Programação Orientada pelos Objectos

Exame (época de recurso)
2010/07/13

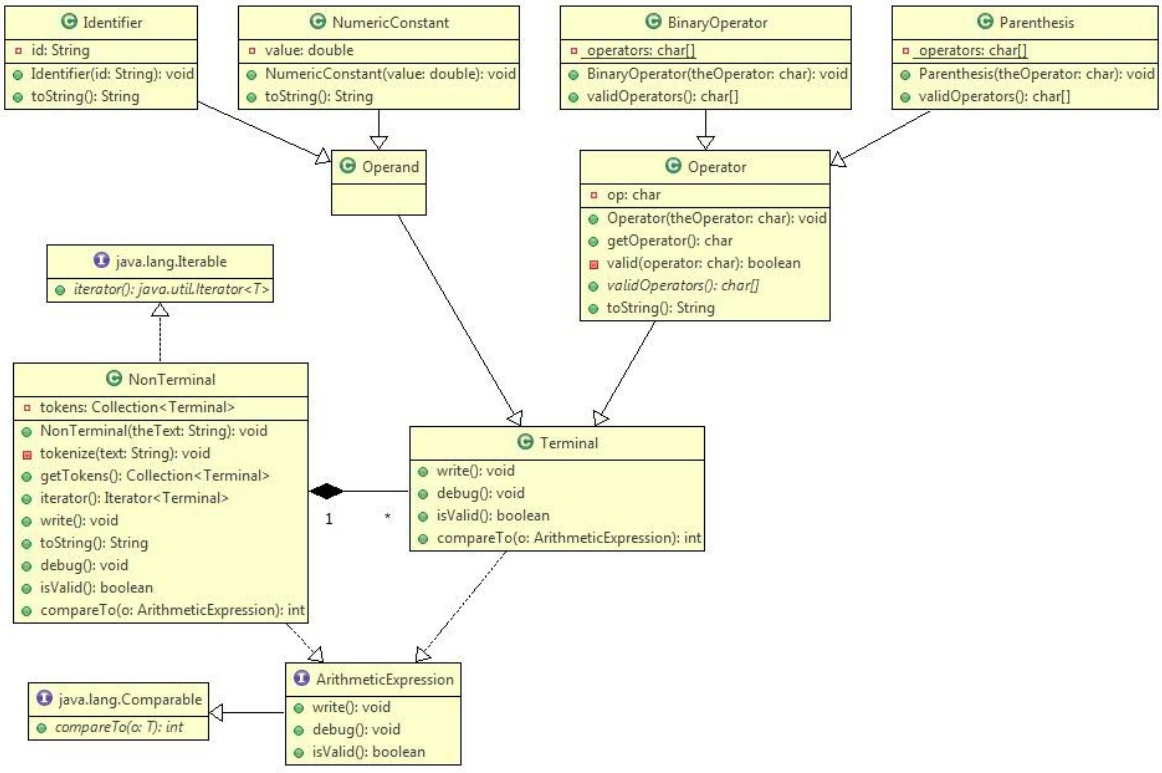
Atenção: A fraude, mesmo quando detectada após a prova, é punida com a reprovação na cadeira.
Sugestão: Resolva o exame utilizando **lápis e borracha**. Evite rasuras!

Os grupos deste exame referem-se todos ao mesmo contexto, que é o representado no diagrama de classes que a seguir se reproduz. Leia este texto explicativo antes de iniciar a resolução das perguntas que são colocadas no verso deste enunciado. Tal como no mundo profissional, é muito importante que perceba o desenho de classes antes de começar a implementação.

O contexto deste exame é o de um **processador de expressões aritméticas**. Para processar uma expressão aritmética é preciso "parti-las" nos seus elementos básicos. Ora uma expressão aritmética é constituída por operandos e operadores (classes **abstractas** Operand e Operator, respectivamente). Na expressão -5 + 3.4 * (x - y), os operandos são o "-5", "3.4", "x" e "y". Os primeiros dois são constantes numéricas (classe NumericConstant) e os últimos dois são identificadores de constantes ou variáveis (classe Identifier). Na mesma expressão, os operadores são o "+", "(", "-" e ")". O primeiro é um operador binário (classe BinaryOperator) e os seguintes são parêntesis (classe Parenthesis). Repare-se que o sinal que afecta o "-5" não é um operador binário, mas sim um operador unário, que neste problema vamos considerar como sendo parte do operando respectivo.

Os operandos e operadores em conjunto são designados por símbolos terminais (classe **abstracta** Terminal), porque neste contexto não se decompõem em mais partes. Já uma expressão como a acima indicada, que pode ser decomposta em símbolos terminais, é designada como um símbolo não terminal (classe NonTerminal). A operação de "partir" um símbolo não terminal nos símbolos terminais que contém é por vezes designada em Inglês por tokenize, quando os símbolos terminais são designados por tokens. Repare na agregação entre as classes NonTerminal e Terminal, que é implementada através da variável tokens. A classe NonTerminal implementa a interface Iterable<Terminal>, para ser possível iterar sobre os seus *tokens*.

Os símbolos terminais e não terminais implementam a interface ArithmeticExpression e esta última estende a interface Comparable<ArithmeticExpression> para que seja possível ordenar um conjunto de expressões numéricas.



Grupo 1 – Terminais

A) Escreva as classes BinaryOperator e Parenthesis, considerando que as variáveis operators correspondem ao conjunto de caracteres válidos que os objectos dessas classes podem armazenar e que são, respectivamente, {'+', '-', '*', '/'} e {'(', ')'}. O construtor dessas duas classes apenas chama o construtor da superclasse.

B) Escreva a classe abstracta Operator em que o construtor deva validar, através de uma asserção (assertion), que o valor do seu argumento op é válido. O método que faz essa validação (valid) deve usar o método abstracto validOperators que é implementado nas subclasses. O método toString devolve uma *string* contendo apenas o valor de op.

C) Implemente a classe Terminal. O método isValid devolve sempre verdade; o método write escreve na consola o resultado da chamada a toString; o método debug escreve o nome da classe correspondente ao objecto a que for aplicado (repare que pode ser qualquer classe concreta descendente de Terminal) seguido de um *tab* e do valor devolvido por toString; o método compareTo realiza a comparação com base nos métodos toString do objecto this e o seu argumento.

D) Escreva a interface ArithmeticExpression.

Grupo 2 – Não Terminais

Considere a classe NonTerminal.

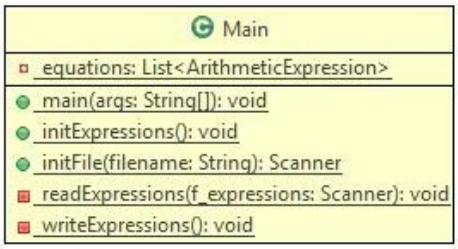
A) Implemente o seu cabeçalho, atributo tokens (não se esqueça da inicialização) e os seguintes métodos: construtor, que chama o método tokenize; getTokens que devolve a colecção de símbolos terminais correspondentes (*tokens*); iterator que devolve um iterador sobre os *tokens*; toString que devolve uma string com a serialização de todos os *tokens*, um por cada linha; write que escreve na consola o resultado da chamada a toString; compareTo que realiza a comparação com base no número de *tokens*.

B) Implemente o método isValid. Um símbolo não terminal é válido quando todos os seus parênteses estão devidamente emparelhados, isto é, fecham todos os que abrem e não podem fechar antes de abrir.

C) Programe o método tokenize que a partir do argumento preenche a colecção tokens. Para simplificar considere que todos os símbolos terminais estão separados por um espaço, como na expressão já atrás referida: -5_+_3.4_*_(_x_-_y_). Repare que devem ser criados objectos de todas as classes concretas descendentes de Terminal.

Grupo 3 – Finalização

A) Crie uma classe de testes testBattery para o JUnit, com dois métodos testTerminals e testNonTerminals, que permitam testar pelo menos uma instância de cada uma das classes concretas deste enunciado.



B) Considere a classe Main acima representada. Programe os métodos initExpressions e readExpressions que lêem a variável equations com base num ficheiro de teste "equations.txt" como o a seguir representado. Note que cada linha pode ser lida com o construtor da classe NonTerminal. Deve fazer o tratamento de excepções na leitura do ficheiro. Só as linhas com expressões válidas são carregadas. Deve ser escrita na consola uma mensagem de carregamento. No caso do ficheiro em baixo a mensagem deve ser "Leu 6 equações válidas e 3 inválidas!".

```
-2 * ( -3 + x + 4y ) - 23
-2 * ( -3 + x + 4y ) - 23 ) -2
2 / ( x - y
2 / ) x - y (
-2 * x + 4y
x
4y
2 + y
2 / ( x - y )
```

C) Programe o método writeExpressions que escreve na consola todas as equações lidas, depois de as mesmas serem ordenadas.