I/O and Parsing Tutorial
22-02-13

Structure of tutorial

1. Example program to access and write to an XML file
2. Example usage of JFlex

Tasks program

- Program to help people plan and manage their work on a project
- Task class used to represent tasks that a user needs to complete
- Tasks have a description and priority
- Task objects should be written to an XML file
- The XML file should be read and parsed into a human readable format

JDOM

- This program will work with XML files
- We want to write tasks to an XML file
- We want to read tasks from an XML file
- JDOM library is perfect for this
- Interoperates with the Standard API for XML (SAX)
- Interoperates with the Document Object Model (DOM)
- Download from http://jdom.org

Structure of program - main

- Searches for an argument stating where the tasks should be written to i.e tasks.xml
- Creates an ArrayList of Task objects
- Saves to XML file
- Clears ArrayList
- Loads tasks from XML file
- Prints these out

Structure of method - saveToXML

- Takes an ArrayList and XML file as parameters
- Builds up structure of XML file
- Validates the structure of the XML file using a DTD called tasks.dtd
- For each task
  - Create an element
  - Set its description
  - Set its priority as an attribute
  - Append the element to the end of the XML structure

Structure of method - saveToXML

- Format the XML using a pretty format
- Output the XML to the XML file defined in the method's argument
- Print the XML tree to System.out

Structure of method - loadFromXML

- Parameters are an ArrayList and an XML file
- Creates a SAXBuilder to parse the XML document
- Set up a document to receive the in-memory XML file
  - readdoc = builder.build(xmlfile);
- Get the root element
- Get the children of the root and put in a list
- Create new task objects using element text and attributes

Task Class

- Description and priority are private String fields
- Constructor initializes fields
- toString() returns a human readable version of the task
- Getters for the fields

JFlex

Lexing

- JFlex is a lexer
- A lexer breaks a stream of characters that can be read from a file into easier to manage streams of tokens
- Java tokens are typically an object representing an integer of a token type
- Token name – INT
- Token value – 1

Lexing

- Lexers are useful for parsers
- Less objects to deal with:
  - 12 tokens instead of 20 characters
- Tokens contain useful information
- A lexer describes the patterns that can make a token of a particular type using a regular expression

JFlex

- Writing a lexer is tedious
- Lexer generators help overcome this
- These generate lexer code for you
- JFlex is a lexer generator
  - http://jflex.de
- Download and add the .jar to your classpath
  - Linux: Setenv CLASSPATH /path-to-jflex/jflex.jar
  - Windows(add to PATH): Control Panel – System – Advanced – Environment Variables
  - Add ;/path-to-jflex/bin

Setting up JFlex – Windows

- In bin/ jflex.bat has to be editted:
  - JFLEX_HOME must be set to the location where JFlex is installed
  - JAVA_HOME must also be set to the location where you have installed your JDK

JFlex

- Turn a JFlex specification file into a lexer java class:
  - java JFlex.Main lcalc.flex
- Lexer has two constructors:
  - One for a Reader object
  - One for an InputStream object
- Tokens from the Lexer are accessed by the next_token method
- Tokens are defined in sym.java
- End of file returns token sym.EOF

Running JFlex

- In the directory with your .flex file run:
  - jflex lcalc.flex
    - Or
  - Java Jflex.Main lcalc.flex
- This will create Lexer.java
- Ensure sym.java is present and compiled
- Compile Lexer.java
  - javac lexer.java

Structure of lcalc.flex

- %class Lexer tells JFlex to give the generated class the name ``Lexer'' and to write the code to a file ``Lexer.java''.
- %cup switches to CUP compatibility mode to interface with a CUP generated parser.

Structure of lcalc.flex

- %line switches line counting on (the current line number can be accessed via the variable yyline)
- %column switches column counting on (current column is accessed via yycolumn)
- %unicode defines the set of characters the scanner will work on. For scanning text files, %unicode should always be used.

Structure of lcalc.jflex

- The code included in %{...%} is copied verbatim into the generated lexer class source.
- Here you can declare member variables and functions that are used inside scanner actions.
- We define the symbol methods here with positional information

Structure of lcalc.flex - macros

- The specification continues with macro declarations.
- Macros are abbreviations for regular expressions, used to make lexical specifications easier to read and understand.
- A macro declaration consists of a macro identifier followed by =, then followed by the regular expression it represents.
- This regular expression may itself contain macro usages.

Structure of lcalc.flex - macros

- LineTerminator stands for the regular expression that matches an ASCII CR, an ASCII LF or an CR followed by LF.
- WhiteSpace stands for the white space character
- dec_int_lit stands for an integer
- dec_int_id is the ID representing this integer

Structure of lcalc.flex – lexical rules

- These outline actions that are taken when the scanner matches the associated regular expression
- The scanner keeps track of all characters in order to match a regular expression
- Lexical states can also be checked for
- These act like a start condition
- YYINITIAL is predefined
- This is the state that the lexer begins scanning

Regular Expressions

- These are specific patterns that provide a flexible way to match strings of characters.
- In lcalc.flex:
  - dec_int_lit = 0 | [1-9][0-9]*
  - This regular expression matches:
  - 0 or (or is the | symbol)
  - The digit 1,2,3,4,5,6,7,8,9 followed by the possibility of 1-9 repeated any number of times

Regular Expressions

- dec_int_id = [A-Za-z_][A-Za-z_0-9]*
- Matches:
- Alphabetic characters followed by “_”
- Possibly followed by any combination of alphanumeric characters including an underscore
- e.g “s_sc_b”

Structure of lcalc.flex – lexical rules

- <YYINITIAL> {

```
    ";"                { return symbol(sym.SEMI); }
```

- This matches the semi-colon input symbol only if the scanner is in its start state “YYINITIAL”
- When the symbol is matched, the scanner function returns the CUP symbol sym.SEMI

JFlex and CUP

- Download CUP .jar
  - http://www2.cs.tum.edu/projects/cup/java-cup-11a.jar
- To compile a CUP file:
  - java -jar /location-of-jar/java-cup-11a.jar file.cup
- This will create
  - parser.java
  - sym.java
- Then compile them using
  - javac parser.java
  - javac sym.java