ICT/KTH
05-sep-2010/FK

id1006 Java Programming

Assignment 1 - Parser and SimpleMetric

It is recommended that you submit this work no later than Tuesday, 12 October 2010. Solution examples will be presented on 13 October.

This assignment is part of the individual student examination on the course id1006 Java Programming. The assignment is to be done by an individual student. When the assignment has been approved, it corresponds to 1/5th credit of the 7.5 credits (hp) given for the completed course.

HOW TO SUBMIT THE COMPLETED ASSIGNMENT

The completed work is delivered electronically in the form of an email addressed to

java.assignments@fc.dsv.su.se

In an emergency, send the email to fki@kth.se.

Send one email per assignment. Since there are three programming tasks and one essay in this course, a student is expected to submit a total of four emails.

The subject of the email must contain the text 'id1006 Assignment xxx' or 'id1006 Essay' etc.

The body of the email must contain the submitting student's name and optional civic registration number (Sv. 'personnummer'). The sender id on the email is NOT sufficient identification.

The body of the email must also contain all additional information needed to identify the submitted work and the context in which it is being submitted. For example, a re-submission.

Submitted files (e.g. program sources) should be adjoined to the email as one or more attachments.

SOURCE CODE is to be submitted as PLAIN TEXT, ie files that can be compiled by the Java standard development kit (javac). All files necessary to build the program or programs must be submitted together. If you send an archive, let it be ZIP.

Typeset documents (e.g. Ms Word, Open Office, LaTex etc) MUST be submitted in PDF, the Portable Document Format by Adobe. This is currently the optimal way to guarantee cross-platform readability of electronic documents.

Submitted work is expected to be carefully prepared, annotated,

commented and above all original. Where it is not, quotes, citations, and references are to be CLEARLY indicated. Images, graphics and other multimedia products can only be incorporated into the submitted work with the permission of the copyright holder, and the permission must be expressed in the submitted work.

Submitted work will be tested for originality.

The three programming assignments for the fall 2010 instance of the course are all related. Together, they create a simple and extensible application for estimating the readability of english text.

A readability index (of which there are several) is language dependent, statistical and computable. They are usually constructed by computing a ratio between the average number of words per sentence, and the proportion of complicated words to simple words. As a result, they usually return a single figure, like 30, or 14.2.

There is for example the LIX (Sv. "läsbarhetsindex", Eng. "readability index") which is computed thus:

$$lix = \frac{O}{P} + \frac{L * 100}{O}$$

where

- O = the number of words
- L = the number of long words (longer than six characters)
- P = the number of sentences

in the text.

LIX is primarily for swedish texts, but can of course be computed on english too. As can be seen, it is the sum of two values: the average number of words per sentence, and the percentage of long words in the text. This means that long sentences and long words will give a higher value, indicating a text that is harder to read.

Another famous readability metric for english is the Flesch index:

$$Flesch = 206.876 - \left(1.015 * \frac{O}{P}\right) - \left(84.6 * \frac{Y}{O}\right)$$

where O and P are as for Lix, while

- Y = the number of syllables

in the text.

This is a weighted sum of (again) the average number of words per sentence, and the average number of syllables per word. Since more syllables requires more letters in a word, it is quite similar to the

Lix metric. The major difference, however, is that Flesch is continuous over word length, while Lix uses a threshold for long words at six characters.

In order to complete the three assignments, you are given certain Java classes in source code form. You will also create other Java classes to specification, like a parser, readability meters, a syllable counter and main programs.

Whenever possible, you should strive to reuse your classes and code. For example, the parser written for assignment 1 is to be reused without copying or modification in assignments 2 and 3. For the readability meters and the main programs, it is better to copy the source code to a new class name and modify it.

With the first assignment you receive five source files. These are to be used in all assignments (except for ParserTest which is the main program for the first assignment):

Token.java	Superclass for WorkToken and PunctToken.
WordToken.java	Subclass of Token, represents a word.
PunctToken.java	Subclass of Token, represents punctuation.
TextMeter.java	The interface definition implemented by metric modules.
ParserTest.java	A example main program.

Assignment 1 - Parser.java and SimpleMeter.java

The first assignment is to write a parser for english text (like the one you are reading now). The parser only needs to recognise two kinds of structure: words and punctuation.

The readability of the text is measured by a plug-in given to the parser when it is created. This plug-in is a class that implements the interface TextMeter (given as source-code). The parser informs the TextMeter what tokens it has parsed from the text, and the TextMeter measures and remembers the statistics it needs.

The communication between the parser and the TextMeter works like this: for each word and punctuation found, an instance of class WordToken or PunctToken is created, and passed on to the TextMeter. The TextMeter inspects each token and extracts the information it needs from it.

For the parser, you should name the source file Parser.java and the skeleton for the class is:

```
public class Parser {

  public Parser (TextMeter tm) { // constructor with argument
    ...
  }

  public TextMeter getMeter () {
    ...
  }


  public void parse (String s) {
    ...
  }
}
```

The constructor takes an argument, which is a reference to a TextMeter implementation. All the parser needs to know about it, is that it implements the TextMeter interface. The parser should use a local variable to remember the reference, so that it can call it later when method parse(String) is called.

The method

```
  TextMeter getMeter()
```

should return the reference that was given in the constructor. This is because the main program that calls the parser should be able to get the reference back without having to remember it itself.

The method

```
  void parse (String)
```

is called with a string to be parsed. The parser should scan the string, generate instances of WordToken or PunctToken, and send these to the TextMeter, using TextMeter.add (tkn). For example, if the local reference to the TextMeter is a variable called 'meter':

```
  WordToken wtk;
  PunctToken ptk;

  ...
  meter.add (wtk);
  ...
  meter.add (ptk);
```

Simple parsers can be defined and described by a Deterministic Finite State Automaton (DFSA). This is a simple state machine and for the parser needed here only four states are required: start, space, word, and punct. The following table describes the whole automata, including the next character and the action to take:

State	Next character	Actions	Next state
0:start	space	-	start
0:start	punctuation	-	start
0:start	letter	start a WordToken	word
1:word	letter	add char to token	word
1:word	punctuation	send token to meter start a PunctToken	punct
1:word	space	send token to meter	space
2:space	space	-	space
2:space	letter	start a WordToken	word
2:space	punctuation	start a PunctToken	punct
3:punct	punctuation	add char to token	punct
3:punct	letter	send token to meter start a WordToken	word
3:punct	space	send token to meter	space

A traditional and efficient way to iterate over all characters in a string is this:

```
  String s = ...;

  for (int i = 0; i < s.length (); i++) {
    char c = s.charAt(i);

    ...
  }
```

However, while it is NOT possible to do:

```
  for (char c : s) { ... }
```

it IS possible to write:

```
  for (char c : s.toCharArray ()) { ... }
```

BUT! this will require the creation of an array object each time the loop is executed. If this is done rarely, that is no problem. But if it is done for every word in a large text, then the memory requirements per word will more than double, since there will be both the string for the word and the array object holding a copy of each character in the string.

A word in the parser is defined as a string of letters, as indicated by

```
public static boolean Character.isLetter(c)
```

which is a method in the java.lang package. The java.lang package is always available and does not need to be imported.

Space between words and punctuation is defined by

```
public static boolean Character.isSpaceChar(c)
```

However, for punctuation there is no predicate defined, you must write one yourself. Punctuation is defined any of the characters from the set ".!?:;" (period, exclamation mark, question mark, colon, and semicolon). Such a predicate can be written like:

```
protected boolean isPunctuation (char c) {
  if (c == '.') {
    return true;
  }
  else if (c == '!') {
    return true;
  }
  ...
  else {
    return false;
  }
}
```

However, that requires several lines of code. A shorter version would be to use a boolean expression directly in the return statement:

```
protected boolean isPunctuation (char c) {
  return (c == '.') || (c == '!') || ... || (c == ';');
}
```

In fact, there is an even shorter way of doing it, by clever use of one of the methods in java.lang.String. Finding it out is left as an excercise.

Finally, any character which is not a letter, a space, or punctuation, is regarded as invisible and skipped over.

Important note: when you reach the end of the string being parsed, make sure that any token you have been building also is sent to the TextMeter. Otherwise you will not get correct results.

The SimpleMeter

In order to test the parser, you will need to write an implementation of a TextMeter, called SimpleMeter. The SimpleMeter counts two things, the number of words, and the number of sentences (the number of punctuations).

The file should be named SimpleMeter.java and the skeleton looks like:

```
public class SimpleMeter implements TextMeter {

  ...

} // SimpleMeter
```

Since the SimpleMeter class implements the TextMeter interface, all methods in that interface must be given implementations. For example:

```
  int nofWords = 0;

  ...

  public void add (WordToken wt) {
    // Increment the count of words.
  }

  ...
```

The two methods TextMeter.add(WordToken) and TextMeter.add(PunctToken) are used to put information into the meter. The other two methods, TextMeter.properties() and TextMeter.get(String) requires some explanation.

Since readability metrics can differ a lot, and we want to be as general as possible, we let the TextMeter implementation return several values, expressed as a map from keys to values. That is why the return value of properties has that type:

```
import java.util.Map;

...

   public Map<String,String> properties() {
     ...
   }
```

The Map type is an interface defined in the standard library package java.util. Using the interface as a return type, allows the implementation to choose any class which implements that interface to hold the map. A Map, then, is a lookup table where values are stored under keys. The general definition of a Map is

```
  Map<K,V>
```

meaning that the key (K) is the first type and the value (V) is the second type. When we declare that something is a Map, we can choose what the types should be for the key and the value. The most general type of Map would be:

```
  Map<Object,Object>
```

Because all Java objects automatically extends class Object, this declaration says that any type can be a key, and any type can be a

value. However, that is rarely useful. It is much more common that we want to restrict the allowed types to something specific, so that we can be sure of what types are expected. This is especially important when retrieving values from the map. So, when we declare a type to be

```
Map<String,String>
```

we are telling the Java compiler that instances of this type may only have strings as keys, and strings as values. The compiler will then check our code to make sure we do not try to use something else (although null is usually allowed as a special kind of value).

Having the Map type, we also need an implementation for it, something for the SimpleMeter to use to actually store the data. In java.util there is such an implementation:

```
HashTable<K,V>
```

As can be seen, it also can be restricted in terms of which types it accepts (again, it is the Java compiler which enforces this), so a suitable declaration in class SimpleMeter would be:

```
import java.util.Map;
import java.util.HashTable;
  ...
  protected Map<String,String> stats = new HashTable<String,String>();
```

The import clauses at the beginning of the source file are needed so that the compiler understands which Map and HashTable we want. Then we declare a variable called stats (statistics), saying it is protected (it can only be accessed from within class SimpleMeter), and that is has the type Map<String,String>. In the same line, we also initialize it, by creating an instance of a HashTable, restricted in type to match the type of the variable. Since the HashTable class implements the Map interface, it IS a Map, and this is exactly what we want.

It is now quite easy to implement the TextMeter.properties() method, and it simply returns the value of the stats variable:

```
  public Map<String,String> properties () {
    // Load the stats table with its properties
    ...
    return stats;
  }
```

However, before we return the property map (stats), we need to put some properties into it. The Map interface contains a suitable method:

```
Map.put(K key, V value)
```

so in our case all we need to do is to decide on names for the properties and put them in. For example:

```
  ...
  stats.put ("words", /* the count of words */);
  // More properties...
  ...
```

The Map.put() method will overwrite any previous value put in using the same key, so there is no need to remove any old values. The expected property names for the SimpleMeter are "words" and "sentences".

As for the values, you must make sure the expression you put in evaluates to a String type, or the compiler will complain if it cannot resolve the conversion automatically. For all the primitive data types, like int, float, double, char etc, their wrapper classes in package java.lang (Integer, Float, Double, etc) have static toString() methods that will do this. For example, the statement:

```
String s = Integer.toString (4711);
```

would give s the value "4711", which is a string.

> (Note: the wrapper classes exist in order to be able to treat the primitive data types as objects. While you can put a String in a Map, you cannot put an int in there. It is a primitive data type, and not an object, and collections like maps, sets and lists can only contain objects. For that reason instances of e.g. Integer can be created to 'wrap' around the primitive data type value. In addition to that, the wrapper classes also contain useful functions for converting the values to and from strings of text.)

There are some more things to consider here. Since we do not know when someone will call the properties() method, we must make sure that the stats map is updated before we return it. Likewise, we also have the

```
String TextMeter.get(String key)
```

method to implement, and it should of course return exactly the same up-to-date value as can be found in the properties. There are several ways of doing so: one way is to have a separate method which stuffs the current properties into the stats Map, and both properties() and get(String key) calls it. Another is to load the map only in the implementation of the properties() method, and then implement get(String key) by calling properties() first.

The main program - ParserTest.

The source code for this provided. It is really nothing complicated; a new parser is created and given a new SimpleMeter instance. The strings taken from the commandline (the argument to main, an array of String) are each sent to the parser. Then, finally, the meter and the property map are retrieved from the parser and all the properties printed. That last operation is rather involved, so it will be explained in detail. This is what the code looks like at the end of the main() method:

```
for (Map.Entry<String,String> M
       : parser.getMeter ().properties ().entrySet ()) {
  System.out.printf ("%10s : %8s%n", M.getKey (), M.getValue ());
}
```

First of all there is a for-loop, using the rather recent (Java 5) foreach construct:

```
for ( type variable : collection-or-array) { ... }
```

which iterates over all elements in the collection or array (collections are found int the java.util package; Map and HashTable are examples of collections). So the type in the statement above is

```
Map.Entry<String,String>
```

the variable is

```
M
```

and the collection-or-array is a series of method calls that in the end returns a collection. Broken down the method calls look like this:

```
parser      is the reference to the parser instance.

  .getMeter ()   returns the reference to the SimpleMeter given
                 to the parser when it was constructed.

    .properties ()   returns the Map from the SimpleMeter

       .entrySet ()  is a method in the Map interface, and it
                     returns a set of Entry objects. Since a Set is
                     a collection (interface java.util.Set) this
                     works well with the foreach loop.
```

The general declaration of the value returned by Map.entrySet() is

```
Map.Entry<K,V>
```

but since we know that the property map contains keys and values that are all strings, we can specialize this type in the declaration of the iteration variable M, which is why it is declared

```
Map.Entry<String,String>
```

Inside the for-loop, is a call to

```
System.out.printf(String, expression ...)
```

There are other ways of printing to the standard output, like

```
System.out.print( )
System.out.println( )
```

but the printf method offers formatted printing. The first argument to printf is a formatting string, similar in spirit to the printf function in the C language. The printf method is documented in the java.io.PrintStream class and the complete syntax for the formatting string can be found in java.util.Formatter.

The formatting string used here looks like this:

```
  "%10s : %8s%n"
```

Each percent sign starts a formatting field. After the field comes the width of the field and a conversion indicator, in this case 's', for string. The %n conversion generates a newline at the end of the formatted string.

After the formatting string follows two arguments, since we have specified two string conversions in the formatting string. These reference the iteration variable M (which will be referring to the current Map.EntrySet instance in the foreach loop), and from that the key and the value or that pair is retrieved and can be formatted and printed.

Final important points

Do not use packages! We do not need them.

Put all your files for assignment 1 in the same folder.

When you do assignment 2 and 3, put them in their own folders and only with the new files you get or create. Set the CLASSPATH environment variable so that the javac and java commands can find your classes. For example, if your assignment folders are named a1, a2 and a3, then in a Windows environment:

```
SET CLASSPATH=.;..\a1;..\a2;..\a3
```

The first dot includes the current directory, which probably is one of the three folders. The .. syntax means the parent directory to the current directory, and then down again into a1, a2 or a3.

If you do this you will be able to compile and run assignemnts 2 and 3 without having to copy the TextMeter, Parser, and token classes into each folder.

Example output:

```
>java ParserTest "A white fox smiled. Why? Because the hens were out..."
     words :       10
 sentences :        3

>

>java ParserTest "1.2.3.4.5."
     words :        0
 sentences :        0

>
```

Grades for this assignment:

For the E, D and C grades the general grading criteria apply.

For the C grade it is important that you use the proper javadoc syntax in the source code you write. You must document the class, using the @author tag.

You must document every method, explaining what it is for. If the method takes arguments, use the @param tag. If the method returns values, use the @return tag. If you have programmed the method to throw exceptions, use the @throws tag. Remember that a javadoc comment starts with /** and ends with */, and it goes immediately before that which is commented.

For the B and A grades, your solution fulfills the requirement for a C, and you modify the property listing in ParserTest.java so that the properties are printed sorted alphabetically by the key string.

B - an unambiguous algorithmic outline
A - you code it, document it, and it works.

-fk