14.7 Regular Expressions, Class `Pattern` and Class `Matcher`

A **regular expression** is a `String` that describes a *search pattern* for *matching* characters in other `String`s. Such expressions are useful for *validating input* and ensuring that data is in a particular format. For example, a ZIP code must consist of five digits, and a last name must contain only letters, spaces, apostrophes and hyphens. One application of regular expressions is to facilitate the construction of a compiler. Often, a large and complex regular expression is used to *validate the syntax of a program*. If the program code does *not* match the regular expression, the compiler knows that there's a syntax error in the code.

Class `String` provides several methods for performing regular-expression operations, the simplest of which is the matching operation. `String` method `matches` receives a `String` that specifies the regular expression and matches the contents of the `String` object on which it's called to the regular expression. The method returns a `boolean` indicating whether the match succeeded.

A regular expression consists of literal characters and special symbols. Figure 14.19 specifies some **predefined character**

classes that can be used with regular expressions. A character class is an *escape sequence* that represents a group of characters. A digit is any numeric character. A **word character** is any letter (uppercase or lowercase), any digit or the underscore character. A white-space character is a space, a tab, a carriage return, a newline or a form feed. Each character class matches a single character in the `String` we're attempting to match with the regular expression.

Character	Matches	Character	Matches
`\d`	any digit	`\D`	any nondigit
`\w`	any word character	`\W`	any nonword character
`\s`	any white-space character	`\S`	any non-whitespace character

Fig. 14.19

Predefined character classes.

Regular expressions are not limited to these predefined character classes. The expressions employ various operators and other forms of notation to match complex patterns.

We examine several of these techniques in the application in Figs. 14.20 and 14.21, which *validates user input* via regular expressions. [*Note:* This application is not designed to match all possible valid user input.]

```java
// Fig. 14.20: ValidateInput.java
// Validating user information using regular expr

public class ValidateInput {
   // validate first name
   public static boolean validateFirstName(String
      return firstName.matches("[A-Z][a-zA-Z]*");
   }

   // validate last name
   public static boolean validateLastName(String
      return lastName.matches("[a-zA-z]+(['-][a-z
   }

   // validate address
   public static boolean validateAddress(String a
      return address.matches(
         "\\d+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]
   }

   // validate city
   public static boolean validateCity(String city
      return city.matches("([a-zA-Z]+|[a-zA-Z]+\\
   }

   // validate state
   public static boolean validateState(String sta
      return state.matches("([a-zA-Z]+|[a-zA-Z]+\
   }

   // validate zip
   public static boolean validateZip(String zip)
      return zip.matches("\\d{5
   }

   // validate phone
   public static boolean validatePhone(String pho
      return phone.matches("[1-9]\\d{2}-[1-9]\\d{
   }
}
```

Fig. 14.20

Validating user information using regular expressions.

```
// Fig. 14.21: Validate.java
// Input and validate data from user using the Va
import java.util.Scanner;

public class Validate {
   public static void main(String[] args) {
      // get user input
      Scanner scanner = new Scanner(System.in);
      System.out.println("Please enter first name
      String firstName = scanner.nextLine();
      System.out.println("Please enter last name:
      String lastName = scanner.nextLine();
      System.out.println("Please enter address:")
      String address = scanner.nextLine();
      System.out.println("Please enter city:");
      String city = scanner.nextLine();
      System.out.println("Please enter state:");
      String state = scanner.nextLine();
      System.out.println("Please enter zip:");
      String zip = scanner.nextLine();
      System.out.println("Please enter phone:");
      String phone = scanner.nextLine();

      // validate user input and display error me
      System.out.printf("%nValidate Result:");

      if (!ValidateInput.validateFirstName(firstN
         System.out.println("Invalid first name")
      }
      else if (!ValidateInput.validateLastName(la
            System.out.println("Invalid last name");
        }
        else if (!ValidateInput.validateAddress(add
            System.out.println("Invalid address");
        }
        else if (!ValidateInput.validateCity(city))
            System.out.println("Invalid city");
        }
        else if (!ValidateInput.validateState(state
            System.out.println("Invalid state");
        }
        else if (!ValidateInput.validateZip(zip)) {
            System.out.println("Invalid zip code");
        }
        else if (!ValidateInput.validatePhone(phone
            System.out.println("Invalid phone number
        }
        else {
            System.out.println("Valid input.  Thank
        }
    }
}
```

```
Please enter first name:
Jane
Please enter last name:
Doe
Please enter address:
123 Some Street
Please enter city:
Some City
Please enter state:
SS
Please enter zip:
123
Please enter phone:
123-456-7890
```

```
Validate Result:
Invalid zip code
```

```
Please enter first name:
Jane
Please enter last name:
Doe
Please enter address:
123 Some Street
Please enter city:
Some City
Please enter state:
SS
Please enter zip:
12345
Please enter phone:
123-456-7890

Validate Result:
Valid input.  Thank you.
```

Fig. 14.21

Input and validate data from user using the `ValidateInput` class.

Figure 14.20 validates user input. Line 7 validates the first name. To match a set of characters that does not have a predefined character class, use square brackets, `[]`. For example, the pattern `"[aeiou]"` matches a single character that's a vowel. Character ranges are represented by placing a dash (`-`) between two characters. In the example, `"[A-Z]"` matches a single uppercase letter. If the first character in the brackets is `"^"`, the expression accepts any character other than those indicated. However, `"[^Z]"` is not the same as `"[AY]"`, which matches uppercase letters A–Y—`"[^Z]"` matches *any character other than* capital Z, including lowercase letters and nonletters such as the newline character. Ranges in character classes are determined by the letters' integer values. In this example, `"[A-Za-z]"` matches all uppercase and lowercase letters. The range `"[A-z]"` matches all letters and also matches those characters (such as `[` and `\`) with an integer value between uppercase `Z` and lowercase `a` (for more information on integer values of characters see Appendix B). Like predefined character classes, character classes delimited by square brackets match a single character in the search object.

In line 7, the asterisk after the second character class indicates that any number of letters can be matched. In general, when the regular-expression operator `"*"` appears in a regular expression, the application attempts to match zero or more occurrences of the subexpression immediately preceding the `"*"`. Operator `"+"` attempts to match one or more occurrences of the subexpression immediately preceding `"+"`. So both `"A*"` and `"A+"` will match `"AAA"` or `"A"`, but only `"A*"` will match an empty string.

If method `validateFirstName` returns `true` (line 27 of Fig. 14.21), the application attempts to validate the last name (line 30) by calling `validateLastName` (lines 11–13 of

Fig. 14.20). The regular expression to validate the last name matches any number of letters split by apostrophes or hyphens.

Line 33 of Fig. 14.21 calls method `validateAddress` (lines 16–19 of Fig. 14.20) to validate the address. The first character class matches any digit one or more times (`\\d+`). Two `\` characters are used, because `\` normally starts an escape sequence in a string. So `\\d` in a `String` represents the regular-expression pattern `\d`. Then we match one or more white-space characters (`\\s+`). The character `"|"` matches the expression to its left or to its right. For example, "`Hi (John|Jane)"` matches both "`Hi John"` and "`Hi Jane"`. The parentheses are used to group parts of the regular expression. In this example, the left side of `|` matches a single word, and the right side matches two words separated by any amount of white space. So the address must contain a number followed by one or two words. Therefore, `"10 Broadway"` and `"10 Main Street"` are both valid addresses in this example. The city (lines 22–24 of Fig. 14.20) and state (lines 27–29 of Fig. 14.20) methods also match any word of at least one character or, alternatively, any two words of at least one character if the words are separated by a single space, so both `Waltham` and `West Newton` would match.

Quantifiers

The asterisk (`*`) and plus (`+`) are formally called **quantifiers**. Figure 14.22 lists all the quantifiers. We've already discussed how the asterisk (`*`) and plus (`+`) quantifiers work. All quantifiers affect only the subexpression immediately preceding the quantifier. Quantifier question mark (`?`) matches zero or one occurrences of the expression that it quantifies. A set of braces containing one number (`{`*n*`}`) matches exactly *n* occurrences of the expression it quantifies. We demonstrate this quantifier to validate the zip code in Fig. 14.20 at line 33. Including a comma after the number enclosed in braces matches at least *n* occurrences of the quantified expression. The set of braces containing two numbers (`{`*n*`,`*m*`}`) matches between *n* and *m* occurrences of the expression that it qualifies. Quantifiers may be applied to patterns enclosed in parentheses to create more complex regular expressions.

Quantifier	**Matches**
`*`	Matches zero or more occurrences of the pattern.
`+`	Matches one or more occurrences of the pattern.
`?`	Matches zero or one occurrences of the pattern.
`{`*n*`}`	Matches exactly *n* occurrences.
`{`*n*`,}`	Matches at least *n* occurrences.
`{`*n*`,`*m*`}`	Matches between *n* and *m* (inclusive) occurrences.

Fig. 14.22

Quantifiers used in regular expressions.

All of the quantifiers are **greedy**. This means that they’ll match as many occurrences as they can as long as the match is still successful. However, if any of these quantifiers is followed by a question mark (`?`), the quantifier becomes **reluctant** (sometimes called **lazy**). It then will match as few occurrences as possible as long as the match is still successful.

The zip code (line 33 in Fig. 14.20) matches a digit five times. This regular expression uses the digit character class and a quantifier with the digit 5 between braces. The phone number (line 38 in Fig. 14.20) matches three digits (the first one cannot be zero) followed by a dash followed by three more digits (again the first one cannot be zero) followed by four more digits.

`String` method `matches` checks whether an entire `String` conforms to a regular expression. For example, we want to accept `"Smith"` as a last name, but not `"9@Smith#"`. If only a substring matches the regular expression, method `matches` returns `false`.

14.7.1 Replacing Substrings and Splitting Strings

Sometimes it’s useful to replace parts of a string or to split a string into pieces. For this purpose, class `String` provides methods `replaceAll`, `replaceFirst` and `split`. These methods are demonstrated in Fig. 14.23.

```
// Fig. 14.23: RegexSubstitution.java
// String methods replaceFirst, replaceAll and sp
import java.util.Arrays;

public class RegexSubstitution {
   public static void main(String[] args) {
      String firstString = "This sentence ends in
      String secondString = "1, 2, 3, 4, 5, 6, 7,

      System.out.printf("Original String 1: %s%n"

      // replace '*' with '^'
      firstString = firstString.replaceAll("\\*",

      System.out.printf("^ substituted for *: %s%

      // replace 'stars' with 'carets'
      firstString = firstString.replaceAll("stars

      System.out.printf(
         "\"carets\" substituted for \"stars\": %

      // replace words with 'word'
      System.out.printf("Every word replaced by \
         firstString.replaceAll("\\w+", "word"));

      System.out.printf("Original String 2: %s%n"

```

```
      // replace first three digits with 'digit'
      for (int i = 0; i < 3; i++) {
         secondString = secondString.replaceFirst
      }

      System.out.printf(
         "First 3 digits replaced by \"digit\" :

```

```
37         System.out.print("String split at commas: "
38         String[] results = secondString.split(",\\s
39         System.out.println(Arrays.toString(results)
                     40     }
                       41 }
```

```
Original String 1: This sentence ends in 5 stars ****
^ substituted for *: This sentence ends in 5 stars ^^
"carets" substituted for "stars": This sentence ends
Every word replaced by "word": word word word word wo

    Original String 2: 1, 2, 3, 4, 5, 6, 7, 8
First 3 digits replaced by "digit" : digit, digit, di
String split at commas: [digit, digit, digit, 4, 5, 6
```

Fig. 14.23

`String` methods `replaceFirst`, `replaceAll` and `split`.

Method `replaceAll` replaces text in a `String` with new text (the second argument) wherever the original `String` matches a regular expression (the first argument). Line 13 replaces every instance of `"*"` in `firstString` with `"^"`. The regular expression (`"\\*"`) precedes character `*` with two backslashes. Normally, `*` is a quantifier indicating that a regular expression should match *any number of occurrences* of a preceding pattern. However, in line 13, we want to find all occurrences of the literal character `*`—to do this, we must escape character `*` with character `\`. Escaping a special regular-expression character with `\` instructs the matching engine to find the actual character. Since the expression is stored in a Java `String` and `\` is a special character in Java `String`s, we must include an additional `\`. So the Java `String "\\*"` represents the regular-expression pattern `\*` which matches a single `*` character in the search string. In line 18, every match for the regular expression `"stars"` in `firstString` is replaced with `"carets"`. Line 25 uses `replaceAll` to replace all words in the string with `"word"`.

Method `replaceFirst` (line 31) replaces the first occurrence of a pattern match. Java `String`s are immutable; therefore, method `replaceFirst` returns a new `String` in which the appropriate characters have been replaced. This line takes the original `String` and replaces it with the `String` returned by `replaceFirst`. By iterating three times we replace the first three instances of a digit (`\d`) in `secondString` with the text `"digit"`.

Method `split` divides a `String` into several substrings. The original is broken in any location that matches a specified regular expression. Method `split` returns an array of `String`s containing the substrings between matches for the regular expression. In line 38, we use method `split` to tokenize a `String` of comma-separated integers. The argument is the regular expression that locates the delimiter. In this case, we use the regular expression `",\s*"` to separate the substrings wherever a comma occurs—again, the Java

`String ",\\s*"` represents the regular expression `,\s*`. By matching any white-space characters, we eliminate extra spaces from the resulting substrings. The commas and white-space characters are not returned as part of the substrings. Line 39 uses `Arrays` method `toString` to display the contents of array `results` in square brackets and separated by commas.

14.7.2 Classes `Pattern` and `Matcher`

In addition to the regular-expression capabilities of class `String`, Java provides other classes in package `java.util.regex` that help developers manipulate regular expressions. Class `Pattern` represents a regular expression. Class `Matcher` contains both a regular-expression pattern and a `CharSequence` in which to search for the pattern.

`CharSequence` (package `java.lang`) is an *interface* that allows read access to a sequence of characters. The interface requires that the methods `charAt`, `length`, `subSequence` and `toString` be declared. Both `String` and `StringBuilder` implement interface `CharSequence`, so an instance of either of these classes can be used with class `Matcher`.

Common Programming Error 14.2

A regular expression can be tested against an object of any class that implements interface `CharSequence`*, but the regular expression must be a* `String`*. Attempting to create a regular expression as a* `StringBuilder` *is an error.*

If a regular expression will be used only once, `static Pattern` method `matches` can be used. This method takes a `String` that specifies the regular expression and a `CharSequence` on which to perform the match. This method returns a `boolean` indicating whether the search object (the second argument) *matches* the regular expression.

If a regular expression will be used more than once (in a loop, for example), it's more efficient to use `static Pattern` method `compile` to create a specific `Pattern` object for that regular expression. This method receives a `String` representing the regular expression and returns a new `Pattern` object, which can then be used to call method `matcher`. This method receives a `CharSequence` to search and returns a `Matcher` object.

`Matcher` provides method `matches`, which performs the same task as `Pattern` method `matches`, but receives no arguments—the search pattern and search object are encapsulated in the `Matcher` object. Class `Matcher` provides other methods, including `find`, `lookingAt`, `replaceFirst` and `replaceAll`.

Figure 14.24 presents a simple example that employs regular expressions. This program matches birthdays against a regular expression. The expression matches only birthdays that do not occur in April and that belong to people whose names begin with `"J"`.

```
// Fig. 14.24: RegexMatches.java
// Classes Pattern and Matcher.
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {
   public static void main(String[] args) {
      // create regular expression
      Pattern expression =
         Pattern.compile("J.*\\d[0-35-9]-\\d\\d-\

      String string1 = "Jane's Birthday is 05-
         "Dave's Birthday is 11-04-68\n" +
         "John's Birthday is 04-28-73\n" +
         "Joe's Birthday is 12-17-77";

      // match regular expression to string an
      Matcher matcher = expression.matcher(str

      while (matcher.find()) {
         System.out.println(matcher.group());
      }
   }
}
```

```
Jane's Birthday is 05-12-75
Joe's Birthday is 12-17-77
```

Fig. 14.24

`Classes Pattern` and `Matcher`.

Lines 9–10 create a `Pattern` by invoking its `static` method `compile`. The dot character `"."` in the regular expression (line 10) matches any single character except a newline. Line 18 creates the `Matcher` object for the compiled regular expression and the matching sequence (`string1`). Lines 20–22 use a `while` loop to *iterate* through the `String`. `Matcher` method `find` (line 20) attempts to match a piece of the search object to the search pattern. Each call to `find` starts at the point where the last call ended, so multiple matches can be found. `Matcher` method `lookingAt` performs the same way, except that it always starts from the beginning of the search object and will always find the *first* match if there is one.

Common Programming Error 14.3

Method `matches` *(from class* `String`*,* `Pattern` *or* `Matcher`*) will return* `true` *only if the entire search object matches the regular expression. Methods* `find` *and* `lookingAt` *(from class* `Matcher`*) will return* `true` *if a portion of the search object matches the regular expression.*

Line 21 uses `Matcher` method `group`, which returns the `String` from the search object that matches the search pattern. The `String` that's returned is the one that was last matched by a call to `find` or `lookingAt`. The output in Fig. 14.24 shows the two matches that were found in `string1`.

Java SE 8

8

As you'll see in Section 17.13, you can combine regular-expression processing with Java SE 8 lambdas and streams to implement powerful `String`- and file processing applications.

Java SE 9: New `Matcher` Methods

9

Java SE 9 adds several new `Matcher` method overloads—`appendReplacement`, `appendTail`, `replaceAll`, `results` and `replaceFirst`. Methods `appendReplacement` and `appendTail` simply receive

`StringBuilder`s rather than `StringBuffer`s. Methods `replaceAll`, `results` and `replaceFirst` are meant for use with lambdas and streams. We’ll show these three methods in Chapter 17.