Chapter 1

Using Regular Expressions within Java

1.1 Introduction

Since Java version 1.4, the Java platform has offered extensive support for regular expressions. Regular expressions can, in most cases, be used *implicitly* through support built in to the `java.lang.String` class, or *explicitly* through *patterns* and *matchers*, for a greater degree of control. Java supports:

- testing whether a string of characters matches a regular expression pattern,
- splitting a string around occurrences of a regular expression pattern (similar to, but more powerful, than the historic `StringTokenizer` class' functionality),
- searching for regions within a body of text that match a regular expression, and
- replacing all occurrences of text that matches a regular expression pattern with some other piece of text

Regular expressions support very complex matching behaviour, whilst keeping the amount of code required to accomplish this to a minimum.

1.2 Regular expressions

Regular expressions are used to match text to certain patterns. They are supported by a wide range of tools ranging from many editors (including vi, emacs, jedit, ...) to text processing tools like Perl, grep, sed and awk. They can be used in technologies such as XML Schema to validate XML data, the Java programming language, and even via ECMAScript (JavaScript)in web browsers.

1.2.1 Matching on characters by defining a character class

Regular expressions use the concept of a character class to specify a set of characters against which another character is matched. To be able to check whether a character falls within a particular class of characters, one defines the class of characters within square brackets.

```
[ABC]
```

matches on either “A”, “B” or “C”.

These character classes can now be used within a more general regular expression. For example

```
[vs]ector
```

matches on either “vector” or on “sector”.

Similarly,

```
[sg][oa]ng
```

matches on “song”, “sang”, “gong” and “gang”.

1.2.1.1 Matching on a range of characters

To match on a range of characters, we use a − between the two characters which delimit the range. For example

```
[A-Z]
```

matches on any capital letter of the western alphabet and

```
[Ee]xercises [1-9]
```

matches one exercises 1 to 9 irrespective of whether “exercise” starts with a lower or upper case letter.

To match on any letter or numeral we could use

```
[0-9a-zA-Z]
```

Note
When matching on a hyphen, `-`, place the hyphen as first character in a character set so that it is not interpreted as range. Thus

```
[-+*/^!&|]
```

matches on any one of these characters while

```
[+-*/^!&|]
```

matches on any character between “+” and “*” or on “/”, “^”, “!” “&” or “|”.

1.2.1.2 Excluding certain characters

At times it is simpler to specify which characters should *not* be included than to specify which should. To this end regular expressions provides a *circumflex* character, ^. Inserting a ^ as first character within the class specifier declares a class whose characters are all the characters not specified. Thus

```
[^0-9]
```

specifies a class of all characters except numerals.

1.2.1.3 Matching on any character

At times we want to match on any character. Regular expressions use a `.` to specify that a particular position in a potential match may be filled by any character.

For example,

```
codeA.
```

would match, for example, on “codeA1”, “codeAX”, “codeA#” or even on “codeA ”.

1.2.1.4 Escaped characters

Because certain characters have a special meaning within regular expressions, we need to have some way to request these characters literally. This is done via escaped characters which are specified by a backslash, `\`, followed by the character which should be taken literally. Below we list a few examples:

```
\.  \{  \}  \[  \]  \(  \)
```

1.2.1.5 Pre-defined character classes

Most regular expressions implementations support a common set of pre-defined character classes. These include:

Character class	**Matches**
\d	A digit: `[0-9]`
\D	A non-digit: `[^0-9]`
\s	A whitespace character: `[ \t\n\x0B\f\r]`
\S	A non-whitespace character: `[^\s]`
\w	A *word* character: `[a-zA-Z_0-9]`
\W	A non-word character: `[^\w]`

Table 1.1: Regular expressions pre-defined character classes

1.2.2 Multiplicity constraints

So far we gave no optionality on the multiplicity of the characters we are matching on. If we wanted to match, say, on 2 numbers we would have said

```
[0-9][0-9]
```

But, what if there are very many occurances or even worse, what if we want to match on a variable number of occurances.

To this end regular expressions provide multiplicity constraints which are specified directly after the element to which they apply. The multiplicity constraints supported by regular expressions are listed in the following table:

Multiplicity constraint	**Description**
`?`	Zero or one.
`*`	Zero or more.
`+`	One or more.
`{n}`	Exactly `n`.
`{n,}`	`n` or more.
`{n,m}`	Between `n` and `m`.

Table 1.2: Multiplicity constraints supported by regular expressions

Thus,

```
[A-Z]{3} [0-9]{3} GP
```

matches on Gauteng number plates which start with 3 capital letters followed by a space followed by 3 numbers followed by a space which finally is followed by the characters “GP”.

As a second example, assume you want to match on any XML (or HTML or SGML) tag (markup). This could be done with the following regular expression:

```
<[a-zA-Z_].*>
```

which specifies that the tag name must start with a letter or an underscore character and may contain any number of further arbitrary characters -- this is admittedly not quite true but will work most of the time.

Consider next the following regular expression:

```
lo+ng
```

This will match on “long”, “loong”, ..., “looooooooooong”, and so forth.

1.2.3 Positional characters

Regular expressions provide limited support for matching at particular positions in a line. Specifically, they support matching a pattern either

- at the beginning of the line via a `^`, or
- at the end of a line via a `$`.

For example,

```
^ *<!--
```

matches on any XML/HTML comment which is preceded by nothing else but an arbitrary number of spaces.

In a similar way we can look for a certain pattern at the end of a line. For example, if we want to remove all trailing spaces, we could use the following regular expression to find them:

```
" +$"
```

In the above expression we added inverted commas to highlight the space in the regular expression.

1.2.4 Optional matches and groupings

At times one wants to match on one of a number of options. This is supported in regular expressions via the *or* operator, `|`.

Options are often used in conjunction with a grouping construct. One uses parantheses to group elements of a regular expression.

For example, to match on a time using 12-hour format with AM/PM specifiers, we could use the following regular:

```
([0-9]|1[012]):[0-5][0-9] (AM|PM)
```

This would match, for example, on `9:00 AM` and `11:22 PM`. The expression contains two options grouped together via parentheses:

1. The time contains either a single digit between 0 and 9 for the hour or two digits where the first digit is a 1 and the second is between 1 and 2.
2. The time digits are followed by either an `AM` or a `PM`.

1.2.5 Greedy versus non-greedy matching

By default, pattern matching is greedy, which means that the matcher returns the longest match possible. For example, applying the pattern `a.*c` to `bcabcabcab` matches `abcabc` and not `abc`.

To do nongreedy matching, a question mark must be added to the quantifier. For example, the pattern `a.*?c` will find the shortest match possible yielding `abc`.

1.3 Regular expressions support built-in to java.lang.String

`java.lang.String` contains services that support several use-cases based on regular expressions. These may conveniently be invoked on any instance, and in many cases offer a simpler, more compact alternative to the explicit usage of Patterns and Matchers in the `java.util.regex` package.

1.3.1 Testing whether a String matches a pattern

`java.lang.String` provides the service

```
public boolean matches( String regex )
```

which returns true if the String instance itself matches the given pattern. For example,

```
String carRegistration = "KLM 535 GP";

boolean isGautengRegistration =
  carRegistration.matches("[A-Z]{3} [0-9]{3} GP");
```

will test whether the string is a Gauteng, South Africa vehicle registration number.

1.3.2 Replacing patterns

`java.lang.String` provides the services

```
public String replaceAll( String regex, String replacement)
public String replaceFirst( String regex, String replacement)
```

which allow the replacement of all, or the first, part of the string which matches the given regular expression pattern respectively.

The following example replaces all vehicles with the word “vehicle”:

```
String s = "John drove his blue car right into the " +
        "freight aircraft, causing a spectacular explosion that " +
        "severely damaged the tank.";

String s2 = s.replaceAll("(car|aircraft|tank)", "vehicle");
System.out.println( s2 );
```

to produce the following output:

```
John drove his blue vehicle right into the freight vehicle,
causing a spectacular explosion that severely damaged the vehicle.
```

The next example replaces only the first occurrence of “en”:

```
String k = "The planned supported languages were 'af', 'en' " +
    "and 'fr', although the contractor only managed to implement " +
    "'en' in its entirety.";

String k2 = k.replaceFirst("en", "English (en)");
System.out.println( k2 );
```

to produce the following output:

```
The planned supported languages were 'af', 'English (en)' and 'fr',
although the contractor only managed to implement 'en' in its entirety.
```

1.3.3 Splitting a String around a regular expression

We can *split* (or *tokenize*) a *java.lang.String* by breaking it up into an array of smaller strings around all parts that match a given regular expression, through the services:

```
public String[] split( String regex )
public String[] split( String regex, int limit )
```

The overloaded version allows the developer to place an upper limit of the number of tokens the string should be broken up into - after which all remaining characters form part of the last token.

For example, the following list of colours are split around a range of possible delimiters:

```
String s = "Red,Green,Blue&Purple.";
String[] colours = s.split("[,&\\.]");
for (String colour : colours)
{
  System.out.println(">" + colour + "<");
}
```

to producing the output:

```
>Red<
>Green<
>Blue<
>Purple<
```

Note
The double-backslash in the String literal regular expression is necessary: One backslash to escape the period symbol, which has special meaning to regular expressions (matches on any character), and another to escape the backslash itself, which has special meaning to java (as an escape character).

As another example, a simplistic method to tokenize the two parts of an e-mail address is shown below:

```
String e = "info@solms.co.za";
if (e.matches(".*@.*"))
{
  String[] a = e.split("@");
  System.out.println("User:   " + a[0]);
  System.out.println("Domain: " + a[1]);
}
```

which produces the output:

```
User:   info
Domain: solms.co.za
```

1.4 Regular expressions support through patterns and matchers

The classes `Pattern` and `Matcher` in the package `java.util.regex` offer more powerful and often higher-performing regular expressions capability than the convenience service provided by `java.lang.String`.

If this added functionality is not needed, however, it is recommended to use the aforementioned convenience services due to their simplicity. Specifically, Patterns and Matchers are typically used when

- for performance reasons, the developer wishes to compile a regular expression pattern once, and re-use it in compiled form

- the developer wishes to move through an input string in a stateful manner, iteratively finding and/or replacing sections, with the ability to change the regular expression pattern on the fly, reset state, etc.
- the regular expression contains *capturing groups*, sub-regions of the regular expression which can be referred to at a later stage for more involved searching or replacement

1.4.1 Patterns

A `java.util.regex.Pattern` represents a compiled regular expression, and serves as a factory for `Matcher` objects. Patterns are re-usable, and are typically created with the static `compile(String)` class service:

```
// Matches South-African Identity numbers
Pattern saIdNo = Pattern.compile("[0-9]{13}");
```

1.4.1.1 Creating case-insensitive patterns

As a special case, a case-insensitive version of a matcher can be created by using the factory method that accepts a *bit mask* of "flags" as second argument. To utilise ASCII-text case insensitivity, the `Pattern.CASE_INSENSITIVE` constant is used. To enable general Unicode case-insensitivity, it is used in conjunction with the `Pattern.UNICODE_CASE` flag. For example,

```
Pattern animalName = Pattern.compile("dog|cat|bird",
  Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE);
```

matches on any case variation of the three animal names.

Note
In practise, most regular expressions do care about case, with the case-insensitive parts rather specified as part of the regular expression itself.

1.4.2 Matchers

A `java.util.regex.Matcher`, created from a `java.util.regex.Pattern`, is an engine that performs match operations on a character sequence (such as a `java.lang.String`) by interpreting a Pattern.

A matcher is typically created, from a Pattern, to perform matching operations on a single sequence of characters, and thereafter discarded. For example:

```
// Matches telephone numbers in the form "(011) 555-5555'
Pattern telephoneNumber = Pattern.compile("\\([0-9]{3}\\) [0-9]{3}-[0-9]{4}");

// Create a matcher, which allows us to find, replace, etc
Matcher teleMatcher = telephoneNumber.matcher( someInputString );
```

1.4.2.1 Testing whether the whole input string matches the pattern

The `matches()` service returns a boolena, indicating whether the entire input matches the pattern from which the matcher was compiled. For example:

```
// Matches telephone numbers in the form "(011) 555-5555'
Pattern telephoneNumber = Pattern.compile("\\([0-9]{3}\\) [0-9]{3}-[0-9]{4}");

// Create a matcher, which allows us to find, replace, etc
Matcher teleMatcher = telephoneNumber.matcher("(012) 555-5555");

if (teleMatcher.matches())
```

```
{
  //....
}
```

1.4.2.2 Finding matching occurrences in an input string

A Matcher can be used to repeatedly find the next matching character sequence in an input string. The `find()` service attempts to find the next matching sequence, and returns a `boolean` indicating whether a match was found.

If a match is found, more information can be obtained with the `start() :int`, `end() :int` and `group() :String` services.

- **start()** returns the index of the first character of the text that matched the pattern after the last `find()` was performed
- **end()** returns the index of the last character of the text that matched the pattern after the last `find()` was performed
- **group()** return the character sequence that matched the pattern

For example, to find all the telephone numbers in a piece of text, the following could be used:

```
String s = "I tried to call you on (011) 555-5555 but " +
  "failed to reach you. Should I have tried (013) 123-4567 " +
  "or (012) 321-7654 instead?";

// Matches telephone numbers in the form "(011) 555-5555'
Pattern telephoneNumber = Pattern.compile("\\([0-9]{3}\\) [0-9]{3}-[0-9]{4}");

// Create a matcher, which allows us to find, replace, etc
Matcher m = telephoneNumber.matcher( s );

// Find each telephone number in the input text
while ( m.find() )
{
  String telNo = m.group();
  System.out.println("Tel: " + telNo);
}
```

which produces the following output:

```
Tel: (011) 555-5555
Tel: (013) 123-4567
Tel: (012) 321-7654
```

1.4.2.2.1 Using capturing groups

Capturing groups are groups of characters in a regular expression indicated with parentheses. The are *numbered*, and are done so by counting their opening parentheses from left to right. In the regular expression `((A)(B(C)))`, for example, there are four such groups:

1. ((A)(B(C)))
2. (A)
3. (B(C))
4. (C)

Capturing groups are so named because, during a match, each subsequence of the input sequence that matches such a group is saved. The captured subsequence may be used later in the expression, via a *back reference*, and may also be retrieved from a `Matcher` once the match operation is complete.

To find the character sequence within a match that matches a particular capturing group, it can be requested by number, through the `Matcher.group(int) :String` service. For example, if, for each found telephone number, we wished to distinguish between the area code and local number, we could do so easily by adding two capturing groups to the regular expression:

```
String s = "I tried to call you on (011) 555-5555 but " +
    "failed to reach you. Should I have tried (013) 123-4567 " +
    "or (012) 321-7654 instead?";

// Matches telephone numbers in the form "(011) 555-5555'
Pattern telephoneNumber =
  Pattern.compile("\\(([0-9]{3})\\) ([0-9]{3}-[0-9]{4})");

// Create a matcher, which allows us to find, replace, etc
Matcher m = telephoneNumber.matcher( s );

// Find each telephone number in the input text
while (m.find())
{
  String code = m.group(1);
  String num = m.group(2);
  System.out.println(num + " (in area code " + code + ")");
}
```

Note
`.group(0)` will always match the whole expression, and is equivalent to calling `.group()`

1.4.2.3 Iteratively replacing matches

Though the `replaceAll()` and `replaceFirst()` services (available both in `java.util.regex.Matcher` and `java.lang.String`) are the preferred, and most convenient, way of replacing matches, a Matcher offers a greater degree of control if required.

This is accomplished by

- Constructing or providing a new `java.lang.StringBuffer` which will iteratively be built up as the replaced sequences are appended
- for each `find()`, calling the `Matcher.appendReplacement(StringBuffer, String)` service, indicating the desired replacement for the matched sequence
- finally, calling `Matcher.appendTail(StringBuffer)` to append the rest of the input string (if any)

For example, we may wish to, for each found telephone number, append the name of the area code:

```
// Maps are codes to place names
Map<String, String> placeNames = new HashMap<String, String>();
placeNames.put("011", "Johannesburg");
placeNames.put("012", "Pretoria");
//...

String s = "I tried to call you on (011) 555-5555 but " +
    "failed to reach you. Should I have tried (012) 123-4567 " +
    "or (012) 321-7654 instead?";

// Matches telephone numbers in the form "(011) 555-5555'
```

```
Pattern telephoneNumber =
  Pattern.compile("\\(([0-9]{3})\\) ([0-9]{3}-[0-9]{4})");

// Create a matcher, which allows us to find, replace, etc
Matcher m = telephoneNumber.matcher( s );

// Find each telephone number in the input text, and
// replace and append to the output text
StringBuffer sb = new StringBuffer();
while (m.find())
{
  String code = m.group(1);
  String num = m.group(2);

  m.appendReplacement(sb, placeNames.get(code) + " " + num );
}
m.appendTail(sb);

System.out.println( sb );
```

to produce the output:

```
I tried to call you on Johannesburg 555-5555 but failed to reach you.
Should I have tried Pretoria 123-4567 or Pretoria 321-7654 instead?
```

1.5 Using regular expressions with files

The Java Regular Expressions framework does not include the functionality to directly work with information in domains such as the file system, network sockets, user interfaces, etc - this is because Java already contains the infrastructure to produce instances of character sequences.

A common requirement is to search for patterns within files. These files are often massive (such as log files) and a major benefit of Java (introduced at version 1.4) is the ability to use the NIO framework to memory-map a file to an object which can be directly exposed as a character sequence. In this scenario, the infrastructure (largely the operating system) takes full responsibility for managing the data access and memory required.

For example, the following program uses a regular expression to find all the names of the declared types (variables) in use in a Java source file:

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.channels.FileChannel.*;
import java.nio.charset.*;
import java.util.regex.*;

/** A small program to list all the variables (and their types)
 * by applying a regular expression pattern to a .java source
 * file. */
public class ListVarNames
{
  public static void main(String[] args) throws IOException
  {
    // Check program arguments
    if (args.length < 1)
    {
      System.err.println("Usage: <java source file>");
      return;
    }
    // Establish a NIO channel to the file
    FileChannel channel = new FileInputStream( args[0] ).getChannel();

    // Map it to memory
    ByteBuffer buf = channel.map( MapMode.READ_ONLY, 0, channel.size());

    // Decode it as UTF-8 text (the encoding used for .java files)
    CharBuffer text = Charset.forName("UTF-8").newDecoder().decode(buf);

    // Find all variable declarations (this is an overly-simplistic
    // pattern which may not be 100% accurate)
    String varPattern = "([\\w]+)\\s+([\\w]+)\\s*(;|=)";
    Matcher m = Pattern.compile( varPattern).matcher( text );
    while (m.find())
    {
      String varName = m.group(2);
      String type = m.group(1);
      System.out.println( varName + " (of type " + type + ")");
    }
  }
}
```

Which, if invoked on its own source code via:

```
java ListVarNames ../source/ListVarNames.java
```

produces the following output:

```
channel (of type FileChannel)
buf (of type ByteBuffer)
text (of type CharBuffer)
varPattern (of type String)
m (of type Matcher)
varName (of type String)
type (of type String)
```

Note
Using Java NIO and Memory-mapping will enable this small program to scale very well, even with the ability to read files larger than what could fit into memory.

1.6 Exercises

1.6.1 E-Mail address filter

Write a class which offers a service to validate e-mail addresses, and specifically, only positively validates addresses originating from a South African (.*.za) domain name.

Test your class by creating a small program that accepts e-mail addresses, and indicates whether they are valid or not.

1.6.2 Find TODO tags

It is common practice by developers to add snippets of text with a "TODO" tag to source files. For example:

```
public void foo()
{
  // TODO This service needs to be implemented!
}
```

Write a small program which will, given any UTF-8 text file (such as Java Source Code or XML), find all sentences which start with the term “TODO”, and display the outstanding work (indicated by the text following the TODO) to the user. Only the text on the same line (terminated by a end-of-line) should be considered, and use capturing groups to exclude the word “TODO” from the displayed text (to capture only the relevant text).

Chapter 2

Index