# Regular Expressions with Java

Week 14 – Presentation 4

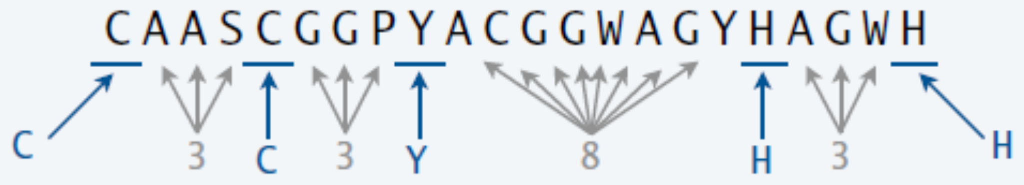
## REs in Java

Java's String class implements GREP.

| public class String | |
|---|---|
| ... | |
| boolean matches(String re) | *does this string match the given RE?* |
| ... | |

```
String re = "C.{2,4}C...[LIVMFYWC].{8}H.{3,5}H";
String zincFinger = "CAASCGGPYACGGAAGYHAGAH";
boolean test = zincFinger.matches(re);
```

true!



## Java RE client example: Validation

```
public class Validate
{
   public static void main(String[] args)
   {
      String re = args[0];
      while (!StdIn.isEmpty())
      {
         String input = StdIn.readString();
         StdOut.println(input.matches(re));
      }
   }
}
```

Does a given string match a given RE?

- Take RE from command line.
- Take strings from StdIn.

Applications

- Scientific research.
- Compilers and interpreters.
- Internet commerce.
- ...

need quotes to "escape" the shell

```
% java Validate "C.{2,4}C...[LIVMFYWC].{8}H.{3,5}H"
CAASCGGPYACGGAAGYHAGAH
true
CAASCGGPYACGGAAGYHGAH
false
```

$C_2H_2$ type zinc finger domain

```
% java Validate "[$_A-Za-z][$_A-Za-z0-9]*"
ident123
true
123ident
false
```

legal Java identifier

```
% java Validate "[a-z]+@([a-z]+\.)+(edu|com)"
wayne@cs.princeton.edu
true
eve@airport
false
```

valid email address (simplified)

## Beyond matching

Java's String class contains other useful RE-related methods.

- RE search and replace
- RE delimited parsing

| public class String | |
|---|---|
| ... | |
| String replaceAll(String re, String to) | *replace all occurrences of substrings matching RE with* to |
| String[] split(String re) | *split the string around matches of the given RE* |
| ... | |

Tricky notation (typical in string processing): \ signals "special character" so "\\\\" means "\\" and "\\\\s" means "\\s"

Examples using the RE "\\\\s+" (matches one or more whitespace characters).

Replace each sequence of at least one whitespace character with a single space.

```
String s = StdIn.readAll();
s = s.replaceAll("\\s+", " ");
```

Create an array of the words in StdIn (basis for StdIn.readAllStrings() method)

```
String s = StdIn.readAll();
String[] words = s.split("\\s+");
```

# Java String API
# for regular expressions' evaluation

| Method Signature | Purpose |
|---|---|
| boolean matches(String regex) | Matches the given regular expression against the string that the method is invoked on and returns true/false, indicating whether the match is successful (true) or not (false). |
| String `replaceAll`(String regex, String replacement) | Replaces each substring of the subject string that matches the given regular expression with the replacement string and returns the new string with the replaced content. |
| String `replaceFirst`(String regex, String replacement) | This method does the same as the previous one with the exception that it replaces only the first substring of the subject string that matches the given regular expression with the replacement string and returns the new string with the replaced content. |
| `String[]` split(String regex) | Splits the subject string using the given regular expression into an array of substrings (example given ahead). |
| `String[]` split(String regex, int limit) | This overloaded method does the same as the previous one but there is an additional second parameter. The `limit` parameter controls the number of times regular expressions are applied for splitting. |

## Way beyond matching

Java's Pattern and Matcher classes give fine control over the GREP implementation.

| public class Pattern | |
|---|---|
| ... | |
| `static Pattern compile(String re)` | *parse the re to construct a* Pattern |
| `Matcher matcher(String input)` | *create a* Matcher *that can find substrings matching the pattern in the given input string* |
| ... | |

Why not a constructor?
Good question.

| public class Matcher | |
|---|---|
| ... | |
| `boolean find()` | *set internal variable* match *to the next substring that matches the RE in the input. If none, return* false, *else return* true |
| `String group()` | *return* match |
| `String group(int k)` | *return the kth group (identified by parens within RE) in* match |
| ... | |

[A sophisticated interface designed for pros, but very useful for everyone.]

## Java pattern matcher client example: Harvester

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

public class Harvester
{
   public static void main(String[] args)
   {
      String re      = args[0];
      In in          = new In(args[1]);
      String input   = in.readAll();
      Pattern pattern = Pattern.compile(re);
      Matcher matcher = pattern.matcher(input);
      while (matcher.find())
         StdOut.println(matcher.group());

   }
}
```

Harvest information from input stream

- Take RE from command line.
- Take input from file or web page.
- Print all substrings matching RE.

```
% java Harvester "gcg(cgg|agg)*ctg" chromosomeX.txt
gcgcggcggcggcggcggctg
gcgctg
gcgctg
gcgcggcggcggaggcggaggcggctg

% java Harvester "[a-z]+@([a-z]+\.)+(edu|com)" http://www.cs.princeton.edu/people/faculty
...
rs@cs.princeton.edu
...
wayne@cs.princeton.edu
...
```

harvest patterns from DNA

harvest email addresses from web for spam campaign.
(no email addresses on that site any more)

## Using regular expressions in Java Scanner API

A scanner is a utility class used for parsing the input text and breaking the input into tokens of various types, such as Boolean, int, float, double, long, and so on. It generates tokens of various types using regular expression-based delimiters. The default delimiter is a whitespace. Using the Scanner API, we can generate tokens of all the primitive types in addition to string tokens.

The `String`, `Pattern`, and `Matcher` classes are able to parse the input and generate tokens of the `String` type only, but the `Scanner` class is very useful for checking and generating tokens of different types from the input source. The `Scanner` instance can be constructed using the `File`, `InputStream`, `Path`, `Readable`, `ReadableByteChannel`, and `String` arguments.

| **Method Signature** | **Purpose** |
|---|---|
| Scanner useDelimiter(String pattern) | Sets this scanner's delimiter regex pattern to a String regex argument. |
| Scanner<br><br>useDelimiter(Pattern pattern) | This method is almost the same as the previous one but gets a `Pattern` as an argument instead of a `String`. This means that we can pass a regular expression already compiled. If we are forced to use the version with the `String` argument, the scanner would compile the string to a `Pattern` object even if we have already executed that compilation in other parts of the code.<br><br>We will discuss the `Pattern` and `Matcher` class in the next chapter. |
| Pattern delimiter() | Returns the pattern being used by this scanner to match delimiters. |
| MatchResult match() | Returns the match result of the latest scan operation performed by this scanner. |

| **Method Signature** | **Purpose** |
|---|---|
| boolean hasNext(String pattern) | Returns `true` if the next token matches the pattern constructed from the specified string. |
| boolean hasNext(Pattern pattern) | This method is almost the same as the previous one but gets `Pattern` as an argument instead of `String`. |
| String next(String pattern) | Returns the next token if it matches the pattern constructed from the specified string. |
| String next(Pattern pattern) | This method is almost the same as the previous one but gets `Pattern` as an argument instead of `String`. |
| String findInLine(String pattern) | Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters. |

| **Method Signature** | **Purpose** |
|---|---|
| String findInLine(Pattern pattern) | This method is almost the same as the previous one but gets `Pattern` as an argument instead of `String`. |
| Scanner skip(String pattern) | Skips the input that matches a pattern constructed from the specified string, ignoring delimiters. |
| Scanner skip(Pattern pattern) | This method is almost the same as the previous one but gets `Pattern` as an argument instead of `String`. |
| String findWithinHorizon(String pattern, int horizon) | Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters. |
| String findWithinHorizon(Pattern pattern, int horizon) | This method is almost the same as the previous one but gets `Pattern` as an argument instead of `String`. |

## Package java.util.regex

## Java Regular Expression API - Pattern and Matcher Classes

- The `MatchResult` interface
- Using `Pattern` class
- Using `Matcher` class
- Various methods of `Pattern` and `Matcher` classes and how to use them for solving problems involving regular expressions

# The MatchResult interface

`MatchResult` is an interface for representing the result of a match operation. This interface is implemented by the `Matcher` class.

| Method Name | Description |
|---|---|
| int start() | Returns the start index of the match in the input |
| int start(int group) | Returns the start index of the specified capturing group |
| int end() | Returns the offset after the last character matched |
| int end(int group) | Returns the offset after the last character of the subsequence captured by the given group during this match |
| String group() | Returns the input substring matched by the previous match |
| String group(int group) | Returns the input subsequence captured by the given group during the previous match operation |
| int groupCount() | Returns the number of capturing groups in this match result's pattern |

Let's take an example to understand this interface better.

Suppose, the input string is a web server response line from HTTP response headers:

```
HTTP/1.1 302 Found
```

Our regex pattern to parse this line is as follows:

```
HTTP/1\.[01] (\d+) [a-zA-Z]+
```

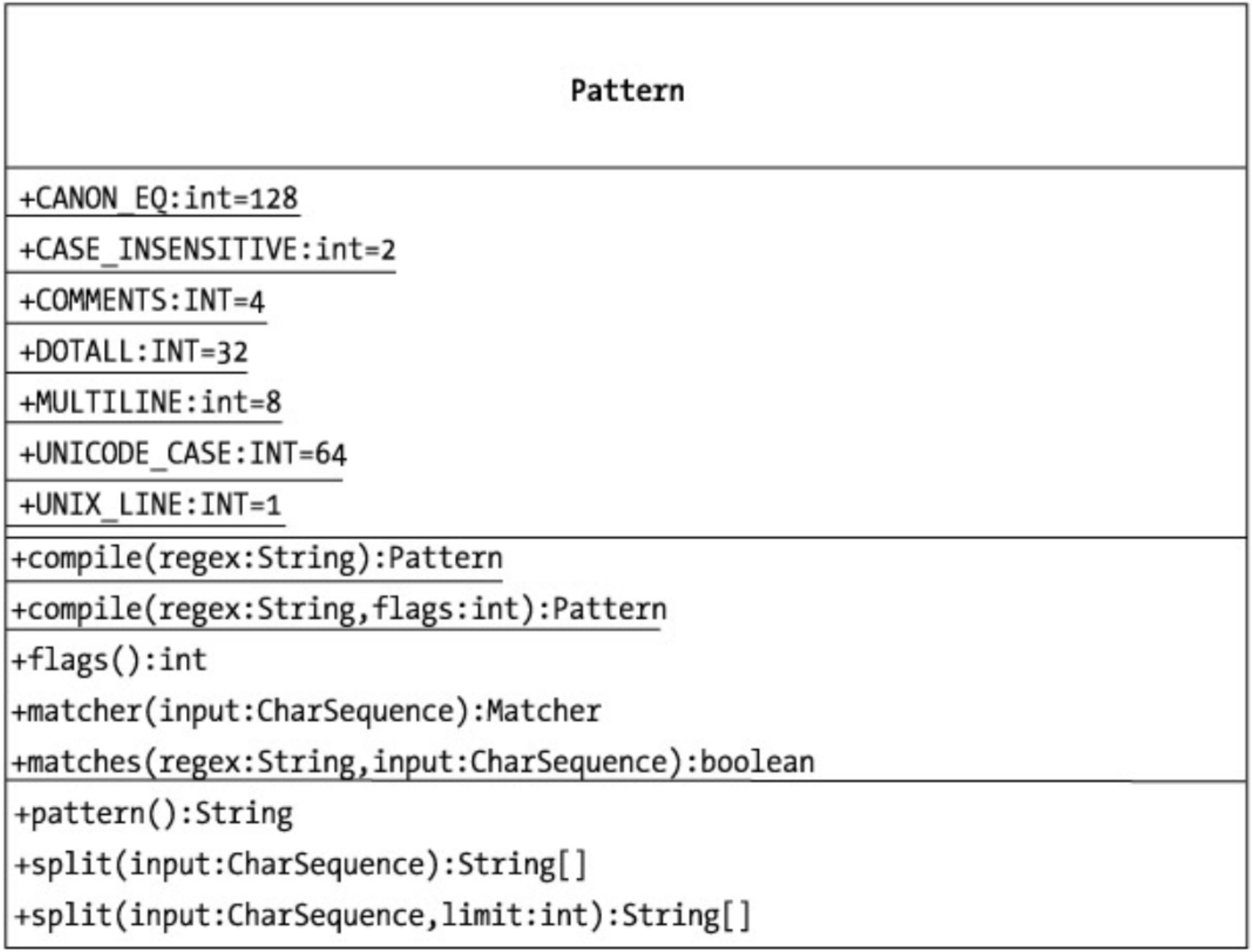
Note that there is only one captured group that captures integer status code.

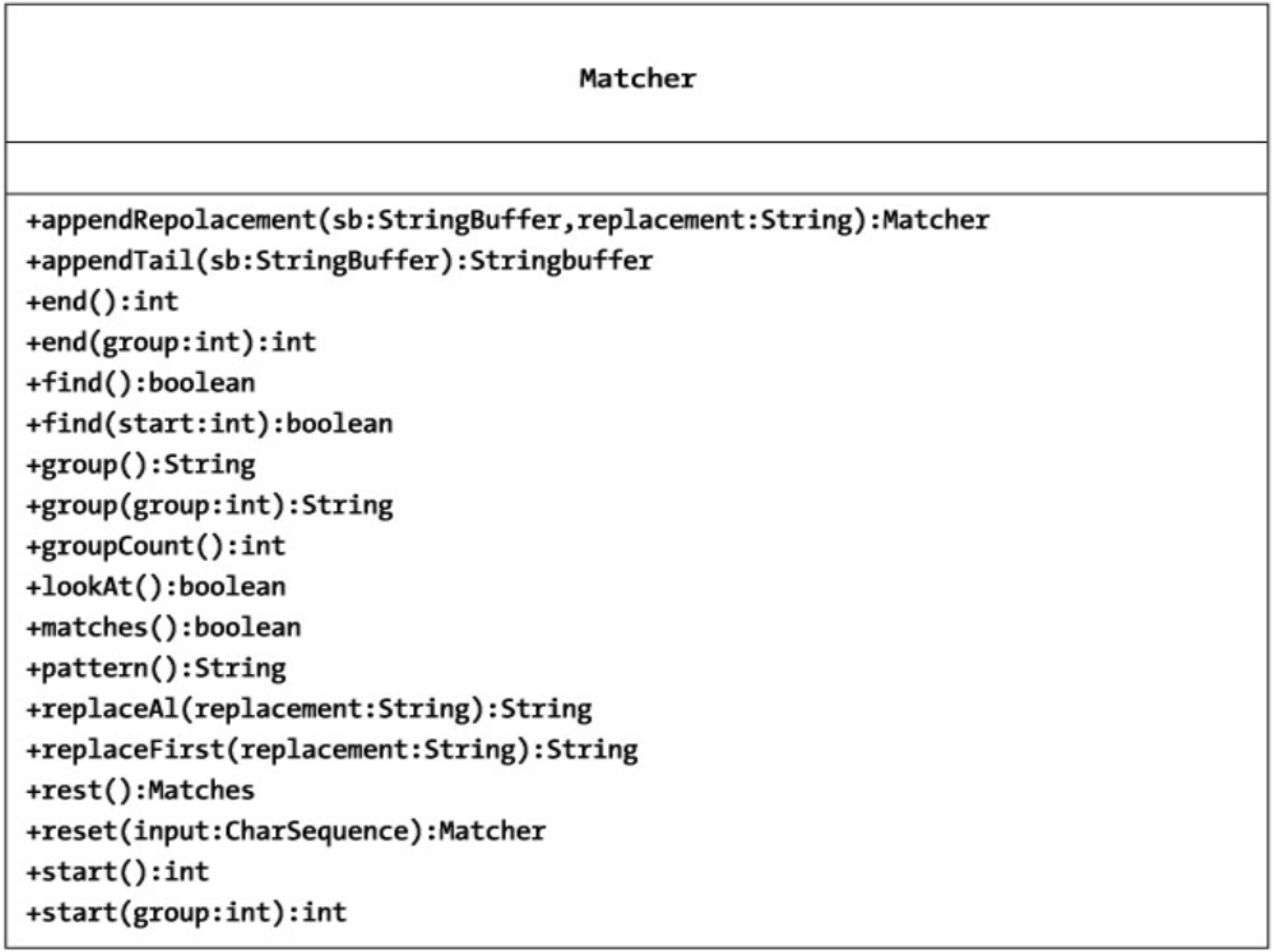
Let's look at this code listing to understand the various methods of the `MatchResult` interface better:

```
package example.regex;

import java.util.regex.*;

public class MatchResultExample
{
  public static void main(String[] args)
  {
    final String re = "HTTP/1\\.[01] (\\d+) [a-zA-Z]+";
    final String str = "HTTP/1.1 302 Found";

    final Pattern p = Pattern.compile(re);
    Matcher m = p.matcher(str);

    if (m.matches())
    {
      MatchResult mr = m.toMatchResult();

      // print count of capturing groups
      System.out.println("groupCount(): " + mr.groupCount());

      // print complete matched text
      System.out.println("group(): " + mr.group());

      // print start position of matched text
      System.out.println("start(): " + mr.start());

      // print end position of matched text
      System.out.println("end(): " + mr.end());

      // print 1st captured group
      System.out.println("group(1): " + mr.group(1));

      // print 1st captured group's start position
      System.out.println("start(1): " + mr.start(1));

      // print 1st captured group's end position
      System.out.println("end(1): " + mr.end(1));
    }
  }
}
```

We retrieve a `MatchResult` instance after calling the required `Pattern` and `Matcher` methods (discussed in the next section). After compiling and running the preceding code, we will get the following output, which shows the use of the various methods of this interface:

```
groupCount(): 1
group(): HTTP/1.1 302 Found
start(): 0
end(): 18
group(1): 302
start(1): 9
end(1): 12
```


Pattern
+CANON_EQ:int=128
+CASE_INSENSITIVE:int=2
+COMMENTS:INT=4
+DOTALL:INT=32
+MULTILINE:int=8
+UNICODE_CASE:INT=64
+UNIX_LINE:INT=1
+compile(regex:String):Pattern
+compile(regex:String,flags:int):Pattern
+flags():int
+matcher(input:CharSequence):Matcher
+matches(regex:String,input:CharSequence):boolean
+pattern():String
+split(input:CharSequence):String[]
+split(input:CharSequence,limit:int):String[]



```
Matcher

+appendRepolacement(sb:StringBuffer,replacement:String):Matcher
+appendTail(sb:StringBuffer):Stringbuffer
+end():int
+end(group:int):int
+find():boolean
+find(start:int):boolean
+group():String
+group(group:int):String
+groupCount():int
+lookAt():boolean
+matches():boolean
+pattern():String
+replaceAl(replacement:String):String
+replaceFirst(replacement:String):String
+rest():Matches
+reset(input:CharSequence):Matcher
+start():int
+start(group:int):int
```

## Convert Regex to Predicate

Use `Pattern.compile().asPredicate()` method to get a predicate from compiled regular expression.

This predicate can be used with lambda streams to apply it on each token into stream.