Object Oriented Software Design

I/O subsystem API

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

October 28, 2010

Outline

1. String
2. I/O and files
3. ArrayList
4. Exercises

Outline

1. String
2. I/O and files
3. ArrayList
4. Exercises

Utilities for manipulating strings

- Since you will have to analyse strings to implement your program, let's have a closer look at class String
- The String class is immutable, so that once it is created a String object cannot be changed.
  - The String class has a number of methods that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

Utilities for manipulating strings

- Usually, methods that are used to “read” the value of some property of a class are called accessor methods
  - method **length()** returns the number of characters in a string
  - To read the character at position **i**, we can call **charAt(i)**
  - You can copy a substring of a string into an array of characters as follows:

```
String mystring = "This is a lecture";
char[] temp = new char[5];
mystring.getChars(5, 10, temp, 0);
```

  - It means: copy from the 5^{th} character (included) to the 10^{th} character (excluded) into **temp** starting at position 0
  - After the copy, temp contains `"is a "`
- If you want to obtain another string, it is possible to use substring:

```
String mystring = "This is a lecture";
String sub = mystring.substring(5,10);
```

Concatenating strings

- The String class includes a method for concatenating two strings:

```
string1.concat(string2);
```

  - This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in:

```
"My name is ".concat("Rumplestiltskin");
```

  - Strings are more commonly concatenated with the + operator, as in

```
"Hello," + " world" + "!"
```

Formatted printing

- In Java, you can output formatted printing using **System.out.printf()**, as follows:

```
System.out.printf("A float %f, and an integer %d", 3.754, 20);
```

- You can also use the **format()** method of class String:

```
String fs;
fs = String.format("The value of the float variable is %f, " +
                   "while the value of the integer variable " +
                   "is %d, and the string is %s", floatVar, intVar,
                   stringVar);
System.out.println(fs);
```

Converting strings into numbers

- The **Number** subclasses that wrap primitive numeric types (Byte, Integer, Double, Float, Long, and Short) each provide a class method named valueOf that converts a string to an object of that type

```
public class ValueOfDemo {
    public static void main(String[] args) {
        if (args.length == 2) {
            Float a = Float.valueOf(args[0]);
            Float b = Float.valueOf(args[1]);

            float c = Float.parseFloat(args[0]);
            float d = Float.parseFloat(args[1]);

            System.out.printf("a = %f", a.floatValue());
            System.out.printf("b = %f", b.floatValue());
            System.out.printf("c = %f", c);
            System.out.printf("d = %f", d);
        } else {
           System.out.println("Insert two command-line arguments");
        }
    }
}
```

Other methods

- The String class has a large amount of different methods for manipulating strings:
  - searching character, replacing substrings, etc.
- Please refer to the Java 6 API to get a complete documentation for String

Outline

1 String

2 I/O and files

3 ArrayList

4 Exercises

The File class

- The File class does not represent a file, but one or more file names.
  - It is used to get the list of files in a directory, as in `./examples/08.java-examples/DirList.java`
- Explanation:
  - the **list()** method of class File needs as argument an object of type **FilenameFilter**, which is a very simple interface:

```
public interface FilenameFilter {
  boolean accept(File dir, String name);
}
```

  - The **list()** will call the **accept()** on every file contained in the directory, to see if a file name is “acceptable”,
  - if **accept()** returns **true** the name is inserted in the list, otherwise it is not
- This technique is called callback

Making directories

- It is possible to use the File class to create directories, see if a file exist, get the file type, etc.
- `./examples/08.java-examples/MakeDirectories.java`

Input and output

- The Java library classes for I/O are divided by input and output
  - everything derived from the InputStream or Reader classes have basic methods called **read()** for reading a single byte or array of bytes
  - everything derived from OutputStream or Writer classes have basic methods called **write()** for writing a single byte or array of bytes
  - However, you won't generally use these methods; they exist so that other classes can use them - these other classes provide a more useful interface
  - Thus, you'll rarely create your stream object by using a single class, but instead will layer multiple objects together to provide your desired functionality.
  - The fact that you create more than one object to create a single resulting stream is the primary reason that Java's stream library is confusing.
- The rest of the slides are just descriptions of examples

Reading input by line

```
BufferedReader in = new BufferedReader(
    new FileReader("IOStreamDemo.java"));
String s, s2 = new String();
while((s = in.readLine())!= null)
    s2 += s + "\n";
in.close();
```

- Comment
  - **in** represents the file handle, we open file IOStreamDemo.java
  - we read one string at time, and append all strings in **s2**.
  - The reading is done using buffering
    - i.e. a block of data is read in an internal buffer, and then we read line by line from the buffer
  - This technique is called decorator pattern or wrapper pattern

Reading from std input

```
BufferedReader stdin = new BufferedReader(
    new InputStreamReader(System.in));
System.out.print("Enter a line:");
System.out.println(stdin.readLine());
```

- Comment:
  - In this case, we read one line from the standard input (the keyboard), represented by object **System.in**
  - This same code can be used in the assignment

Reading from a string

```
StringReader in2 = new StringReader(s2);
int c;
while((c = in2.read()) != -1)
    System.out.print((char)c);
```

- It is possible to treat a string as a file
  - In the example, we read one character at time from the string
- It is also possible to “unread” one character, as follows:

```
String s = "This is a string!";
PushbackReader r = new PushbackReader(new StringReader(s));
char c = (char)r.read();   // read character 'T'
r.unread('P');
System.out.println((char)r.read()); // prints 'P'
```

- In some cases, this may be useful for low-level parsing of strings

Writing onto a file

```
try {
    BufferedReader in4 = new BufferedReader(
        new StringReader(s2));
    PrintWriter out1 = new PrintWriter(
        new BufferedWriter(new FileWriter("IODemo.out")));
    int lineCount = 1;
    while((s = in4.readLine()) != null )
        out1.println(lineCount++ + ": " + s);
    out1.close();
} catch(EOFException e) {
    System.err.println("End of stream");
}
```

- The previous example reads one line at time from a string, and writes on a output file
  - **PrinterWriter** is used to output text files, it wraps a **BufferedWriter** which wraps an output file writer
  - If you need to write a binary file, you only need to remove the **PrinterWriter** class, and only use the **BufferedWriter**.
  - The explicit **close()** for **out1** is needed otherwise the buffers don't get flushed, so they're incomplete

Outline

1. String
2. I/O and files
3. ArrayList
4. Exercises

Arrays

- We have seen arrays
  - Arrays have fixed length: once you create an array, it is not possible to add further elements to it
- Variable-length arrays are supported by class ArrayList
  - Three constructors:
    - default constructor (an empty array),
    - a constructor that takes an integer (the initial capacity, but the array is still empty)
    - a constructor that takes a **Collection** of object
    - ArrayList may only contain references to Objects
  - API: `http://download.oracle.com/javase/1.4.2/docs/api/jav`

Example

ArrayListExample.java

```
import java.util.*;

class ArrayListExample {
    public static void main(String args[]) {
        ArrayList al = new ArrayList();
        for (int i=0; i<args.length; i++) al.add(args[i]);

        System.out.println("al.size() = " + al.size());
        for (int i=0; i < al.size(); i++) {
            System.out.println(al.get(i));
        }
        al.add("This is the last one");
        al.add(0, "this is the first one");
        System.out.println("al.size() = " + al.size());
        for (int i=0; i < al.size(); i++) {
            System.out.println(al.get(i));
        }
    }
}
```

Outline

1. String
2. I/O and files
3. ArrayList
4. Exercises

Exercises

- Parenthesis matching
  - Write a class that provides a static method to find matching parenthesis in a String
  - The method takes as input a string, and the position of the first open parenthesis and returns the position of the closing parenthesis

```
String s = "5 * (4 + 2) / 2";
Parenthesis.get(s, 4);     // returns 10
```

In fact:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5		*		(	4		+		2	)		/		2

  - Pay attention to nested parenthesis:

```
String s = "5 * (4 / (2 - 1) - 2) / 2";
Parenthesis.get(s, 4);     // returns 20, not 15
```

Errors

- The function must:
  - raise an exception called **UnmatchedParenthesisException** if it cannot find a matching closing parenthesis
  - raise an exception called **NotAParenthesisException** if the initial position does not contain a left parenthesis symbol “(”
- Write the function a set of at least 5 tests that check the correctness of your implementation
  - Two tests must check that the exceptions are correctly raised
  - One test checks for simple parenthesis
  - Another one checks for 3 levels of parenthesis nesting
  - The last one checks for a matching parenthesis as last character

Use your utility

- Now you should use your utility on a text file
  - Read the file line by line
  - For every line:
    - Print the line on screen
    - below, the number of outer groups of parenthesis, and for every group, and the number of contained groups
    - If there is an error, prints the line number and the error message, and continues with the next line
  - Example:

```
(a + (b+c) + (a + (g+h)))
>>> 1 groups "a + (b+c) + (a + (g+h))"
>>> 2 groups "b+c", "a + (g+h)"
>>> 0 groups
>>> 1 group "g+h"
>>> 0 groups
```