Parsing Multiple data items on a single line of text

To extract multiple data items from a line of text we must first split the line into the individual strings representing each item and then convert each item from its string representation. A utility object called a **StringTokenizer** from the **java.util** package is used to split up the String into its individual components called tokens. A particular character called a delimiter must delineate each component in the String. In normal usage, we use the space character as a delimiter. "We read each individual token as a word." Thus, this string contains 8 tokens. However other characters may be used as a delimiter to enable us to treat all sentences as a complete object. For example, the following String uses the colon (**:**) character to distinguish between individual tokens.

"This is an example of text**:**345**:**another sentence"

The String above contains 3 tokens

1. "This is an example of text
2. 345
3. another sentence

The space character is no longer significant and the sentence as a whole becomes the token. The delimiter character set used by a **StringTokenizer** is a space by default but this default value may be changed when the **StringTokenizer** is created. One can also specify if the delimiter itself is to be a token or simply ignored. Once a **StringTokenizer** has been created for a particular String one can find how many tokens there are in the String and iterate through the String returning each token. If there are only a few tokens in the line then the **nextToken()** function can be called manually. Otherwise the **nextToken()** function may be placed in a loop dependent upon the number of tokens in the line.

Implement ex5.java Java Program

Before implementing the example using the **StringTokenizer** class in ex6, we need to have a text file that will contain data of some specific structure, e.g., <Name> <Last Name> where the name and last name are separated by a space or <x-axes>,<y-axes> where the x- and y-axes are separated by comma. This text file will be then used in ex6.java. Write a Java program with the name ex5.java, that it will output 10 **x** and **y** coordinates in a text file with the name "points.txt" with some format. The **x** coordinates should start from 1 to 10 and the **y** coordinates should start from 10 to 1. For example, the generated "points.txt" file should look like this:

1,10
2,9
3,8
4,7
5,6
6,5
etc…

Notice that the format of this example is **< x,y >,** e.g., separated by comma. You can use any structure you like.

Extracting the Coordinates from “points.txt” File

The following example **ex6** demonstrates how to read in a file with a specific structure. Each line of the file contains two int values representing the **x** and **y** coordinates of a point in a graph. The **StringTokenizer** is used to convert an individual string (i.e., a line from the text file) and return the number of points (i.e., two integers). Notice that the program uses the **java.util** package**.** Implement and run ex6.java. Also, make sure that you modify the program **ex6** (where and if necessary) to match the structure you have used for the text file generate in **ex5.**

```
import java.io.*;
import java.util.*;

class ex6 {
        public static void main(String[] args)  {
                if (args.length != 1) {
                   System.err.println("Usage: java ex6 <file>");
                   System.err.println("you must supply the name of the file");
                   System.exit(0);
                } else {
                   String fileName = args[0];          // the first argument

                   // now create the filestream and connect PrintWriter
                   // the value true enables autoflushing
                   try {
                           FileReader         fin = new FileReader(fileName);
                           BufferedReader din = new BufferedReader(fin);

                           System.out.println("Data in file " + fileName + "is ");

                           //read from the file
                           String line = null;   // line of text
                           int numPoints = 0;  // running total of points

                           while ((line = din.readLine()) != null)  {
                              // here we have read in a line of text
                              // now parse line to extract data and print it out to the screen
                               StringTokenizer st = new StringTokenizer(line, ",");

                               int x = Integer.parseInt(st.nextToken().trim());
                               int y = Integer.parseInt(st.nextToken().trim());

                               numPoints++;
                               System.out.println("\tx=" + x + ", y=" + y + ", x*y=" + (x*y));
                           }
                           System.out.println("There are " + numPoints + " points in this file");
                           din.close();         // close the stream
                   } catch (IOException e) {
                           System.err.println("Error! - " + e.getMessage());
                   }
                }
        }
 }
```

```
while ((line = din.readLine()) != null)
{
        // here we have read in a line of text
        // now parse line to extract data and print it out to the screen

        StringTokenizer st = new StringTokenizer(line,",");

        int x = Integer.parseInt(st.nextToken().trim());
        int y = Integer.parseInt(st.nextToken().trim());
        numPoints++;

        System.out.println("\tx=" + x + " y=" + y);
}
```

This code fragment from ex5.java parses a line into two co-ordinate values. The values are given as comma separated text. Because only two values are required the **nextToken()** function is called twice. The **trim()** function removes all leading and trailing whitespaces from a string. Therefore, the actions on the colored lines are to return the next token, trim it and then convert it into an int which is stored in variable x.

```
while ((line = din.readLine()) != null)
{
        // here we have read in a line of text
        // now parse line to extract data and print it out to the screen

        StringTokenizer st = new StringTokenizer(line,":");

        while (st.hasMoreTokens()) {
                int x = Integer.parseInt(st.nextToken().trim());

                // do something with value

        }
}
```

This code is an example of how to extract a variable number of tokens from string separated by colon (**:**)

Function **hasMoreTokens()** returns true as long as there are tokens left in the String. The result of this function determines the terminating condition of the loop.

```
while ((line = din.readLine()) != null)
{
        // here we have read in a line of text
        // now parse line to extract data and print it out to the screen

        StringTokenizer st = new StringTokenizer(line,":");

        int nTok = st.countTokens();
        for (int i=0;  i < nTok ; i++)  {
                int x = Integer.parseInt(st.nextToken().trim());

                // do something with value

        }
}
```

Alternatively, a for loop could be used after first establishing the actual number of tokens in the line. See the Java Documentation for **countTaken()** function.

Reading from the Keyboard

Because the stream **System.in** is an *InputStream*, its functionality is limited to either reading a single byte of data or reading an array of bytes. But do not forget that whilst using the ASCII character set reading a single byte is the same as reading a char this is not true when using Universal Character sets where each character is 2 bytes long. Therefore, for compatibility it is recommended that the standard **InputStream System.in** be attached to a **BufferedReader** to allow lines to be read in.

For top efficiency, consider wrapping an InputStreamReader within a BufferedReader. For example:

InputStreamReader st = **new** InputStreamReader(System.in);
BufferedReader in = **new** BufferedReader(st);

Do not forget that because the keyboard stream is character based any numbers to be read in must be read in as strings and then converted. Implement and run ex7.java.

```
import java.io.*;
import java.util.*;

class ex7 {

	//declare constant Strings
	final static String PROMPT = "Enter data: ";
	final static String QUIT = "quit";
	final static String PARAMS = "You should supply three values";

	public static void main(String[] args) {
		//variables
		BufferedReader kbd = new BufferedReader(newInputStreamReader(System.in));
		StringTokenizer st = null;
		String reply = null;
		double width = 0.0, height = 0.0, length = 0.0, volume = 0.0;

		ex7.printIntro();

		// loop forever
		while (true) {
			try {
				System.out.print(PROMPT);

				reply = kbd.readLine().trim();
				// test if we have to quit
				if ( reply.startsWith(QUIT) ) {
					break;
				} else {
					// parse items in line
					st = new StringTokenizer(reply);
					if (st.countTokens() < 3 ) {
						System.out.println(PARAMS);
					} else {
						width = Double.valueOf(st.nextToken().trim()).doubleValue();
						height = Double.valueOf(st.nextToken().trim()).doubleValue();
						length = Double.valueOf(st.nextToken().trim()).doubleValue();
						volume = width * height * length;
						System.out.println("Volume of this cuboid is " + volume);
					}
				}
			}
			catch (IOException ioe){} // do nothing if error
		}
	} //end of main method

	/*A method called in the main*/
	static void printIntro() {
		System.out.println("A simple program to calculate the volume of a Cuboid");
		System.out.println("When prompted please supply three values in the order");
		System.out.println("Width then Height then Length separated by spaces");
		System.out.println("or type quit to finish");
		System.out.println();
	} //end of printIntro method

} // end ex7 class
```

Notice that the **ex7.printIntro()** method is of type static. A static method belongs to the class, and you do not have to create an instance of the class to access the static method whereas a non-static method belongs to an object of the class and you have to create an instance of the class to access the non-static method.

For example, **ex7.printIntro()** is invoked in the main method without referencing any instances of the ex7 class. Since it is a method of the class it can be also written simply as **printIntro()**

Suppose that the **printIntro()** was of a non-static type:

```
/*A method called in the main*/
public void printIntro() {
        System.out.println("A simple program to calculate the volume of a Cuboid");
        System.out.println("When prompted please supply three values in the order");
        System.out.println("Width then Height then Length separated by spaces");
        System.out.println("or type quit to finish");
        System.out.println();
}
```

Then the **printIntro()** method is not a method of the ex7 class anymore but it is a method of the instances of ex7 class. Therefore, the non-static method has to be invoked by an instance of the ex7 class otherwise compilation errors will occur. For example:

```
ex7 ex7instance = new ex7();
ex7instance.printIntro();
```

Key points of ex7

Declaration of constants using the keyword **final**.
Define other methods in the class.
Static methods and variables are used for efficiency. This is because they are allocated only once in memory.
Non-static methods/variables are allocated multiple times depending on the number of the instances created.