The Java™ Tutorials

Trail: Essential Classes
Lesson: Regular Expressions

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Methods of the Pattern Class

Until now, we've only used the test harness to create `Pattern` objects in their most basic form. This section explores advanced techniques such as creating patterns with flags and using embedded flag expressions. It also explores some additional useful methods that we haven't yet discussed.

Creating a Pattern with Flags

The `Pattern` class defines an alternate `compile` method that accepts a set of flags affecting the way the pattern is matched. The flags parameter is a bit mask that may include any of the following public static fields:

- `Pattern.CANON_EQ` Enables canonical equivalence. When this flag is specified, two characters will be considered to match if, and only if, their full canonical decompositions match. The expression `"a\u030A"`, for example, will match the string `"\u00E5"` when this flag is specified. By default, matching does not take canonical equivalence into account. Specifying this flag may impose a performance penalty.
- `Pattern.CASE_INSENSITIVE` Enables case-insensitive matching. By default, case-insensitive matching assumes that only characters in the US-ASCII charset are being matched. Unicode-aware case-insensitive matching can be enabled by specifying the UNICODE_CASE flag in conjunction with this flag. Case-insensitive matching can also be enabled via the embedded flag expression `(?i)`. Specifying this flag may impose a slight performance penalty.
- `Pattern.COMMENTS` Permits whitespace and comments in the pattern. In this mode, whitespace is ignored, and embedded comments starting with `#` are ignored until the end of a line. Comments mode can also be enabled via the embedded flag expression `(?x)`.
- `Pattern.DOTALL` Enables dotall mode. In dotall mode, the expression `.` matches any character, including a line terminator. By default this expression does not match line terminators. Dotall mode can also be enabled via the embedded flag expression `(?s)`. (The s is a mnemonic for "single-line" mode, which is what this is called in Perl.)
- `Pattern.LITERAL` Enables literal parsing of the pattern. When this flag is specified then the input string that specifies the pattern is treated as a sequence of literal characters. Metacharacters or escape sequences in the input sequence will be given no special meaning. The flags `CASE_INSENSITIVE` and `UNICODE_CASE` retain their impact on matching when used in conjunction with this flag. The other flags become superfluous. There is no embedded flag character for enabling literal parsing.
- `Pattern.MULTILINE` Enables multiline mode. In multiline mode the expressions `^` and `$` match just after or just before, respectively, a line terminator or the end of the input sequence. By default these expressions only match at the beginning and the end of the entire input sequence. Multiline mode can also be enabled via the embedded flag expression `(?m)`.
- `Pattern.UNICODE_CASE` Enables Unicode-aware case folding. When this flag is specified then case-insensitive matching, when enabled by the `CASE_INSENSITIVE` flag, is done in a manner consistent with the Unicode Standard. By default, case-insensitive matching assumes that only characters in the US-ASCII charset are being matched. Unicode-aware case folding can also be enabled via the embedded flag expression `(?u)`. Specifying this flag may impose a performance penalty.
- `Pattern.UNIX_LINES` Enables UNIX lines mode. In this mode, only the `'\n'` line terminator is recognized in the behavior of `.`, `^`, and `$`. UNIX lines mode can also be enabled via the embedded flag expression `(?d)`.

In the following steps we will modify the test harness, `RegexTestHarness.java` to create a pattern with case-insensitive matching.

First, modify the code to invoke the alternate version of `compile`:

```
Pattern pattern =
Pattern.compile(console.readLine("%nEnter your regex: "),
Pattern.CASE_INSENSITIVE);
```

Then compile and run the test harness to get the following results:

```
Enter your regex: dog
Enter input string to search: DoGDOg
I found the text "DoG" starting at index 0 and ending at index 3.
I found the text "DOg" starting at index 3 and ending at index 6.
```

As you can see, the string literal "dog" matches both occurences, regardless of case. To compile a pattern with multiple flags, separate the flags to be included using the bitwise OR operator "`|`". For clarity, the following code samples hardcode the regular expression instead of reading it from the `Console`:

```
pattern = Pattern.compile("[az]$", Pattern.MULTILINE | Pattern.UNIX_LINES);
```

You could also specify an `int` variable instead:

```
final int flags = Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE;
Pattern pattern = Pattern.compile("aa", flags);
```

Embedded Flag Expressions

It's also possible to enable various flags using *embedded flag expressions*. Embedded flag expressions are an alternative to the two-argument version of `compile`, and are specified in the regular expression itself. The following example uses the original test harness, `RegexTestHarness.java` with the embedded flag expression `(?i)` to enable case-insensitive matching.

```
Enter your regex: (?i)foo
Enter input string to search: FOOfooFoOfoO
I found the text "FOO" starting at index 0 and ending at index 3.
I found the text "foo" starting at index 3 and ending at index 6.
I found the text "FoO" starting at index 6 and ending at index 9.
I found the text "foO" starting at index 9 and ending at index 12.
```

Once again, all matches succeed regardless of case.

The embedded flag expressions that correspond to `Pattern`'s publicly accessible fields are presented in the following table:

Constant	Equivalent Embedded Flag Expression
`Pattern.CANON_EQ`	None
`Pattern.CASE_INSENSITIVE`	`(?i)`
`Pattern.COMMENTS`	`(?x)`
`Pattern.MULTILINE`	`(?m)`
`Pattern.DOTALL`	`(?s)`
`Pattern.LITERAL`	None
`Pattern.UNICODE_CASE`	`(?u)`
`Pattern.UNIX_LINES`	`(?d)`

Using the `matches(String,CharSequence)` Method

The `Pattern` class defines a convenient `matches` method that allows you to quickly check if a pattern is present in a given input string. As with all public static methods, you should invoke `matches` by its class name, such as `Pattern.matches("\\d","1");`. In this example, the method returns `true`, because the digit "1" matches the regular expression `\d`.

Using the `split(String)` Method

The `split` method is a great tool for gathering the text that lies on either side of the pattern that's been matched. As shown below in `SplitDemo.java`, the `split` method could extract the words "`one two three four five`" from the string "`one:two:three:four:five`":

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class SplitDemo {

    private static final String REGEX = ":";
    private static final String INPUT =
        "one:two:three:four:five";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        String[] items = p.split(INPUT);
        for(String s : items) {
            System.out.println(s);
        }
    }
}
```

```
OUTPUT:

one
two
three
```

```
four
five
```

For simplicity, we've matched a string literal, the colon (`:`) instead of a complex regular expression. Since we're still using `Pattern` and `Matcher` objects, you can use split to get the text that falls on either side of any regular expression. Here's the same example, `SplitDemo2.java`, modified to split on digits instead:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class SplitDemo2 {

    private static final String REGEX = "\\d";
    private static final String INPUT =
        "one9two4three7four1five";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        String[] items = p.split(INPUT);
        for(String s : items) {
            System.out.println(s);
        }
    }
}
```

```
OUTPUT:

one
two
three
four
five
```

Other Utility Methods

You may find the following methods to be of some use as well:

- `public static String quote(String s)` Returns a literal pattern `String` for the specified `String`. This method produces a `String` that can be used to create a `Pattern` that would match `String s` as if it were a literal pattern. Metacharacters or escape sequences in the input sequence will be given no special meaning.
- `public String toString()` Returns the `String` representation of this pattern. This is the regular expression from which this pattern was compiled.

Pattern Method Equivalents in `java.lang.String`

Regular expression support also exists in `java.lang.String` through several methods that mimic the behavior of `java.util.regex.Pattern`. For convenience, key excerpts from their API are presented below.

- `public boolean matches(String regex)`: Tells whether or not this string matches the given regular expression. An invocation of this method of the form *`str`*`.matches(`*`regex`*`)` yields exactly the same result as the expression `Pattern.matches(`*`regex, str`*`)`.
- `public String[] split(String regex, int limit)`: Splits this string around matches of the given regular expression. An invocation of this method of the form *`str`*`.split(`*`regex, n`*`)` yields the same result as the expression `Pattern.compile(`*`regex`*`).split(`*`str, n`*`)`
- `public String[] split(String regex)`: Splits this string around matches of the given regular expression. This method works the same as if you invoked the two-argument split method with the given expression and a limit argument of zero. Trailing empty strings are not included in the resulting array.

There is also a replace method, that replaces one `CharSequence` with another:

- `public String replace(CharSequence target,CharSequence replacement)`: Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence. The replacement proceeds from the beginning of the string to the end, for example, replacing "aa" with "b" in the string "aaa" will result in "ba" rather than "ab".

Previous page: Boundary Matchers
Next page: Methods of the Matcher Class