atributos `Fraction.ZERO` e `Fraction.ONE` que correspondam às constantes 0 e 1, em forma de fração. Que qualificadores deve usar na sua declaração? `public`? `private`? `static`? `final`? Corra os testes e confirme que não consegue compilar o programa se descomentar a instrução `Fraction.ZERO = two`.

- Verifique que a adição de frações satisfaz a propriedade comutativa. Acrescente asserções para testar a propriedade associativa da adição e as propriedades comutativa, associativa e distributiva da multiplicação.

Exercício 4.5

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 na classe `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 segredo.

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
java -ea GuessGame
```

Exercício 4.6

Complete o programa `PlayGuessGame` que usa a classe `GuessGame` 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 PlayGuessGame.jar
```

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.