Exercício 5

Testes e Expressões Regulares

Cadeira de
Laboratórios de Programação

2020/2021

Objetivos

- Uso de expressões regulares para reconhecimento de padrões;
- Criação de testes.

Antes de Começar

De modo a poder realizar este exercício, constituído por 2 partes, deverá ter lido os guiões sobre expressões regulares e construção de testes.

O que fazer

Parte I.

Comece por descarregar o zip `alunosExercicio5` da página de LabP. Nele pode encontrar:

- Uma pasta `src` contendo a classe `RecognisePatterns` que deverá ser completada de acordo com o que se descreve abaixo;
- Dois ficheiros `in.txt` e `out.txt` com exemplos de *input* e *output* do método `main` da classe acima.

Complete a classe `RecognisePatterns` com a definição dos métodos a seguir indicados:

1. `public static boolean isJavaClassIdentifier(String s)` – recebe uma *string* `s` e verifica se `s` é um nome de classe válido em Java (relembre-se que em Java os identificadores podem conter os caracteres `_` e `$` e que os nomes de classes devem começar por uma letra maiúscula).

   Este método só deve retornar `true` se a string `s`, no seu todo, é válida como sendo um nome de classe em Java.

2. `public static boolean matchTimeStampLiteral(String s)` – recebe uma *string* `s` e verifica se `s` contém uma constante com a seguinte forma: `hh:mm:ss.sss`
   onde:
   - `hh` representa a hora e `mm` representa os minutos. Qualquer destas componentes deve ser considerada como sendo uma sequência de 1 ou 2 dígitos representando um número no intervalo admissível correspondente ([0-23] para a hora e [0-59] para os minutos);
   - `ss.sss` representa os segundos. A parte inteira desta componente é constituída por 1 ou 2 dígitos, representando um número no intervalo [0,59], e a parte decimal é opcional e tem no máximo 3 dígitos.

- Não devem existir caracteres brancos entre os vários elementos acima referidos mas deve existir pelo menos um caracter branco no início e no final.

Exemplos:

```
8:26:30.53
08:06:05
```

Este método deve retornar `true` sempre que a string `s` contenha uma (ou mais) sequência(s) de caracteres com a forma acima indicada.

3. `public static boolean matchListNotation(String s)` – recebe uma string `s` e verifica se `s` contém uma representação de uma lista num dos seguintes formatos:
   1. `<` $\texttt{elem}_1$`,` $\texttt{elem}_2$`, …,` $\texttt{elem}_n$ `>` – que representa a lista contendo os elementos indicados, separados por virgulas.
   2. $\texttt{elem}_{\texttt{first}}$ `|` $\texttt{lista}_{\texttt{rest}}$ – que representa a lista cujo primeiro elemento é $\texttt{elem}_{\texttt{first}}$, seguido dos elementos na lista $\texttt{lista}_{\texttt{rest}}$

   Considere ainda que:
   - os elementos das listas são números naturais, isto é, uma sequência de um ou mais dígitos.
   - pode haver um qualquer número de espaços em branco antes e após os caracteres delimitadores (isto é, um dos caracteres < ou >) ou antes e após os caracteres separadores (isto é, os caracteres `,` ou `|`).

   Exemplos:

   ```
   < 12, 13, 1, 123, 456>
   < >
   1|<>
   1  |  < 2, 3, 4 >
   1  |  2  |  3  |  4  |  <>
   ```

   Este método deve retornar `true` sempre que a *string* `s` contenha uma (ou mais) sequência(s) de caracteres com a forma acima indicada.

   NOTA: Repare que uma lista representada no formato 2 terminará sempre numa com o formato 1.

4. `public static List<Double> numbersInScientificNotation(String s)` – recebe uma string `s` e reconhece todas as ocorrências de números em notação científica que ocorrem em `s`. Um número na notação científica deve ter o seguinte formato:

`sinal numeroReal E sinal expoente`

onde:

- `sinal` pode corresponder a `+` ou `-` e pode também estar omisso
- `númeroReal` é um número com ou sem parte decimal (a parte inteira tem que ter pelo menos um dígito; não havendo parte decimal deve-se omitir o ponto)
- `expoente` é um número inteiro não negativo, isto é, uma sequência de um ou mais dígitos
- As componentes do número acima referidas não estão separadas por nenhum caracter (não pode haver espaços em branco entre elas)

Exemplos:

```
10.08E-3
+213E4
-0.78954E+21
```

Este método deve ainda converter cada sequência de caracteres, que corresponda a um número no formato acima indicado, no correspondente valor de tipo `Double`, devolvendo a lista com todos esses valores. (sugestão: use o método `Double.valueOf` para fazer essa conversão)
Caso não detete nenhuma ocorrência de número em notação científica, deve devolver a lista vazia.
Para criar a lista pode recorrer a qualquer uma das classes do pacote `java.util` que implementam a interface `List`.

É dado o esqueleto da classe `RecognisePatterns`, que inclui já um procedimento `main` para ler o ficheiro `"in.txt"` e aplicar, sobre as linhas desse ficheiro, os métodos acima referidos, produzindo um outro ficheiro `"out.txt"`. São dados exemplos desses ficheiros.

Não se esqueça de incluir os comentários `javadoc` para cada um dos métodos e de completar o valor da *tag* `@author`, no comentário da classe.

Parte II.

Construa uma classe `Tests` que permita testar os métodos da classe `RecognisePatterns` usando testes `JUnit`.

Consideremos, por exemplo, o primeiro método da lista anterior, `isJavaClassIdentifier`. O nome de uma classe em Java tem que começar por uma letra maiúscula e conter apenas letras, dígitos, _ ou $. Segundo o indicado no guião de testes, temos que ver quais as características úteis e a partir delas definir regiões que particionem o espaço:

Características	Regiões
o nome começa por letra Maiúscula	true ou false (mT, mF)
o nome só tem letras, dígitos, _, ou $	true ou false (nT, nF)
o nome tem K caracteres	K=0, K=1, K>1 (k0, k1, k2)

Poderíamos ter considerado mais (ou menos) características. Quanto mais características considerarmos, maior será o número de combinações de regiões a testar. Temos que optar por um conjunto de características que seja razoável, isto é, que nos permita testar os vários aspetos considerados relevantes, sem no entanto ser excessivo.
Para o método `isJavaClassIdentifier`, com as 3 características indicadas acima, o número total de combinações é 12 (=2*2*3), embora só as 8 indicadas na tabela seguinte sejam viáveis (por exemplo, não é possível começar por letra maiúscula e ter 0 caracteres, assim como também é impossível ter simultaneamente mT, nF e k1):

Combinação de regiões	Caso de teste	Resultado esperado
(mT, nT, k1)	“A”	true
(mT, nT, k2)	“Bb_$1”	true
(mT, nF, k2)	“C+”	false
(mF, nT, k0)	“”	false
(mF, nT, k1)	“d”	false
(mF, nT, k2)	“ef_12”	false
(mF, nF, k1)	“.”	false
(mF, nF, k2)	“_Ef+”	false

A partir da tabela acima poderíamos então definir os métodos de teste. Por exemplo, para o 1º e 3º casos de teste:

```
@Test
/**
* Tests an identifier satisfying:
*   starts with an uppercase letter: true
*   has only letters, digits, _ or $: true
*   has k characters: k = 1
*/
public void testJavaClassIdentifier_mTnTk1 () {
  assertTrue(RecognisePatterns.isJavaClassIdentifier(”A”));
}
```

```
@Test
/**
* Tests an identifier satisfying:
*   starts with an uppercase letter: true
*   has only letters, digits, _ or $: false
*   has k characters: k > 1
*/
public void testJavaClassIdentifier_mTnFk2 () {
  assertFalse(RecognisePatterns.isJavaClassIdentifier(”C+”));
}
```

Deve proceder da mesma forma para os outros métodos a testar.
Para escolher os testes para o último método, `numbersInScientificNotation`, lembre-se que deverá considerar não só o valor do seu parâmetro mas também o resultado do método.
Para cada um dos testes indique, nos comentários `javadoc`, de forma clara, qual a combinação de características que é testada.

O que entregar

Deve entregar apenas os seus ficheiros `RecognisePatterns.java` e `Tests.java`.