Java support for regular expressions

Java library classes for regular expressions

- `java.util.regex.Pattern`
- `java.util.regex.Matcher`
- `java.lang.String`

Example

```
import java.util.regex.*;
...
// simple use for a single match
assert "Java".matches("[A-Z][a-z]+");

// more efficient use for multiple matches
Pattern p = Pattern.compile("[A-Z][a-z]+"); // class factory method
Matcher m = p.matcher("Java");              // object factory method
assert m.matches();
```

Java support for regular expressions

`Pattern` class in a nutshell

- objects of `Pattern` are immutable and represent regular expressions
- patterns are created from strings by the class factory method
  `static Pattern compile(String regex)`
- patterns can create matchers with the object factory method
  `Matcher matcher(CharSequence input)`

Remarks

- `compile` may throw `PatternSyntaxException`
- `CharSequence` is an interface defined in `java.lang`
- `String` implements `CharSequence` $\Rightarrow$ `String` $\leq$ `CharSequence`

Java support for regular expressions

`Matcher` class in a nutshell

- a matcher is a mutable object with an input sequence and a pattern
- a matcher works on a subsequence of its input sequence, called region
- the bounds of the region can be modified with

```
Matcher region(int start, int end)
```

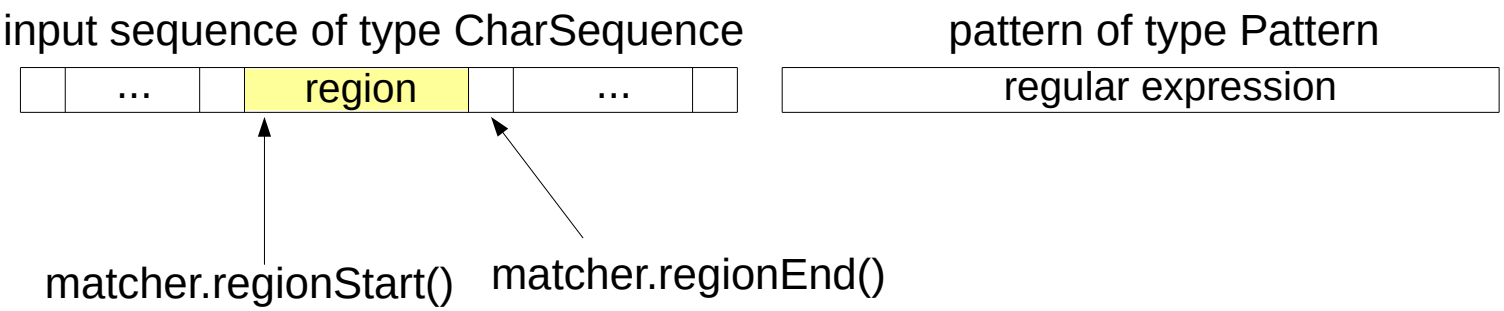
  remark: `start` included, `end` excluded

- the input sequence can be changed with

```
Matcher reset(CharSequence input)
```

  remark: initially the region is the entire input sequence

a matcher object



Java support for regular expressions

Match operations

- **boolean** `matches()`: tries to match the entire region against the pattern
- **boolean** `lookingAt()`: tries to match a subsequence of the region starting at its beginning
- **boolean** `find()`: tries to find the next subsequence of the input sequence that matches the pattern

Example

```
Pattern pt = Pattern.compile("[A-Z][a-z]+");
Matcher mt = pt.matcher("Java"); // region is the whole input
assert mt.matches();             // the entire region matches
mt.reset("Java language");       // region is the whole input
assert !mt.matches();            // the entire region does not match
assert mt.lookingAt();           // "Java" matches
mt.reset("language Java");       // region is the whole input
assert !mt.matches();            // the entire region does not match
assert !mt.lookingAt();          // no subsequence matches from the beginning
assert mt.find();                // "Java" matches
```

Java support for regular expressions

Query operations

- `int start()`: returns the start index of the previous match
- `int end()`: returns the index after the last character matched
- `String group()`: returns the string matched by the previous match

Example

```
Pattern pt = Pattern.compile("[A-Z][a-z]+");
Matcher mt = pt.matcher("Java Language");
assert !mt.matches();                    // the entire region does not match
assert mt.lookingAt();                   // "Java" matches
assert mt.start() == 0;
assert mt.end() == 4;
assert mt.group().equals("Java");
mt.region(mt.end(), mt.regionEnd()); // moves to " Language" and reset mt
assert !mt.matches();                    // the entire region does not match
assert !mt.lookingAt();                  // no subsequence matches from the beginning
assert mt.find();                        // "Language" matches
assert mt.start() == 5;
assert mt.end() == 13;
mt.group().equals("Language");
```

Java support for regular expressions

Remarks on query operations

A query throws `IllegalStateException` if any of the following requirements is verified:

- a match operation has not been called before the query
- the last match operation failed, that is, **`false`** was returned
- the matcher was reset after the last match operation

The following object methods reset the matcher:

- `Matcher reset(CharSequence input)`
- `Matcher region(int start,int end)`

Java support for regular expressions

Interface `java.util.regex.MatchResult`

- the result of the last match operation can be returned with the object method `MatchResult toMatchResult()`
- the result is unaffected by subsequent operations performed upon this matcher

Example

```
Pattern pt = Pattern.compile("[A-Z][a-z]+");
Matcher mt = pt.matcher("Java Language");
assert mt.lookingAt();
MatchResult prevMatch = mt.toMatchResult();
mt.region(mt.end(),mt.regionEnd()); // matcher is reset
assert prevMatch.start() == 0;      // result of the previous query
assert prevMatch.end() == 4;
assert prevMatch.group().equals("Java");
assert mt.start() == 0;             // throws IllegalStateException
```

Java support for regular expressions

Capturing groups

Parentheses force precedence but define also capturing groups

Example of groups

```
Pattern pt = Pattern.compile("(0|[1-9][0-9]*)([Ll]?)");
```

- group 0:

  `"(0|[1-9][0-9]*)([Ll]?)"`

- group 1:

  `"(0|[1-9][0-9]*)([Ll]?)"`

- group 2:

  `"(0|[1-9][0-9]*)([Ll]?)"`

Java support for regular expressions

Rules for capturing groups

- capturing groups are indexed from left to right, starting from 1
- each group starts with (and is a subexpression of the regular expression
- total number of groups = total number of open parentheses
- group 0 is the whole pattern, `mt.group(0)` equivalent to `mt.group()`
- definition of index of a group =
  the number of open parentheses from the beginning of the reg. exp.

Java support for regular expressions

Example 1

```
Pattern pt = Pattern.compile("(0|[1-9][0-9]*)([Ll]?)");
Matcher mt = pt.matcher("42L");
mt.lookingAt();
MatchResult res = mt.toMatchResult();
assert res.group(0).equals("42L");      // 0 is (0|[1-9][0-9]*)([Ll]?)
assert res.group(1).equals("42");       // 1 is (0|[1-9][0-9]*)
assert res.group(2).equals("L");        // 2 is ([Ll]?)
mt.reset("42");
mt.lookingAt();
res = mt.toMatchResult();
assert res.group(0).equals("42");
assert res.group(1).equals("42");
assert res.group(2).equals("");
```

Java support for regular expressions

Example 2

```
Pattern pt = Pattern.compile("([0-9]+)|([a-zA-Z]+)");
Matcher mt = pt.matcher("xy");
mt.lookingAt();
MatchResult res = mt.toMatchResult();
assert res.group(0).equals("xy");       // 0 is ([0-9]+)|([a-zA-Z]+)
assert res.group(1) == null;            // 1 is ([0-9]+)
assert res.group(2).equals("xy");       // 2 is ([a-zA-Z]+)
mt.reset("42");
mt.lookingAt();
res = mt.toMatchResult();
assert res.group(0).equals("42");
assert res.group(1).equals("42");
assert res.group(2) == null;
```

Java support for regular expressions

A selection of regular-expression constructs in Java

- Logical operators
  - `XY` `X` followed by `Y` (concatenation)
  - `X|Y` Either `X` or `Y` (union)
  - `(X)` `X`, as a capturing group (anyway parentheses force precedence)
- Postfix operators (called greedy quantifiers)
  - `X?` `X`, once or not at all (optionality)
  - `X*` `X`, zero or more times (Kleene star)
  - `X+` `X`, one or more times (Kleene star except the empty string)
- Characters
  - `x` The character `x` (if it is not a special character)
  - `\` Nothing, but quotes the following character
    example: `\\` is the backslash character
  - `\t` The tab character
  - `\n` The newline (line feed) character
  - `\r` The carriage-return character

Java support for regular expressions

A selection of regular-expression constructs in Java

- Character classes
  - `[abc]` `a`, `b`, or `c` (simple class)
  - `[^abc]` Any character except `a`, `b`, or `c` (negation)
  - `[a-zA-Z]` `a` through `z` or `A` through `Z`, inclusive (range)
- Predefined character classes
  - `.` Any character (except line terminators, unless the `DOTALL` flag is specified)
  - `\s` A whitespace character
- Boundary matchers
  - `^` The beginning of a line
  - `$` The end of a line
  - `\b` A word boundary
- Non-capturing parentheses
  - `(?:X)` `X`, as a non-capturing group
- See the full documentation in the API documentation

Java support for regular expressions

Useful constructs used in the labs

- `(?:X)` (non-capturing group):
  useful when parentheses are needed with no group
  example: a reg. exp. where only group 0 is defined

```
Pattern pt = Pattern.compile("(?:0|[1-9][0-9]*)[Ll]");
```

- `\b` (word boundary): useful to define keywords

Remarks on regular expressions and strings

Be careful when using white spaces and special characters in strings!

Example

```
assert " ".matches(" | ");
assert "|".matches("\\|"); // \| -> \\|
assert "|".matches("[|]");
```

Java support for regular expressions

Greedy versus non-greedy operators

`?`, `+` and `*` try to match the longest string, concatenation and `|` do not

Example

```
Pattern pt = Pattern.compile("\\d*"); // recall "\\d" is "[0-9]"
mt = pt.matcher("234");
assert mt.lookingAt();
assert mt.group().equals("234"); // longest string matched

pt = Pattern.compile("\\d\\d|\\d\\d\\d");
mt = pt.matcher("234");
assert mt.lookingAt();
assert mt.group().equals("23"); // longest string "234" not matched

pt = Pattern.compile("(\\d\\d)?(\\d\\d\\d)?");
mt = pt.matcher("234");
assert mt.lookingAt();
assert mt.group().equals("23"); // longest string "234" not matched
```

Java support for regular expressions

Problems with keywords and identifiers in lexers

Because | is not greedy, it is not straightforward to define a correct regular expression able to distinguish keywords and identifiers

Example: problem with keywords and identifiers in lexers

Solution 1: not correct, keyword `"if"` matched instead of identifier `"ifvar"`

```
pt = Pattern.compile("(if|let)|([a-zA-Z]\\w*)");// recall "\\w" is "[a-zA-Z_0-9]"
mt = pt.matcher("ifvar");
assert mt.lookingAt();
assert mt.group(1).equals("if");
```

Solution 2: not correct, keyword `"if"` matched as an identifier

```
pt = Pattern.compile("([a-zA-Z]\\w*)|(if|let)");
mt = pt.matcher("if");
assert mt.lookingAt();
assert mt.group(1).equals("if");
```

Java support for regular expressions

Problems with keywords and identifiers in lexers

Because | is not greedy, it is not straightforward to define a correct regular expression able to distinguish keywords and identifiers

Example: problem with keywords and identifiers in lexers

Correct solution with word boundary `"\\b"`

```
pt = Pattern.compile("((?:if|let)\\b)|([a-zA-Z]\\w*)");
mt = pt.matcher("ifvar");
assert mt.lookingAt();
assert mt.group(2).equals("ifvar"); // identifier
mt.reset("if");
assert mt.lookingAt();
assert mt.group(1).equals("if"); // keyword
```

`"\\b"` means:

a non word char (i.e. not in `"\\w"`) must follow

Subtyping and array types

Questions

```
int i=42;
long l=i; // ok, int ≤ long
int[] ia={1,2,3};
long[] la=ia; // allowed?

String s="hello";
Object o=s; // ok, String ≤ Object
String[] sa={"one","two","three"};
Object[] oa=sa; // allowed?
```

Subtyping and array types

Answers

```
int i=42;
long l=i; // ok, int ≤ long
int[] ia={1,2,3};
long[] la=ia; // not allowed

String s="hello";
Object o=s; // ok, String ≤ Object
String[] sa={"one","two","three"};
Object[] oa=sa; // allowed
```

Subtyping and array types

Java rules

- T_1 and T_2 reference types: $T_1 \leq T_2 \Rightarrow T_1[\,] \leq T_2[\,] \leq$ `Object`
- T primitive type: $T[\,] \leq$ `Object`
- T primitive type: the only array type compatible with $T[\,]$ is $T[\,]$

Examples

- `String[]` $\leq$ `Object[]` $\leq$ `Object`
- `Integer[]` $\leq$ `Number[]` $\leq$ `Object`
- `Integer[]` $\not\leq$ `Long[]`
- **`int`**`[]` $\leq$ `Object`
- **`int`**`[]` $\not\leq$ `Integer[]`
- **`int`**`[]` $\not\leq$ `Object[]`
- **`int`**`[]` $\not\leq$ **`long`**`[]`
- the only array type compatible with **`int`**`[]` is itself

Subtyping and array types

Java rules

- T_1 and T_2 reference types: $T_1 \leq T_2 \Rightarrow T_1[\,] \leq T_2[\,] \leq \texttt{Object}$
- T primitive type: $T[\,] \leq \texttt{Object}$
- T primitive type: the only array type compatible with $T[\,]$ is $T[\,]$

Example

```
public static int sum (Integer[] ints) {
    int s = 0;
    for (int n : ints) { s += n; }
    return s;
}
public static void main(String[] args) {
    sum(new int[]{1,2,3,4}); // compile-time error: int[] ⊈ Integer[]
}
```

Subtyping and array types

Remark

- array subtyping rules are not sound in Java
- sound means: if rules hold, then there will be no type errors at runtime
- consequence of not sound subtyping:
  array assignment requires a dynamic type check

Example with code that correctly compiles

```
Object[] oa;
String[] sa = {"a", "b"};
oa = sa;                    // ok: String ≤ Object ⇒ String[] ≤ Object[]
oa[0] = new Circle(42); // ok Circle ≤ Object
assert sa[0].length() == 1;
```

Subtyping and array types

Remark

- array subtyping rules are not sound in Java
- sound means: if rules hold, then there will be no type errors at runtime
- consequence of not sound subtyping:
  array assignment requires a dynamic type check

Execution of the code

```
Object[] oa;
String[] sa = {"a", "b"};
oa = sa;
oa[0] = new Circle(42);      // throws ArrayStoreException at runtime
assert sa[0].length() == 1; // never executed
```