5 String processing

Exam objectives covered in this chapter	What you need to know
[5.1] Search, parse, and build strings (including `Scanner`, `StringTokenizer`, `StringBuilder`, `String`, and `Formatter`)	The methods that can be used to parse and build strings, from classes `Scanner`, `StringTokenizer`, `StringBuilder`, and `Formatter`.
[5.2] Search, parse, and replace strings by using regular expressions, using expression patterns for matching limited to . (dot), * (star), + (plus), ?, \d, \D, \s, \S, \w, \W, \b, \B, [], ()	What regular expressions (regex) are and how they're used to search, parse, and replace strings. Understand the use of the relevant API classes in the `java.util.regex` package.
[5.3] Format strings using the formatting parameters %b, %c, %d, %f, and %s in format strings	The purpose of formatting parameters to format strings and other data types. How to determine code output when invalid combination of format parameters and data types is used.

Imagine that after completing a 500-page draft of your novel, you want to change the name of your main character from *Beri* to *Bery*. This should be simple, right? You can use your word processor's Find/Replace option and be well on your way. From a text file, you pull out a list of email addresses of all the publishers to whom you wish to submit your manuscript. But before using these addresses, you want to run a quick check to ensure that they're valid—that they include an @ sign and a dot (.), followed by a domain name. This should also be simple. You can use your email client to check them. Now, imagine your novel becomes a best seller, and

your publisher wants to translate it into multiple languages. Wow! Assuming that your novel includes numbers and decimal numbers, the publisher would also need to reformat these numbers based on the language that your text is translated into. Various languages might use different separators in decimal numbers. Though not a straightforward task, it's feasible.

Finding and replacing text, validating it, and formatting numbers in different ways are all examples of common requirements. Java applications might need to perform similar common data manipulation and formatting tasks. To accomplish this, Java includes flexible and powerful classes and methods to search, parse, replace, and format data. This chapter covers

- How classes from the Java API (`String`, `StringBuilder`, `Scanner`, `StringTokenizer`, `Formatter`) can help you search, parse, build, and replace strings
- Regular expressions and what you can do with them
- How to format strings by using format specifiers

Even though you might be familiar with working with the Java programming language, it's possible that you didn't get an opportunity to work with regular expressions (regex) in Java (or in any other programming language). Now available to be used with most programming languages, either integrated or as an external library, regex is a powerful and flexible language to describe data and search matching data. But this chapter's coverage of regex is limited to the exam topics.

Let's start with the basic differences between searching text for exact matches and searching for regex patterns.

5.1 Regular expressions

[5.2] Search, parse, and replace strings by using regular expressions, using expression patterns for matching limited to . (dot), * (star), + (plus), ?, \d, \D, \s, \S, \w, \W, \b, \B, [], ().

As opposed to exact matches, you can use regex to search for data that matches a *pattern*. Let's imagine that in addition to changing the name of your main character in your novel, you want to change all references of *sun* to *moon*. Though the solution might seem as simple as using Find/Replace, how would you go about it if in some places in the novel, *sun* is misspelled as *sin*, *son*, or *sbn*?

Figure 5.1 compares searching for a fixed literal value (`sun`) with finding a *regex pattern* (`s.n`) against a target string value: `sun soon son`.

As shown in figure 5.1, the literal search string `sun` finds one match, starting at position 0, in the target string `sun soon son`. Unlike the unmatched values, I've highlighted the matching value with a dark background. On the other hand, the regex pattern `s.n` (the dot is a *metacharacter* that can match any character) finds two

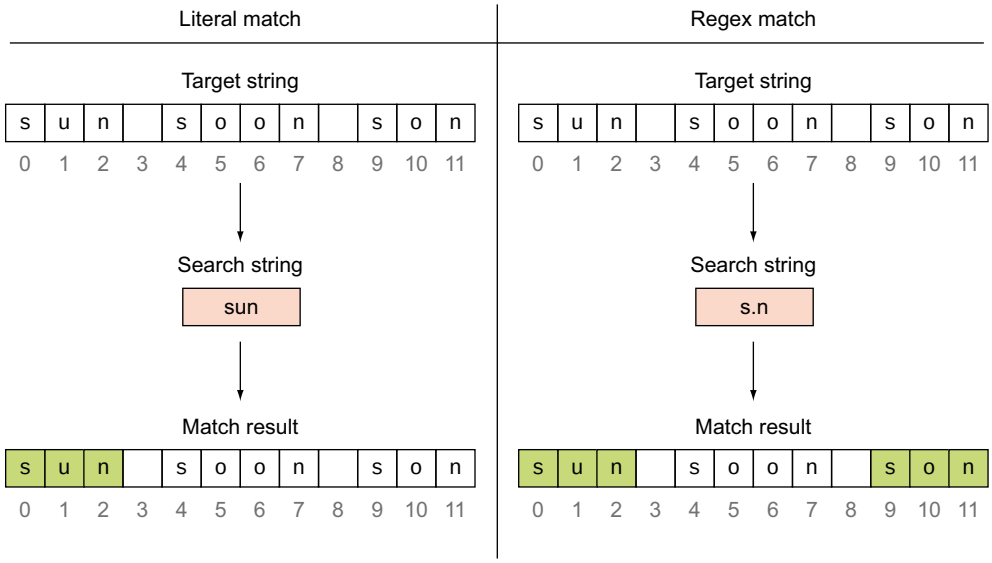


Figure 5.1 Comparing the matching of literal values with finding regex patterns

matches: `sun` and `son`, starting at positions 0 and 9 in the target search string `sun soon son`.

Similarly, you can use regex to find text that matches a pattern and to perform operations such as these:

- *Check method arguments*—Determine whether a string value starts with `Sh` and ends with either `y` or `a`. Determine whether author@manning is a valid email address.
- *Validate user input*—Verify whether 765-981-abc is a correct phone number.
- *Search usernames in a text file*—Find the ones that are exactly 10 characters long, starting with a letter *A* and followed by 4 digits and 5 letters.

All of these are common examples of needing regular expressions to *describe* your target data and find it in a stream (a string, a file, a network connection).

NOTE After you've found your target data, you can manipulate it any way you like: replace it, print it to a file, alert a user about invalid data, and so forth.

As I move on with this chapter, I'll show you how to use metacharacters, character classes, and quantifiers in regex. You'll see how to use them in code to search for matching data and manipulate it.

Before I take a plunge into this topic, let me reiterate that *regular expressions* is a relatively big topic, and this book limits its coverage to the exam topics. Though I can't guarantee that you'll start writing amazing regex after reading this chapter,

you'll definitely be comfortable using them, and of course, ready to answer the relevant exam questions.

Let's see what a regular expression is and how to evolve one.

5.1.1 What is a regular expression?

The example in the previous section showed you that exact matches might not be able to find all your matching data. In such cases, you need to define *patterns* of data (for example, `s.n`) that can match your target data. You can define these patterns by using regular expressions. Regular expressions come with a syntax (which we'll cover in the next few pages). With that syntax you can create a pattern to describe target search data.

What's the difference between describing data and specifying it? When you *describe* data, you detail out its attributes or characteristics. When you *specify* data, you state the exact data. In the example shown in figure 5.1, the regex `s.n` describes the data as follows:

- The first character must be *s*.
- Allow any character at the second position.
- The third character must be *n*.

NOTE *Regular expressions* is also referred to as a language because it has its own syntax. Regex refers to both the language and the data patterns that it defines.

Let's work with some examples so that this definition makes more sense. First up, character classes.

5.1.2 Character classes

Character classes aren't classes defined in the Java API. The term refers to a set of characters that you can enclose within square brackets (`[]`). When used in a regex pattern, Java looks for exactly *one* of the specified *characters* (not words).

Referring to our example of the novel, imagine that you want to search for all occurrences of the phrase *organized an event*. But *organized* is also written as *organised* (in the United Kingdom). Instead of searching your manuscript twice—first for *organized* and then for *organised*—you can use the character class `[sz]` in the search string `organi[sz]ed`. `[sz]` would match either *s* or *z* so you can find both *organized* and *organised*.

Let's work with another example, one you might see on the exam. Figure 5.2 shows how character class `[fdn]` is used to find an exact match of *f*, *d*, or *n*. With a target string *I am fine to dine at nine*, the regex `[fdn]ine` matches the words *fine*, *dine*, and *nine*, at positions 5, 13, and 21.

Let's see how you can use Java classes `Pattern` and `Matcher` (covered in detail later in the chapter) from the `java.util.regex` package to work with searching the text shown in the preceding example.

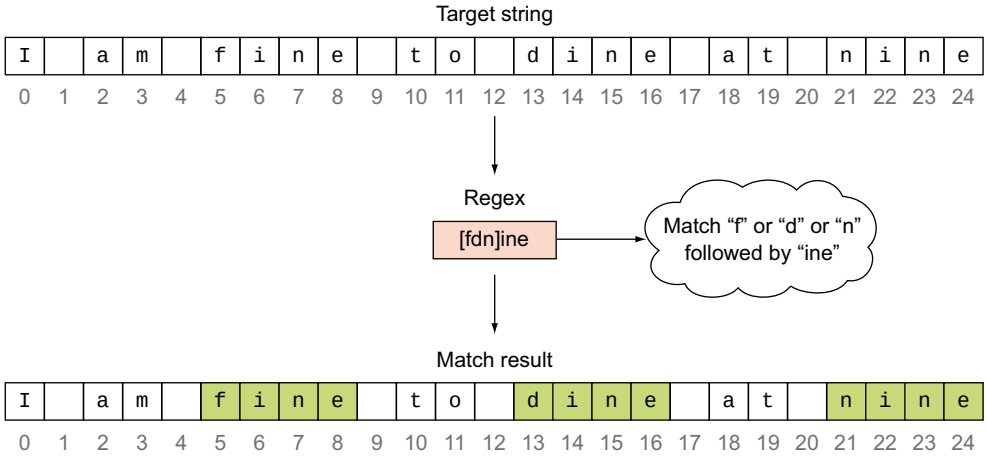


Figure 5.2 Character class `[fdn]` matches exactly one occurrence of *f*, *d*, or *n*.

Listing 5.1 Simple code to work with regex using `Pattern` and `Matcher`

```
import java.util.regex.*;
class UseRegex{
    public static void main(String[] args) {
        String targetString = "I am fine to dine at nine";
        String regex = "[fdn]ine";

        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(targetString);
        while (matcher.find()) {
            System.out.println(matcher.group() + " starts at " +
                               matcher.start() + ", ends at " +
                               matcher.end());
        }
    }
}
```

1 Target string to be searched

Regex pattern (2)

3 Instantiate Pattern using factory method compile().

Matcher created from pattern specifying target string (4)

5 while matches found

6 Prints matching text, start and end position.

NOTE Building a string by using the concatenation operators (+ and +=) isn't a recommended practice. Later in this chapter, you'll see how to use formatting classes and parameters such as `%s` and `%d` to include and format variable values in `String` literal values.

Here's the output of the preceding code:

```
fine starts at 5, ends at 9
dine starts at 13, ends at 17
nine starts at 21, ends at 25
```

The code at ❶ defines the target string to be searched. The code at ❷ defines the regex pattern. The code at ❸, class `Pattern`, a compiled representation of a regex,

is instantiated. Because this class doesn't define any public constructor, you must instantiate it by using its factory method `compile()`. Method `compile()` compiles a regular expression into a `Pattern` object. At ❹, class `Matcher` is instantiated by calling method `matcher()` on the `Pattern` instance. `Matcher` will match the given input against this pattern. Class `Matcher` is an engine that performs match operations on a character sequence by interpreting a regex pattern. It also doesn't define a public constructor.

Method `find()` of class `Matcher` returns `true` as long as it can find more matches of a regex in a target string. At ❺, the `while` loop executes for all matches found. The code at ❻ uses methods `group()`, `start()`, and `end()` to extract the matched string, its start position in the target string, and its end position in the target string.

Compare the values returned by `matcher.start()` and `matcher.end()` in the output shown for the preceding example and in figure 5.2. The substring *fine* occupies positions 5, 6, 7, and 8 in the target string *I am fine to dine at nine*. But `matcher.end()` returns the value 9. Beware of this on the exam. It can be combined in a tricky manner with other methods like `String`'s method `substring()`.

EXAM TIP If a matched string occupies index positions 1, 2, and 3 in a target string, method `end()` of class `Matcher` returns the value 4 for the corresponding call on `end()`. You can expect trick questions on this returned value on the exam.

Table 5.1 list examples of simple character classes that you can use to create regex patterns and find them in target data.

Table 5.1 Examples of regex patterns that use simple character classes

Class type	Regex pattern	Description
Simple	`[agfd]`	Match exactly one from a, g, f, or d
Range	`[a-f0-7]`	Match exactly one from the range a to f (both inclusive) or 0 to 7 (both inclusive)
Negation	`[^123k-m]`	Match exactly one character that is not 1, 2, or 3 or from the range k to m (both inclusive)

EXAM TIP If the Java Runtime engine determines that a pattern is invalid, it throws the runtime exception `PatternSyntaxException`. On the exam, when you see a question on the possible output of a string processing code, examine the regex pattern for invalid values.

5.1.3 Predefined character classes

Java's regex engine supports predefined character classes for your convenience. Table 5.2 lists the predefined classes included on this exam. You can test these regex using the class `UseRegex` included in listing 5.1.

NOTE To use a regex pattern that includes a backslash (\), you must *escape* the \ in the pattern by preceding it with another \. The character literal \ has a special meaning; it's used as an escape character. To use it as a literal, it must be escaped.

Table 5.2 Predefined character classes on this exam

Character class	Description
`.`	Any character (may or may not match line terminators)
`\d`	A digit: `[0-9]`
`\D`	A nondigit: `[^0-9]`
`\s`	A whitespace character: [space, `\t` (tab), `\n` (new line), `\x0B` (end of line), `\f` (form feed), `\r` (carriage)]
`\S`	A nonwhitespace character: `[^\s]`
`\w`	A word character: `[a-zA-Z_0-9]`
`\W`	A nonword character: `[^\w]`

The dot (`.`) is a metacharacter that matches *any* character. Metacharacters are special characters, which have special meanings. The search regex pattern `1.3` is not used to find *1.3* in the target string. It'll find the digit 1 followed by *any* character, followed by the digit 3. For example, it'll also find all of these: 123, 1M3, 193, 1)3, 1.3, 1,3, and 1+3.

Table 5.3 lists example target strings, regex patterns, and the result of finding a regex pattern in the target string. Figure 5.3, a pictorial representation of table 5.3, shows a target string, the regex pattern that is applied to the target string, and the results. The regex pattern that could be matched in the target string is highlighted

Table 5.3 Examples of target strings, regex patterns that use predefined character classes, and their matching results.

Target string	Regex	Match found	Start and end positions of match found
`A5C7M%`	`\d`	Yes	5 starts at 1, ends at 2 7 starts at 3, ends at 4
`A5C7M%`	`\D`	Yes	A starts at 0, ends at 1 C starts at 2, ends at 3 M starts at 4, ends at 5 % starts at 5, ends at 6

Table 5.3 Examples of target strings, regex patterns that use predefined character classes, and their matching results.

Target string	Regex	Match found	Start and end positions of match found
`A B 890`	`\s`	Yes	(First space) starts at 1, ends at 2 (Second space) starts at 3, ends at 4 (Third space) starts at 4, ends at 5
`A B $890`	`\S`	Yes	A starts at 0, ends at 1 B starts at 2, ends at 3 $ starts at 4, ends at 5 8 starts at 5, ends at 6 9 starts at 6, ends at 7 0 starts at 7, ends at 8
`A b$9;`	`\w`	Yes	A starts at 0, ends at 1 b starts at 2, ends at 3 9 starts at 4, ends at 5
`A b$9;`	`\W`	Yes	(Space) starts at 1, ends at 2 $ starts at 3, ends at 4 ; starts at 5, ends at 6

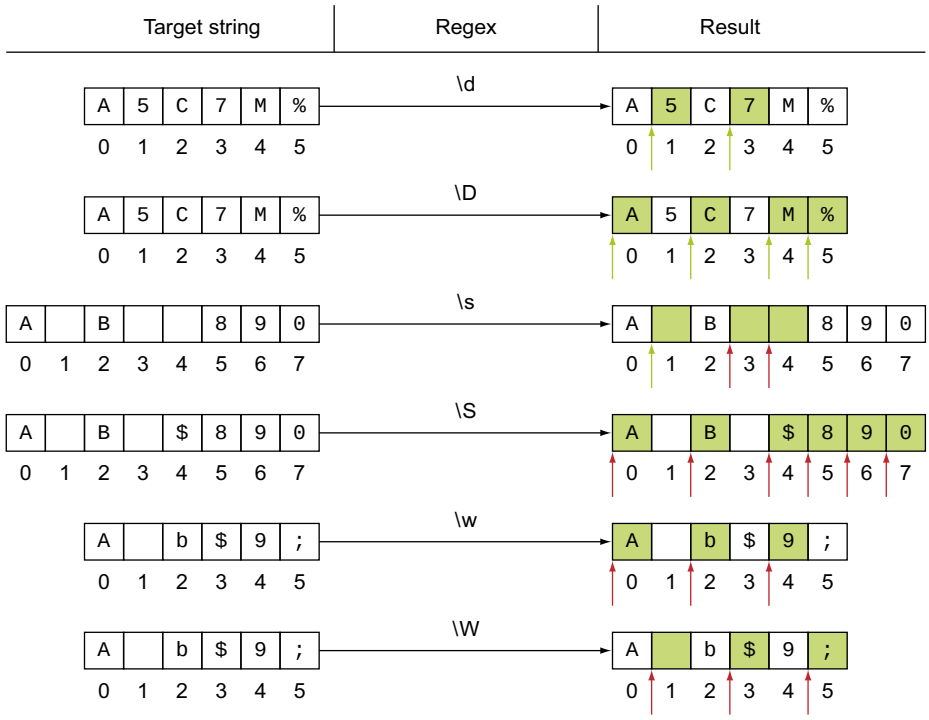


Figure 5.3 Pictorial representation of target strings, regex pattern applied to them, and the matches found, including matching positions

with a colored background. The starting position numbers of the matched string are marked with an up arrow.

Let's code an example that uses a predefined character class and replace all the matching occurrences with a literal string:

```
class UsePredefinedCharacterClass{
    public static void main(String[] args) {
        String targetString = "A b$9;";
        String regex = "\\W";

        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(targetString);
        String replacedStr = matcher.replaceAll("[]");

        System.out.println(replacedStr);
    }
}
```

Regex to be searched

Target search string

Instantiates Pattern and compiles regex pattern.

Creates a matcher that will match given input against this pattern.

Replaces all matches found with literal [].

Prints A[]b[]9[].

The preceding code uses the illustration in figure 5.3. It uses `Matcher`'s `replaceAll()` to replace all matching regex patterns in the target string with a string literal.

EXAM TIP Because `String` objects are immutable, calling `replaceAll()` won't change the contents of `String` referred to by the variable `targetString` in the preceding code example. `replaceAll()` creates and returns a new `String` object with the replaced values. Watch out for questions based on it on the exam.

5.1.4 Matching boundaries

Say you want to find all occurrences of the word *the* in your book. To do that, you'd need to search for the text *the*. Well, the same is true when you want to match *the*, which can be part of another word, for example, *their*, *leather*, or *seethe*. So you need a way to limit your searches to the start or end of a word. *Matching boundaries* can help you with this. You can match boundaries including the start of a line, a word, a nonword, or the end of a line by using regex patterns. Table 5.4 lists the boundary constructs that you're likely to see on the exam. Even though the boundary constructs ^ (beginning of line) and $ (end of line) aren't explicitly included in the exam objectives, you might see them in answer options that are incorrect. To ward off any confusion, I've included them in this section. These constructs also might be helpful in your projects at work.

Table 5.4 Boundary constructs on this exam

Boundary construct	Description
\b	A word boundary
\B	A nonword boundary
^	Beginning of a line
$	End of a line

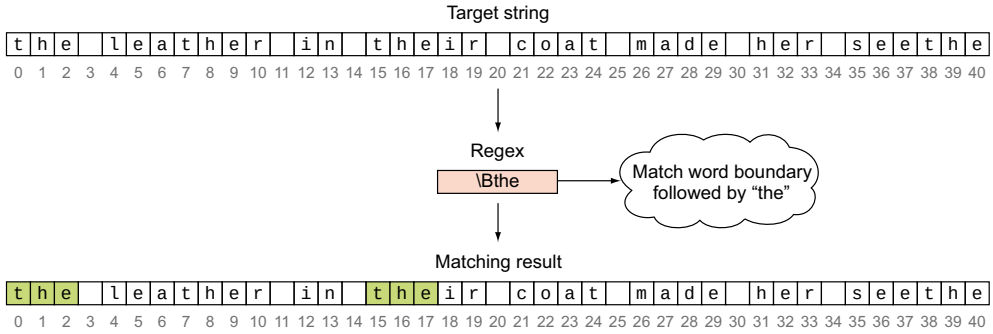


Figure 5.4 Matching regex pattern `\Bthe` against the string value `the leather in their coat made her seethe`. `\B` is a word boundary. When placed before the text *the*, it limits searches to finding words that *start* with *the*.

Let's see what happens when you match the regex pattern `\bthe` against the literal string value `the leather in their code made her seethe`, as shown in figure 5.4.

As shown in figure 5.4, `\bthe` matches words that start with *the*, including *the* and *their*. The first match is found at position 0, and the second match is found at position 15, in *their*. What if you want to find words that include *the* but don't start with *the*?

Let's modify the regex pattern `\bthe` used in the preceding example to `\Bthe`; instead of matching words that start with *the*, you'll match words that don't start with *the*. Figure 5.5 shows the matching values.

The regex pattern `\Bthe` matches all occurrences of *the* that aren't at the beginning of a word. For the match found at position 7 in the word *leather*, it doesn't matter whether *the* is followed by any other character or a word boundary. What do you think will be the output of matching the regex patterns `^the` and `the$` against the literal string value `the leather in their coat made her seethe`? Try it out using a simple code snippet.

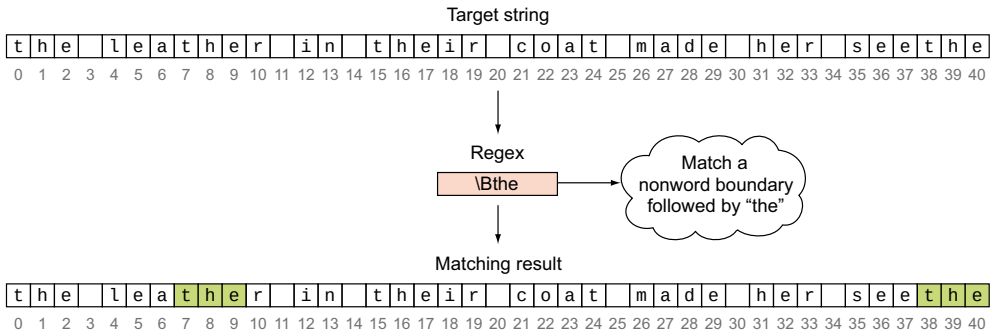


Figure 5.5 Matching regex pattern `\Bthe` against the string value `the leather in their coat made her seethe`. `\B` is a nonword boundary. When placed before the text *the*, it limits its searches to finding words that *don't start* with *the*.

It's time to test your skills in determining matching values for a regex pattern, in the first "Twist in the Tale" exercise.

Twist in the Tale 5.1

Consider the following string literal value:

```
String targetString = "The leather in their coat made her seethe";
```

Which of these options correctly defines a regex pattern for searching the literal string for *the* at either the beginning or end of a word, but not in its middle?

a `String regex = "\\Bthe\\B";`
b `String regex = "\\bthe\\B";`
c `String regex = "\\Bthe\\b";`
d `String regex = "\\bthe|the\\b";`

It's interesting to note that none of these regex options is invalid, and each produces output. Can you also determine the matched *the* for all these options?

5.1.5 Quantifiers

Imagine you want to search for the word *colour* or *color* in your book. A layman would search the book text first for *colour* and then again for *color*. A smart searching tool could search for both strings using a single search string by using quantifiers in the search string. You can specify the number of occurrences of a pattern to match in a target value by using quantifiers. The coverage of quantifiers is limited to greedy quantifiers on this exam. You could also use possessive or reluctant quantifiers. But because they aren't on the exam, I won't cover them any further. Table 5.5 describes the greedy quantifiers.

Table 5.5 Greedy quantifiers

Quantifier (Greedy)	Description
X?	Matching X, once or not at all
X*	Matching X, zero or more times
X+	Matching X, one or more times

The greedy quantifiers are so named because they make the matcher read the complete input string before starting to get the first match. If the matcher can't match the entire input string, it backs off the input string by one character and attempts again. If repeats this until a match is found or until no more characters are left. At the end, depending on whether you asked it to match zero, one, or more occurrences, it'll try to match the pattern against zero, one, or more characters from the input string.

USING ? TO MATCH ZERO OR ONE OCCURRENCE

- In all the preceding examples, we tried to search for exactly one occurrence of a particular digit or character. What if you want to match zero or one occurrence of a letter? For example, *color* in the United States is spelled *colour* in the United Kingdom. How will you match all occurrences of *colour* or *color* in a text file? The metacharacter ? can help: *Search string*—I am colour in UK and color in US
- *Regex*—`colou?r`
- Searches for *colour* or *color*
- In the preceding example, you apply ? to a single letter. You can also apply ? to a group of characters. How would you search for occurrences of *August* and *Aug* in a text? To do so, you can *group* the required characters by using parentheses and place ? right after them: *Search string*—It can be written as August or Aug
- *Regex*—`Aug(ust)?`
- Searches for *August* or *Aug*
- Imagine you need to search for the words *ball*, *mall*, *fall*, and *all* by using a regular expression in the target text: *Search string*—A ball can fall in a mall with all
- *Regex*—`[bmf]?all`
- Searches for *ball*, *mall*, *fall*, or *all*

In this example, because ? is applied to the character class `[bmf]`, it can be used to search for a single occurrence of either *b*, *m*, *f*, or none of these. Note that `[bmf]` without ? wouldn't match the text *all*.

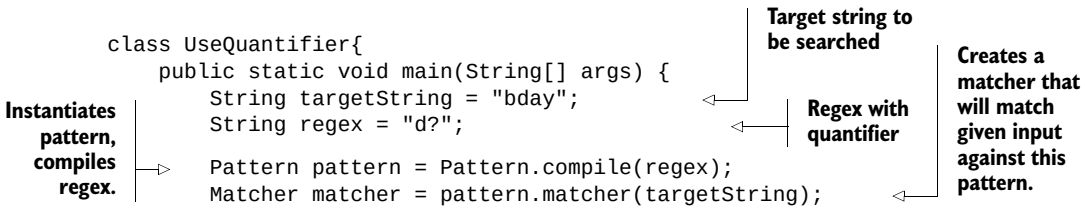
In the sections to follow, I'll cover working with a combination of ?, square brackets, and curly brackets. The set of combinations and permutations that can be used with a language feature makes it interesting, and at the same time overwhelming! The key to conquer such features is to understand a simpler concept before moving on to the next level.

ZERO LENGTH MATCHES WITH ?

Let's try to match the regex `d?` against the target string `bday`. Following are the target strings and the corresponding code with its output:

- *Search string*—bday
- *Regex*—`d?`
- Searches for zero or one occurrence of letter *d*

Following is the relevant code:




```
class UseQuantifier{
    public static void main(String[] args) {
        String targetString = "bday";
        String regex = "d?";

        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(targetString);
```

```
            while (matcher.find()) {
                System.out.printf("Found :%s: starts at %d, ends at %d",
                                  matcher.group(),
                                  matcher.start(),
                                  matcher.end());
                System.out.println();
            }
        }
}
```

NOTE You can use the preceding code (class `UseQuantifier`) to match a regex pattern against a string value, listing the start and end positions of the matches found.

Here's the output of this code:

```
Found :: starts at 0, ends at 0
Found :d: starts at 1, ends at 2
Found :: starts at 2, ends at 2
Found :: starts at 3, ends at 3
Found :: starts at 4, ends at 4
```

Does this output make you wonder why five matches are found? Remember that ? will match zero or one occurrence of the letter *d*. The regex engine found zero matches, with length 0, at positions 0, 2, 3, and 4. It found one match (with length 1) at position 1. Figure 5.6 illustrates the found matches.

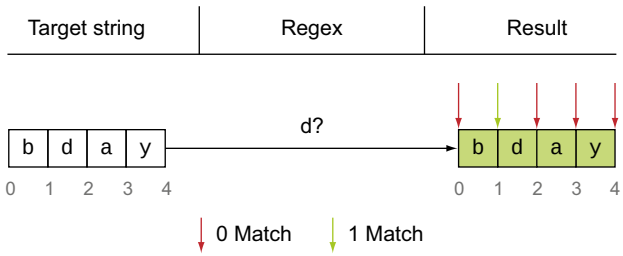


Figure 5.6 Matching regex pattern d? against the string value bday

USING * TO MATCH ZERO OR MORE OCCURRENCES

- You can use the metacharacter * to match zero or more occurrences of a regex. The regex `fo*d` will match all occurrences of words in which *o* occurs zero or more times: *Search string*—food, fod, fooodder, fd
- *Regex*—`fo*d`
- Matches *food*, *fod*, *foood*, *fd*
- You can also apply * to a group of characters. How would you search for text that starts with a letter, ends with a letter, and might contain zero or more digits in between? Let's create the regex pattern to search for this pattern. The letters could be either in lowercase or uppercase. You know that `\d` or `[0-9]` can be

used to match any digit and that `[A-Za-z]` can be used to match any letter in lowercase or uppercase. Because the digit can appear zero or more times, we evolve the regex as follows: *Search string*—b234a A6Z abc

- *Regex*—`[A-Za-z]\d*[A-Za-z]`
- Searches for text starting and ending with a letter and containing zero or more digits in between

The regex `[A-Za-z]\d*[A-Za-z]` matches `b234a`, `A6Z`, and `ab` in the preceding search string. It matches `ab` because `\d*` asked it to look for zero or more occurrences of digits. Because `ab` has zero digits between *a* and *b*, it's matched. It won't match `bc` because `b` was already consumed for matching `ab`.

- Now, what if you need to modify the preceding example, to limit the digits that appear between the letters to a range of 1 to 5? This is simple: just replace `\d*` with `[1-5]`. Examine the following: *Search string*—b234a A6Z abc
- *Regex*—`[A-Za-z][1-5]*[A-Za-z]`
- Searches for text starting and ending with a letter, and containing zero or more digits in the range 1–5 in between

The preceding regex `[A-Za-z][1-5]*[A-Za-z]` matches only `b234a` and `ab` in the target string `b234a A6Z abc`. It doesn't match `A6Z` because 6 isn't in the range 1–5.

Zero Length Matches with *

Since * matches zero or more occurrences of a pattern, it also occurs in zero length matches, that is, a match wherein the specified pattern can't be found. Revisit the previous subsection, "Zero length matches with ?". If you replace the pattern `d?` with `d*`, the code will output the same result because * also matches zero occurrences of a pattern, like the metacharacter `?`.

Using + to match one or more occurrences

- You can use the metacharacter `+` to match one or more occurrences of a regex. For example, the regex `fo+d` will match all occurrences of words, where *o* occurs one or more times: *Search string*—food, fod, fooodder, fd
- *Regex*—`fo+d`
- Matches *food*, *fod*, *foood*

You can also apply `+` to a group of characters. How would you search for text that starts with a letter and ends with a letter and may contain one or more digits in between? We know that `\d` can be used to match any digit and that `[A-Za-z]` can be used to match any letter in lowercase or uppercase. Because the digit can appear one or more times, you can evolve the regex as follows:

- *Search string*—b234a A6Z abc
- *Regex*—`[A-Za-z]\d+[A-Za-z]`
- Searches for text starting and ending with a letter and containing one or more digits in between
- Matches `b234a A6Z`

The regex matches `b234a` and `A6Z` in the preceding search string. It doesn't match `ab` because `\d+` asks it to look for one or more digits. Because `ab` has zero digits between `a` and `b`, it isn't matched!

Now, what if we need to modify the preceding example to limit the digits that appear between the letters in the range of 1 to 5? This is simple: just replace `\d+` with `[1-5]+`. Examine the following:

- *Search string*—b234a A6Z abc
- *Regex*—`[A-Za-z][1-5]+[A-Za-z]`
- Searches for text starting and ending with a letter and containing one or more digits in the range of 1–5 in between
- Matches `b234a`

This regex matches only `b234a` in the search string. It doesn't match `A6Z` because 6 isn't in the range 1–5. It doesn't match `ab` because it doesn't have a digit in the range of 1–5 between the letters *a* and *b*. Since `+` looks for one or more occurrences of a pattern, it *doesn't* qualify for zero length matches.

NOTE For this exam, you should know how to use the `?`, `*`, and `+` metacharacters in regex. Metacharacters have a different meaning for the regex engine.

If you don't remember how many instances `*`, `+`, and `?` match, table 5.6 lists a silly but simple set of questions and answers that might help you remember.

Table 5.6 Questions and answers to help remember the number of matches for `*`, `+`, and `?`

Metacharacter	Occurrence	Funny question	Silly answer to funny question
*	0 or many	How many *stars* can you see?	0 in a cloudy sky, many in a clear sky
?	0 or 1	What can be the answer to one of the most important *questions*: Do you love me?	Yes (1) or no (0)
+	1 or more	How many spouses can you *add* in your life?	1 or more

5.1.6 Java's regex support

Java incorporated regex support by defining the `java.util.regex` package in version 1.4. Regex in Java supports Unicode as it matches against `CharSequence` objects. This package defines classes for matching character sequences against the patterns specified by regular expressions. For this, you particularly need the `Matcher` and `Pattern` classes.

I've used these classes in the coding examples in the previous sections. Class `Pattern` is a compiled representation of a regular expression. It doesn't define a

public constructor. You can instantiate this class by using its factory method `compile()`. Here's an example:

```
Pattern pattern = Pattern.compile("a*b");
```

After you create a `Pattern` object, you must instantiate a `Matcher` object, which can be used to find matching patterns in a target string. Class `Matcher` is referred to as an engine that scans a target `CharSequence` for a matching regex pattern. Class `Matcher` doesn't define a public constructor. You can create and access a `Matcher` object by calling the instance method `matcher()` on an object of class `Pattern`:

```
Matcher m = p.matcher("aaaaab");
```

After you have access to the `Matcher` object, you can do the following:

- Match a complete input sequence against a pattern.
- Match the input sequence starting at the beginning.
- Find multiple occurrences of the matching pattern.
- Retrieve information about the matching groups.

NOTE Class `Matcher` is an engine that interprets a `Pattern` and matches it against a character sequence.

With the addition of the `java.util.regex` package, Java also added methods, like `matches()`, to existing classes, like `String`, which matched string values with the given regular expression. However, behind the scenes, `String.matches()` calls method `matches()` defined in class `Pattern`. At times, such methods might also manipulate the values themselves, *before* using classes `Pattern`/ `Matcher`, to support regex.

In the next section, I'll continue working with more examples of creating and using regex patterns, using classes `String`, `StringBuilder`, `Scanner`, and `StringTokenizer`. Class `String` defines multiple methods to search and replace string values based on exact and regex patterns. But class `StringBuilder` doesn't support search or replace methods based on regex. You'll see how you can use `Scanner` and `StringTokenizer` to parse and tokenize streams (such as text in a file) by using exact text or regex patterns.

NOTE The use of `StringTokenizer` is discouraged in new code. This legacy class is retained for backward compatibility. Use classes from the `java.util.regex` package or `String.split()` to get the functionality of `StringTokenizer`.

5.2 Searching, parsing, and building strings

When did you last search the internet for your favorite music, the latest news, or stock prices? Most people do that every day, every hour. Apart from searching the internet, people also search printed hardcopies. Searching text, and tokenizing and parsing it, are important and integral tasks to complete a lot of other tasks. Java includes multiple classes like `Scanner`, `StringTokenizer`, `StringBuilder`, `String`, and `Formatter` to accomplish searching, parsing, and building strings. Let's get started with how to search for exact matches and regex patterns.

[5.1] Search, parse, and build strings (including Scanner, StringTokenizer, StringBuilder, String, and Formatter)

[5.2] Search, parse, and replace strings by using regular expressions, using expression patterns for matching limited to . (dot), * (star), + (plus), ?, \d, \D, \s, \S, \w, \W, \b, \B, [], ()

5.2.1 Searching strings

Class `String` defines multiple methods to search strings for exact matches of a single character or string. These methods allow searching from the beginning of a string, or starting or ending at a specified position.

METHODS INDEXOF() AND LASTINDEXOF()

Both methods `indexOf()` and `lastIndexOf()` find a matching character or string in a string and return the matching position. Method `indexOf()` returns the *first* matching position of a character or string, starting from the specified position of the string, or from its beginning. Method `lastIndexOf()` returns the *last* matching position of a character in the entire string, or its subset (position `0` to the specified position). Figure 5.7 shows a pictorial representation of a string, use of these methods, and the positions of the matches found.

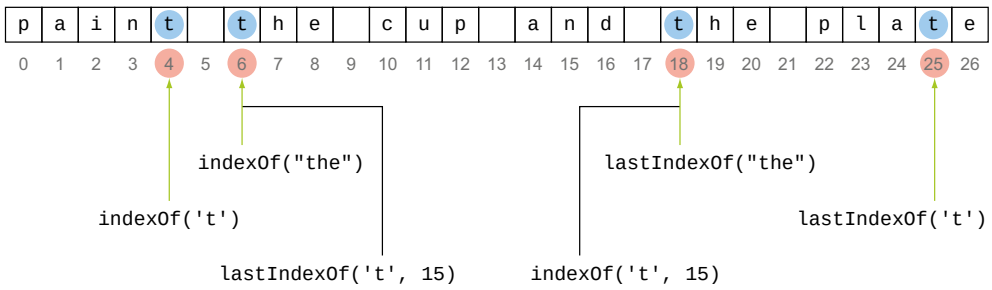


Figure 5.7 Showing use of methods `indexOf()` and `lastIndexOf()`

Here's the code that implements the methods shown in figure 5.7:

```
String sentence = "paint the cup and the plate";

System.out.println(sentence.indexOf('t'));          // Prints "4"
System.out.println(sentence.lastIndexOf('t', 15));  // Prints "6"

System.out.println(sentence.indexOf("the"));        // Prints "6"
System.out.println(sentence.indexOf('t', 15));      // Prints "18"

System.out.println(sentence.lastIndexOf("the"));    // Prints "18"
System.out.println(sentence.lastIndexOf('t'));      // Prints "25"
```

Both methods `indexOf()` and `lastIndexOf()` differ in the manner in which they search a target string: `indexOf()` searches in increasing position numbers, and `lastIndexOf()` searches backward. Due to this difference, `indexOf('a', -100)` will search the complete string, but `lastIndexOf('a', -100)` won't. In a similar manner, because `lastIndexOf()` searches backward, `lastIndexOf('a', 100)` will search this string, but `indexOf('a', 0)` or `indexOf('a', -100)` won't. This is shown in figure 5.8.

Here's the code that implements the methods shown in figure 5.8:

```
String sentence = "paint the cup and the plate";

System.out.println(sentence.indexOf('a'));          // Search
System.out.println(sentence.indexOf('a', 0));       // forward
System.out.println(sentence.indexOf('a', -100));
System.out.println(sentence.indexOf('a', 100));

System.out.println(sentence.lastIndexOf('a'));      // Search
System.out.println(sentence.lastIndexOf('a', 0));   // backward
System.out.println(sentence.lastIndexOf('a', 100));
System.out.println(sentence.lastIndexOf('a', -100));
```

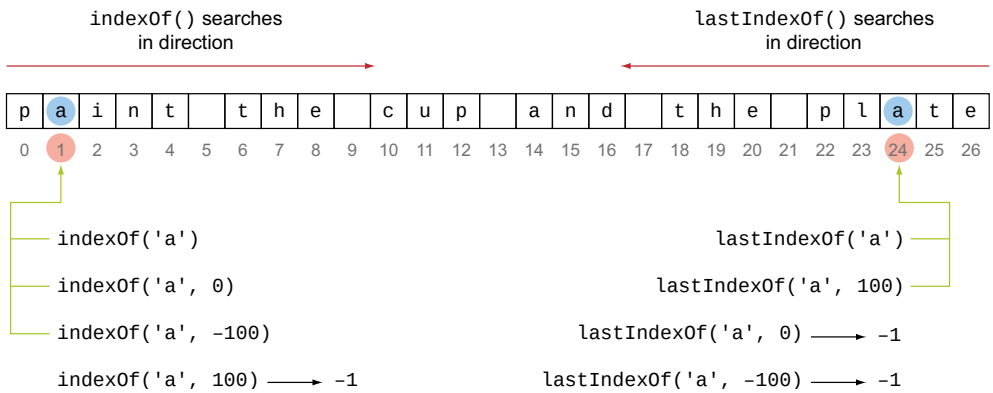


Figure 5.8 Methods `indexOf()` and `lastIndexOf()` search a target string in different directions.

EXAM TIP Methods `indexOf()` and `lastIndexOf()` don't throw a compilation error or runtime exception if the search position is negative or greater than the length of the string. If no match is found, they return `-1`.

METHOD CONTAINS()

Method `contains()` searches for exact matches in a string and returns `true` if a match is found, `false` otherwise. Because `contains()` accepts a method parameter of type `CharSequence`, you can pass to it both a `String` or a `StringBuilder` object:

```
String sentence = "paint the cup and the plate";
StringBuilder sb = new StringBuilder("the");
String str = "the";

System.out.println(sentence.contains(sb));
System.out.println(sentence.contains(str));
```

StringBuilder object

String object

Searches matching StringBuilder value in target string; prints "true".

Searches matching string value in target string; prints "true".

METHODS SUBSEQUENCE() AND SUBSTRING()

Both methods `subSequence()` (uppercase *S*) and `substring()` (no uppercase letter) return a substring of the string. Here's the signature of these methods:

```
CharSequence subSequence(int beginIndex, int endIndex)
String substring(int beginIndex)
String substring(int beginIndex, int endIndex)
```

Returns new character sequence that's subsequence of this sequence

Returns new string that's substring of this string

Returns new string that's substring of this string

EXAM TIP To remember the return types of methods `subSequence()` and `substring()` on the exam, just remember that the names of these methods can be used to determine their return type. Method `subSequence()` returns `CharSequence`, and method `substring()` returns `String`.

Method `subSequence()` simply calls method `substring()`; it was added to class `String` in Java 1.4 to support implementation of the interface `CharSequence` by class `String`. Method `substring()` defines overloaded versions, which accept one or two `int` method parameters to specify the start or the end positions. Method `subSequence()` defines only one variant: the one that accepts two `int` method parameters for the start and the end position. For the exam, you must remember that these methods don't include the character at the end position, as shown in figure 5.9.

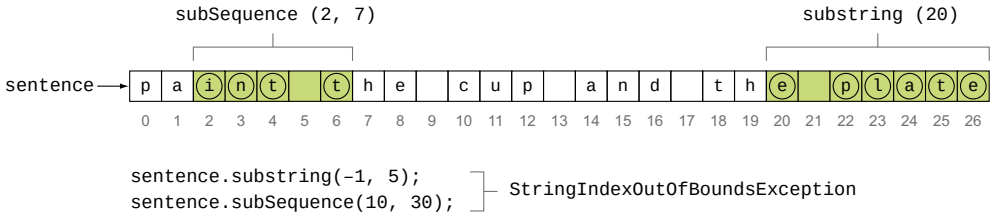


Figure 5.9 Methods `subSequence()` and `substring()` don't include the character at the last position in their return value.

EXAM TIP Methods `subSequence()` and `substring()` don't include the character at the end position in the result string. Also, unlike methods `indexOf()` and `lastIndexOf()`, they throw the runtime exception `StringIndexOutOfBoundsException` for invalid start and end positions. The subtraction value from `endIndex – beginIndex` is the number of chars these methods will return.

METHOD SPLIT()

Method `split(String regex)` and method `split(String regex, int limit)` in class `String` search for a matching regex pattern and split a string into an array of string values. Figure 5.10 shows how the string `paint-the-cup-cop-and-cap` is split with the regex pattern `c.p`. As discussed in the previous section on regex, the dot in regex `c.p`, will match exactly one character.

NOTE *Tokenizing* is the process of splitting a string, based on a separator, into tokens. A separator can be a character, text, or a regex. For example, if string `1234;J Perry;94.75` is split using a semicolon as the separator, the tokens that you'll get are `1234`, `J Perry`, and `94.75`.

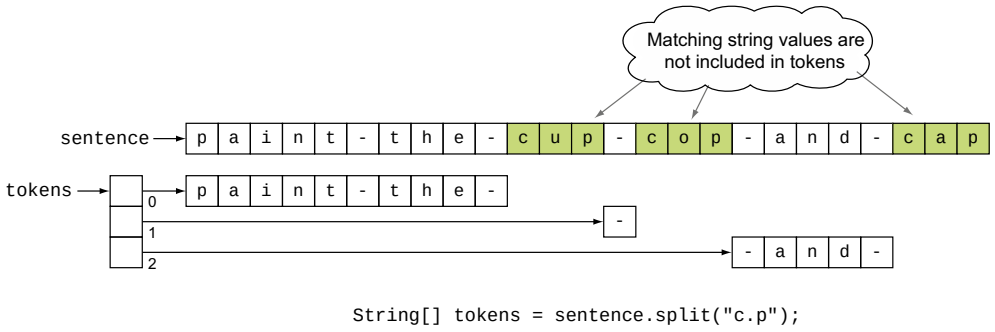


Figure 5.10 The `String` array returned by `split()` doesn't contain the values that it matches to split the target string.

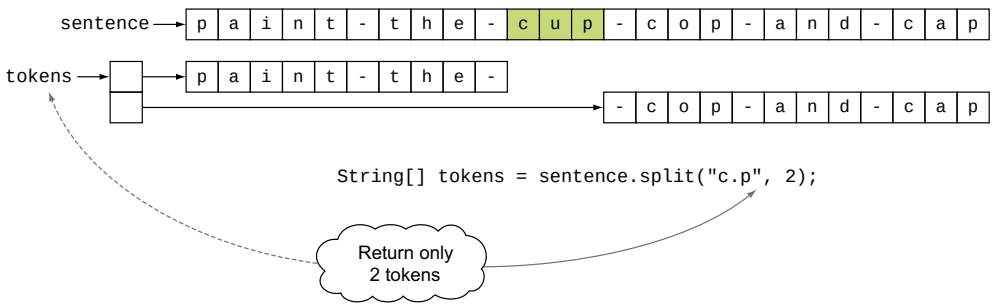


Figure 5.11 You can limit the maximum number of tokens returned by `split()`.

You can limit the maximum number of tokens that you want to retrieve by using `split(String regex, int limit)`. Figure 5.11 shows how the target string `paint-the-cup-cop-and-cap` is split with `split("c.p", 2)`. Because the total number of tokens is limited to two, the regex pattern `c.p` is matched only once. The remaining string after the first match is stored as the second array value.

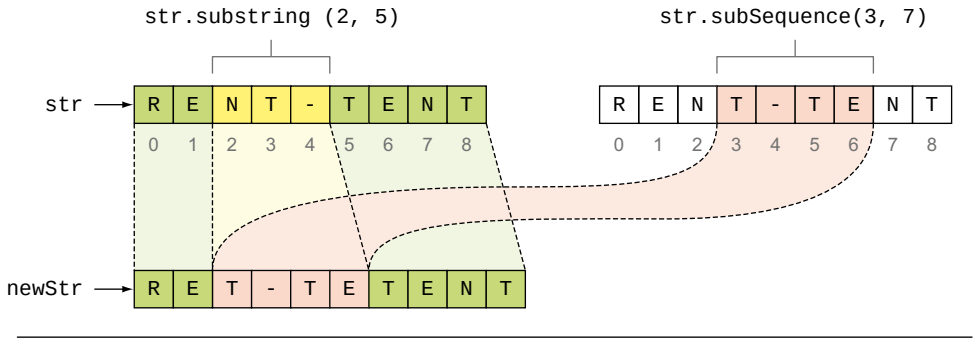
If `limit` is nonpositive, then the regex pattern will be applied as many times as possible and the array `tokens` can have any length. If `limit` is passed `0`, the regex pattern will be applied as many times as possible, but `tokens` won't include trailing empty strings.

5.2.2 Replacing strings

Finding and replacing characters or text is a common requirement. You can use multiple methods to find and replace text or regex by using class `String`. As mentioned previously, the methods that accept the interface `CharSequence` as a parameter can accept arguments of all the implementing classes: `String`, `StringBuffer`, and `StringBuilder`. Table 5.7 lists the replacing methods defined in class `String`.

Table 5.7 Methods to replace string values, using exact matches and regex patterns

Method	Description
`replace(char old, char new)`	Returns a new string resulting from finding and replacing *all* occurrences of `old` character with `new` character
`replace(CharSequence old, CharSequence new)`	Returns a new string resulting from finding and replacing each substring of this string that matches the `old` target sequence with the specified `new` replacement sequence
`replaceAll(String regex, String replacement)`	Replaces each substring of this string that matches the given regular expression with the given replacement
`replaceFirst(String regex, String replacement)`	Replaces the first substring of this string that matches the given regular expression with the given replacement




```
newStr = str.replace(str.substring(2, 5), str.subSequence(3, 7));
```

Figure 5.12 An example of using `replace()` to create a new string (`newStr`) that replaces a substring of `str` with another substring of `str`

METHOD REPLACE()

On the exam, you're likely to see chained method invocation with `String` methods. What happens when method `replace()` tries to replace a substring of a string with another substring of the same string? Also, what happens if these string values that are searched and replaced overlap? Here's an example:

```
String str = "RENT-TENT";

String newString = str.replace(
                        str.substring(2, 5),
                        str.subSequence(str.indexOf("T"),
                                        str.lastIndexOf('N')));
System.out.println(newString);
```

Figure 5.12 helps explain this code. To make the code simpler to show and understand, I've replaced `str.indexOf("T")` and `str.lastIndexOf('N')` with their return values—that is, 3 and 7—which are passed as arguments to `str.subSequence()`.

As shown in figure 5.12, method `replace()` creates and returns a new string, `newStr`, by replacing the occurrence of the characters it finds at positions 2, 3, and 4 of `str`, with the characters it finds at positions 3, 4, 5, and 6 of `str`. The length of the replacement string can be greater than, equal to, or smaller than the substring that it replaces.

METHOD REPLACEALL()

This method searches for matching regex patterns in a string and replaces them with the specified string value. An example follows:

```
String str = "cat cup copp";
String newString = str.replaceAll("c.p\\B", "()");
System.out.println(newString);
```

Target string to be searched

Finds regex pattern c.p\B and replaces it with ().

Prints "cat cup ()p".

If no match in the target string is found, `replaceAll()` returns the contents of the original string.

EXAM TIP Unlike `replace()`, `replaceAll()` doesn't accept method parameters of type `CharSequence`. Watch out for the passing of objects of class `StringBuilder` to `replaceAll()`.

The combination of the overloaded methods `replace()`, `replaceAll()`, and `replaceFirst()` can be confusing on the exam. Take note of the method parameters that can be passed to each of these methods. Let's attempt our next "Twist in the Tale" exercise, which should help you get a better grasp of all these string replacement methods and regex patterns.

Twist in the Tale 5.2

I've modified the code used in a previous example for this exercise. Execute this code on your system and select the correct answer.

```
class ReplaceString2 {
    public static void main(String[] args) {
        String str = "cat cup copp";
        String newString = str.replaceAll("c.p\\b", "()");    //line4
        System.out.println(newString);
    }
}
```

a The code outputs `cat () copp`.

b The code outputs `cat cup ()p`.

c The code outputs `cat cup copp`.

d If code marked with comment line 4 is replaced with the following code, it'll output `cat () copp`:

```
String newString = str.replaceFirst("c.p\\b", "()");
```

e If code marked with comment line 4 is replaced with the following code, it'll output `cat () ()p`:

```
String newString = str.replace("c.p", "()");
```

f If code marked with comment line 4 is replaced with the following code, it'll output `cat cup copp`:

```
String newString = str.replace(new StringBuilder("cat"), "()");
```

OTHER METHODS

For this exam, you must also know about other commonly used `String` methods that you can use to compare string values, match a substring, and determine whether a string starts or ends with a literal value, as listed in table 5.8.

Table 5.8 Methods for comparing string values

Method	Description
`endsWith(String suffix)`	Returns `true` if this string ends with the specified `suffix`
`startsWith(String prefix)`	Returns `true` if this string starts with the specified `prefix`
`startsWith(String prefix, int offset)`	Returns `true` if the substring of this string beginning at the specified index starts with the specified `prefix`
`compareTo(String anotherStr)`	Compares this string with `anotherStr` lexicographically. Returns a negative, zero, or positive value depending on whether this string is less than, equal to, or greater than `anotherStr`.
`compareToIgnoreCase(String anotherStr)`	Compares this string with `anotherStr` lexicographically, ignoring case differences. Returns a negative, zero, or positive value depending on whether this string is less than, equal to, or greater than `anotherStr`.
`equals(Object object)`	Returns `true` if the object being compared defines the same sequence of characters
`equalsIgnoreCase(String anotherStr)`	Compares this `String` to `anotherStr`, ignoring case considerations

You need to be careful with `String` class methods that accept integer values as index positions to start or end their searches. Here's an example of the values returned by the overloaded method `startsWith()`, when you pass negative, zero, or positive values to it:

```
String str = "Start startup, time to start";
System.out.println(str.startsWith("Start"));
System.out.println(str.startsWith("Start", 0));

System.out.println(str.startsWith("Start", -1));
System.out.println(str.startsWith("Start", 1));
```

Prints "true"

Prints "false"

When comparing letters, is `a` greater than, smaller than, or equal to `A`? When comparing letters lexicographically, note that a letter in lowercase is greater than its uppercase. The following example outputs a positive value:

```
String a = "a";
String b = "A";
System.out.println(a.compareTo(b));
System.out.println(b.compareTo(a));
```

Outputs positive number

Outputs negative number

EXAM TIP Lexicographically, a lowercase letter is greater than its equivalent uppercase letter. Method `compareTo()` returns a negative number (not necessarily –1) if the `String` object on which it's called is lexicographically smaller than the one it's compared to.

Don't let the simplicity of these methods take you for a ride. When chained, they can become difficult to answer. What do you think is the output of the following code?

```
String str = "Start startup, time to start";
System.out.println(str.substring(0,1).compareTo(str.substring(6,7)));
```

Execute the preceding code on your system and find out the answer yourself.

The next section covers how to parse and tokenize `String` by using class `Scanner`.

5.2.3 Parsing and tokenizing strings with Scanner and StringTokenizer

Parsing is the process of analyzing a string to find tokens or items. For example, you can parse the text `9187` to find the integer value `9187`. As you know, the same number can be stored as text or as an integer value.

SCANNER CLASS

Class `Scanner` can be used to parse and tokenize streams. (Streams are covered in chapter 7.) `Scanner` is a simple text scanner, which can parse primitive types and strings. `Scanner` tokenizes its input string by using a pattern, which can be a character, a string literal, or a regex. You can then use the resulting tokens, converting them to different data types. Here's a quick look at the constructors relevant for this exam:

```
Scanner(File source)
Scanner(Readable source)
Scanner(String source)
```

Use File as source to Scanner.

FileReader, BufferedReader implement Readable.

Use String as source to Scanner.

Now that you know the sources available to `Scanner`, let's work with an example of tokenizing a `String` with the default delimiter:

```
Scanner scanner = new Scanner("The \tnew \nProgrammer exam");
while (scanner.hasNext())
    System.out.println (scanner.next());
```

String contains spaces, tab (\t), and newline character (\n).

Here's the output of this code:

```
The
new
Programmer
exam
```

If no delimiter is specified, a pattern that matches whitespace is used by default for a `Scanner` object. You can specify the required regex by calling its method `useDelimiter()` as follows:

```
Scanner scanner = new Scanner("The1new22Programmer exam6");
scanner.useDelimiter("[\\d]+");
while (scanner.hasNext())
    System.out.println(scanner.next());
```

Target string with letters and digits

Uses regex [\d}+ to tokenize text.

Finds and returns next token

The output of this code is as follows:

```
The
new
Programmer exam
```

What if you want to do the reverse in this example: print out the numbers 1, 22, and 6 and leave the characters? You just change the regex to be used as a delimiter, as follows:

```
scanner.useDelimiter("[\\sA-Za-z]+");
```

Reads regex [\sA-Za-z]+ as a match for 1 or more occurrences of whitespace, character, or letter in uppercase or lowercase.

Method `next()` defined in class `Scanner` returns an object of type `String`. This class defines multiple `nextXXX()` methods, where `XXX` refers to a primitive data type. Instead of returning a string value, these methods return the value as the corresponding primitive type.

```
Scanner scanner = new Scanner ("Shreya,20,true");
scanner.useDelimiter(",");
System.out.println(scanner.next());
System.out.println(scanner.nextInt());
System.out.println(scanner.nextBoolean());
```

Retrieves next string value

Retrieves next int value

Retrieves next Boolean value

The following example parses the target string by using the regex `[\\sA-Za-z]+`, which is for one or more occurrences of a whitespace or a letter. The tokenized numbers are retrieved using `nextInt()`:

Target string contains combination of words, numbers, spaces

```
Scanner scanner = new Scanner ("1 2 4 The new 55 Programmer 44 exam");
scanner.useDelimiter("[\\sA-Za-z]+");
int total = 0;
while (scanner.hasNextInt())
    total = total + scanner.nextInt();
System.out.println(total);
```

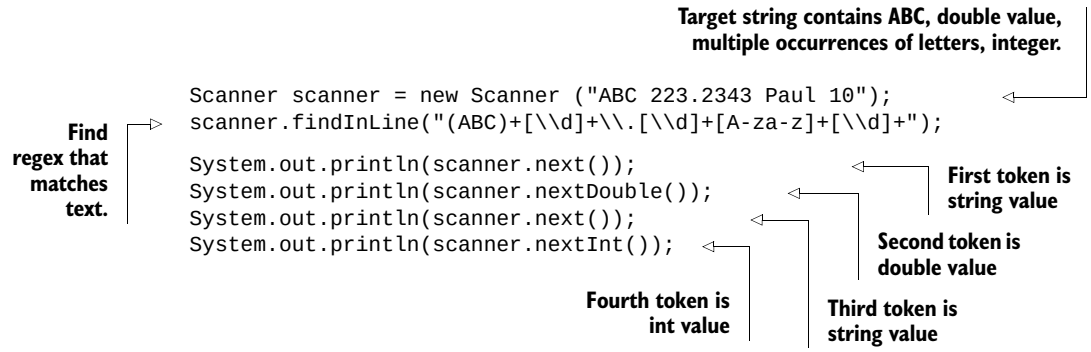
Delimiter is any combination of whitespace/letters.

nextInt returns int value.

The output of this code is `106` (1 + 2 + 4 + 55 + 44). For the exam, make note of the multiple `hasNext()`, `hasNextXxx()`, `next()`, and `nextXxx()` methods. Methods `hasNext()` and `hasNextXxx()` only return `true` or `false` but don't advance. Only methods `next()` and `nextXxx()` advance in the input. So you'll have to be careful or you'll end up with a program which runs eternally, like this one:

```
Scanner scanner = new Scanner ("1 2 4 The new 55 Programmer 44 exam");
scanner.useDelimiter("[\\s]+");
int total = 0;
while (scanner.hasNext())
    if (scanner.hasNextInt())
        total = total + scanner.nextInt();
System.out.println(total);
```

Class `Scanner` also defines method `findInLine()`, which tries to match the specified pattern with no regard to delimiters in the input. Here's an example:



The output of this code is as follows:

```
ABC
223.2343
Paul
10
```

NOTE The decimal values (`float` and `double`) are `Locale`-specific (`Locale` is covered in chapter 12). For decimal numbers, a `Locale` might use a decimal comma or a decimal point.

What happens when there's a mismatch in the next token and the method used to retrieve the data? Does it lead to a compilation error or a runtime exception? Let's uncover an important concept in our next "Twist in the Tale" exercise.

Twist in the Tale 5.3

The following code contains a mismatch in the type of token retrieved (`a` can be stored as a character or string) and the method (`nextInt`) used to retrieve this data. What's the output of the following code?

```
import java.util.*;
public class MyScan {
    public static void main(String[] args) {
        String in = "1 a 10 . 100 1000";
        Scanner s = new Scanner(in);
        int accum = 0;
        for(int x = 0; x < 4; x++) {
            accum += s.nextInt();
        }
        System.out.println(accum);
    }
}
```

a The code prints 1111.
b The code doesn't compile.
c The code throws `java.util.InputMismatchException`.
d The code throws `java.util.MismatchException`.
e The code throws `java.util.ParsingException`.

STRINGTOKENIZER CLASS

You can use the `StringTokenizer` class to break a string into tokens. To separate the tokens you can specify the delimiter to be used either at the time of instantiating `StringTokenizer` or on a per-token basis. In the absence of an explicit delimiter, a whitespace is used.

```
StringTokenizer st = new StringTokenizer("start your startup");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

The preceding code uses a whitespace as a delimiter and outputs:

```
start
your
startup
```

Methods `hasMoreTokens()` and `hasMoreElements()` in class `StringTokenizer` return a `boolean` value indicating whether more tokens are available or not. Methods `nextToken()` and `nextElement()` return the next token. The return type of method `nextToken()` is `String` and of method `nextElement()` is `Object`.

If the delimiter used to instantiate `StringTokenizer` is `null`, the constructor doesn't throw an exception. But trying to access the resultant tokens or invoking any other method on the `StringTokenizer` instance results in a `NullPointerException`:

```
StringTokenizer st = new StringTokenizer("start your startup", null);
System.out.println(st.hasMoreElements());
```

Throws NullPointerException because delimiter is null.

NOTE `StringTokenizer` is a legacy class that's retained for compatibility reasons. Oracle recommends use of method `split()` of class `String` or classes from a regex package to get the same functionality.

Apart from searching, parsing, and manipulating strings, formatting strings is another frequently used feature. Let's format some string values.

5.3 Formatting strings

[5.3] Format strings using the formatting parameters %b, %c, %d, %f, and %s in format strings

How often do you use the operator `+` to concatenate a `String` value with the value of another variable (for example, `"name:"+emp+"age:"+age`)? Do you find this expression cumbersome to use? This section discusses formatting classes, methods, and parameters that can replace the `+` operator for concatenating `String` and other variable values. You can also write the formatted text to `OutputStream`, `File`, or `StringBuilder`. Let's start with identifying the classes that you can use to format strings.

5.3.1 Formatting classes

Class `java.util.Formatter` and I/O classes like `PrintStream` and `PrintWriter` define methods to format strings (coverage of formatting classes is limited to exam topics).

You can use class `Formatter` to write formatted strings to a file, stream, or `StringBuilder` objects. Class `Formatter` is an interpreter for `printf`-style format strings. This class provides support for layout justification and alignment; common formats for numeric, string, and date/time data; and locale-specific output. This type of formatted printing is heavily inspired by C's `printf`.

On the exam you'll be queried on how to write formatted strings to the standard output. You can use class `System` to access the standard input, standard output, and error output streams. The standard output in class `System` is made accessible using a static variable, `out`, which is of type `PrintStream` (remember using `System.out.println()`). Class `PrintStream` defines methods to output formatting strings.

5.3.2 Formatting methods

Class `Formatter` defines the overloaded method `format()`to write a formatted string to its destination using the default or specified `Locale` (covered in chapter 12), format string, and arguments:

```
format(String format, Object... args)
format(Locale l,String format, Object... args)
```

To output formatted strings to the standard output, you can use class `PrintStream` and its overloaded methods `format()` and `printf()`, which use the default or specified `Locale`, format strings, and arguments:

```
format(String format, Object... args)
format(Locale l,String format, Object... args)
printf(String format, Object... args)
printf(Locale l, String format, Object... args)
```

Behind the scenes, method `printf()` simply calls method `format()`. Also the number of method arguments that these methods accept is the same. So you really need to work with one method to format the strings.

EXAM TIP `System.out.println()` can't write formatted strings to the standard output; `System.out.format()` and `System.out.printf()` can.

Here's a quick example to write a formatted string to the standard output and to a file:

```
import java.util.Formatter;
import java.io.File;
class FormattedStrings {
    public static void main(String args[]) throws Exception {
        String name = "Shreya";

        Formatter formatter = new Formatter(new File("data.txt"));
        formatter.format("My name is %s", name);
        formatter.flush();

        System.out.printf("My name is %s", name);
    }
}
```

Destination is file data.txt.

Format parameter %s.

Destination is standard output.

In the preceding code, `%s` is replaced with the value of variable `name`. In the next section, let's see how to define format strings, the formatting parameters (on the exam), and their elements.

NOTE Because class `Locale` and Java I/O classes are covered in chapters 12 and 7, I won't include other examples about these classes here.

5.3.3 Defining format strings

To use methods `format()` and `printf()`, you need to define a format string that defines *how* to format text and an object argument list that defines *what* to format. You can define a combination of fixed text and one or more embedded format specifiers, to be passed to formatting methods. The format specifier takes the following form:

```
%[argument_index$][flags][width][.precision]conversion_char
```

Table 5.9 describes the format specifier elements.

Table 5.9 Format specifier elements and their purposes

Format specifier element	Optional/ required	What it means
argument_index	Optional	Decimal integer indicating the position of the argument in the argument list. The first argument is referenced by `1$`, the second by `2$`, and so forth.
flags	Optional	Set of characters that modify the output format. The set of valid flags depends on the conversion.
width	Optional	A non-negative decimal integer indicating the minimum number of characters to be written to the output.
precision	Optional	A non-negative decimal integer usually used to restrict the number of characters. The specific behavior depends on the conversion.
conversion_char	Required	A character indicating how the argument should be formatted. The set of valid conversions for a given argument depends on the argument's data type.

NOTE The conversion characters on the exam are `%b`, `%c`, `%d`, `%f` ,and `%s`.

How to work with format specification

The general syntax is as follows:

```
%[arg_index$][flags][width][.precision]conversion_char
```

- Compulsory element—A format specification must start with a `%` sign and end with a conversion character.
- Optional elements—`arg-index`, `flags`, `width`, and `.precision` are all optional.
- Anything before a `%` and after the conversion character is printed as it is. For example, `printf("xxx%1$dyyy%2$dzzz", 10, 20)` outputs `xxx10yyy20zzz`.
- You need to remember the following flags:
  - `-` Left-justify this argument; must specify width as well.
  - `+` Include a sign (`+` or `-`) with this argument. Applicable only if conversion character is `d` or `f` (for numbers).
  - `0` Pad this argument with zeros. Applicable only if conversion character is `d` or `f` (for numbers). Must specify width as well.
  - `,` Use locale-specific grouping separators (for example, the comma in 123,456). Applicable only if conversion character is `d` or `f` (for numbers).
  - `(` Enclose negative numbers in parentheses. Applicable only if conversion character is `d` or `f` (for numbers).

Let's get started with how to use the formatting parameter `%b`.

5.3.4 Formatting parameter %b

If the target argument `arg` is `null`, then `%b` prints the result as `false`. If `arg` is `boolean` or `Boolean`, the result is the `String` returned by `String.valueOf()`. Otherwise, the result is `true`.

```
String name = "Shreya";
Integer age = null;
boolean isShort = false;

System.out.format("Name %b, age %b, isShort %b", name, age, isShort);
```

%b prints "true" for non-null values.

%b prints "false" for null values.

%b evaluates and outputs boolean value

The preceding code formats the text as follows:

```
Name true, age false, isShort false
```

Now, what happens if there's a mismatch in the number of arguments passed to method `format()` and the number of times `%b` appears in the format string? If the number of arguments exceeds the required count, the *extra* variables are quietly ignored by the compiler and JVM. In the following example, the extra variables in the first line are ignored and it outputs `Name true`. But for the second line because the number of required arguments falls short, the JVM throws `java.util.MissingFormatArgumentException` at runtime:

```
System.out.format("Name %b", name, age, isShort);
System.out.printf("Name %b, age %b", name);
```

Ignores extra variables

Throws runtime exception

EXAM TIP If the count of formatting parameters is more than the arguments passed to methods `format()` or `printf()`, then `java.util.MissingFormatArgumentException` is thrown at runtime.

This format specifier accepts primitive and reference variables and, hence, you can pass any type of argument to this format specifier. Examine the following examples, which use different combinations of *flags*, *width*, and *precision*:

```
System.out.format("\nName defined %10b.", name);
System.out.format("\nName defined %.1b.", name);
System.out.format("\nName defined %-10b.", name);
```

Minimum width specified as 10 adds 6 spaces before result true.

Precision 1 truncates length of result to t.

Flag left-justifies the text true.

Here's the output of the preceding code:

```
Name defined       true.
Name defined t.
Name defined true      .
```

EXAM TIP You can pass any type of primitive variable or object reference to %b.

5.3.5 Formatting parameter %c

%c outputs the result as a Unicode character. Examine the following examples:

```
System.out.printf("\nChar %c", new Character('\u007b'));
System.out.printf("\nChar %c", '\u6124');
System.out.printf("\nChar %c", new Boolean(true));
System.out.printf("\nChar %c", '\affff');
```

Prints "{"

Prints "(0m)" in Devanagari script.

Throws runtime exception if target can't be converted to Unicode character.

Values with invalid Unicode values (\affff) won't compile.

If the target can't be converted to a Unicode character, a runtime exception is thrown:

```
Exception in thread "main" java.util.IllegalFormatConversionException: c !=
java.lang.Boolean
```

EXAM TIP You can pass only literals and variables that can be converted to a Unicode character (`char`, `byte`, `short`, `int`, `Character`, `Byte`, `Short`, and `Integer`) to the %c specifier. Passing variables of type `boolean`, `long`, `float`, `Boolean`, `Long`, `Float`, or any other class will throw `IllegalFormatConversionException`.

5.3.6 Formatting parameters %d and %f

Did you ever notice that the amount displayed in your bank statements is formatted in a particular manner—say, for example, a maximum width of 20 digits, with exactly 2 digits after the decimal point, and grouped according to the locale-specific information? Let's see how you can do this by formatting a `float` or `double` value, using the format specifier %f. In the following examples, I've used square brackets in the results displayed ([]) to help you determine how the numbers are formatted with padding and left justification:

```
System.out.printf("[%f]", 12.12345);
System.out.printf("[%010f]", 12.12345);
System.out.printf("[%-10f]", 12.12345);
System.out.printf("[%10.2f]", 12.98765);
System.out.printf("[%,f]", 123456789.12345);
```

Prints "[12.123450]".

Outputs value with width 10, zero padded; prints "[012.123450]".

Value with width 10, left-justified; prints "[12.123450]".

Value with width 10, exactly 2 digits after decimal point; prints "[12.99]".

Locale-specific grouping using ','; prints "[123,456,789.123450]".

NOTE Because the formatting of the numbers is specific to your default locale, you might not see the same output as mentioned in the preceding code.

Though you can assign an `int` literal to a `float` or `double` variable (`float f = 10` or `long d = 10`), you can't use `int` variables or literal values with `%f`. The following code will throw an `IllegalFormatConversionException` runtime exception:

```
System.out.printf("[%,f]", 12345);
```

EXAM TIP By default, `%f` prints six digits after the decimal. It also rounds off the last digit. You can pass literal values or variables of type `float`, `double`, `Float`, and `Double` to the format specifier `%f`.

You can format the integers as follows:

```
System.out.printf("[%d]", 12345);
System.out.printf("[%010d]", 12345);
System.out.printf("%(,d", -123456789);
System.out.printf("[%-10.2d]", 12345);
```

Prints "[12345]".

Outputs value with width 10, zero padded; prints "[0000012345]".

Negative numbers enclosed within parentheses; prints "(-123,456,789)".

Throws java.util.IllegalFormatPrecisionException at runtime; can't specify precision with integers.

EXAM TIP You can pass literal values or variables of type `byte`, `short`, `int`, `long`, `Byte`, `Short`, `Integer`, or `Long` to the `%d` format specifier. The code throws runtime exceptions for all other types of values.

Also, the flags `+`, `0`, `(`, and `,` (comma) can be specified only with the numeric specifiers `%d` and `%f`. If you try to use them with any other format specifier (`%b`, `%s`, or `%c`), you'll get a runtime exception.

5.3.7 Formatting parameter %s

`%s` is a general-purpose format specifier that can be applied to both primitive variables and object references. For primitive variables, the value will be displayed; for object references, method `toString()` of the object is called:

```
String name = "Harry";
Integer age = null;
String[] skills = {"Java", "Android"};
System.out.format("Name is %s, age is %s, skills are %s", name,
age,skills);
```

In the preceding code, `format()` sends the following to the standard output (the exact integer value following `@` will vary on your system):

```
Name is Harry, age is null, skills are [Ljava.lang.String;@1d9dc39
```

EXAM TIP You can pass any type of primitive variable or object reference to the format specifier `%s`.

It's interesting to note how you can specify the `argument_index` to change the arguments used at a particular location. Let's specify the argument index in the preceding example to swap the variables `name` and `age`:

```
age = 40;
System.out.format("Name is %2$s, age is %1$s", name, age);
```

The output of the code is as follows:

```
Name is 40, age is Harry
```

EXAM TIP You can specify that the `argument_index` change the arguments used at a particular location.

5.4 Summary

String processing is a relatively big topic, so we started this chapter with an outline of what to expect on the exam. The exam covers searching, parsing, replacing, and formatting strings. It also covers formatting primitive data types and object references. We covered the classes that you need to know and their respective methods to work with this functionality. Classes `String` and `StringBuilder` are used to work with strings, to modify, search, and replace their values. Class `Scanner` is used to tokenize data and manipulate it.

When you search data, you can look for either exact matches or a pattern of data. A regular expression (or regex) is a powerful language that can be used to define data patterns to be searched for. I limited discussion of regex to the patterns included on the exam. I covered Java's regex support with the `java.util.regex` package. At the end of the chapter, you learned how to control and define the format of your data (primitive values and objects) by using class `Formatter` and its utility methods, in a locale-specific or custom format.

REVIEW NOTES

This section lists the main points covered in this chapter.

Regular expressions

- Regular expressions, or regex, are used to define patterns of data to be found in a stream.
- A regex has a syntax, which can be defined by using regular and special characters.
- As opposed to exact matches, you can use regex to search for data that matches a pattern.
- Character classes aren't classes defined in the Java API. The term refers to a set of characters that you can enclose within square brackets `[]`.
- Java supports predefined and custom character classes.

- You create a custom character class by enclosing a set of characters within square brackets []:
  - `[fdn]` can be used to find an exact match of `f`, `d`, or `n`.
  - `[^fdn]` can be used to find a character that doesn't match either `f`, `d`, or `n`.
  - `[a-cA-c]` can be used to find an exact match of either `a`, `b`, `c`, `A`, `B`, or `C`.
- You can use these predefined character classes as follows:
  - A dot matches any character (and may or may not match line terminators).
  - `\d` matches any digit: `[0-9]`.
  - `\D` matches a nondigit: `[^0-9]`.
  - `\s` matches a whitespace character: [(space), `\t` (tab), `\n` (new line), `\x0B` (end of line), `\f` (form feed), `\r` (carriage)]
  - `\S` matches a non-whitespace character: `[^\s]`.
  - `\w` matches a word character: `[a-zA-Z_0-9]`.
  - `\W` matches a nonword character: `[^\w]`.
- To use a regex pattern in Java code that includes a backslash, you must escape the \ by preceding it with another \. The character literal \ has a special meaning; it's used as an escape character. To use it as a literal, it must be escaped.
- For the exam, you'll need to know these boundary matchers:
  - `\b` indicates a word boundary.
  - `\B` indicates a nonword boundary.
  - `^` indicates the beginning of a line.
  - `$` indicates the end of a line.
- You can specify the number of occurrences of a pattern to match in a target value by using quantifiers.
- The coverage of quantifiers on this exam is limited to the following greedy quantifiers:
  - `X?` matches X, once or not at all.
  - `X*` matches X, zero or more times.
  - `X+` matches X, one or more times.
  - `X{min,max}` matches X, within the specified range.
- Regex in Java supports Unicode, as it matches against the `CharSequence` objects.
- Class `Pattern` is a compiled representation of a regular expression. It doesn't define a public constructor. You can instantiate this class by using its factory method `compile()`.
- Class `Matcher` is referred to as an engine that scans a target `CharSequence` for a matching regex pattern. Class `Matcher` doesn't define a public constructor. You can create and access a `Matcher` object by calling the instance method `matcher()` on an object of class `Pattern`.
- When you have access to the `Matcher` object, you can match a complete input sequence against a pattern, match the input sequence starting at the beginning,

find multiple occurrences of the matching pattern, or retrieve information about the matching groups.

Search, parse, and build strings

You can search strings for exact matches of characters or strings, at the beginning of a string, or starting at a specified position, using `String` class's overloaded methods `indexOf` (note the capital O).

- Method `indexOf()` returns the first matching position of a character or string, starting from the specified position of this string, or from its beginning.
- Method `lastIndexOf()` returns the last matching position of a character in the entire string, or its subset (position 0 to the specified position).
- Methods `indexOf()` and `lastIndexOf()` differ in the manner that they search a target string—`indexOf()` searches in increasing position numbers and `lastIndexOf()` searches backward. Due to this difference, `indexOf('a', -100)` will search the complete string, but `lastindexOf('a', -100)` won't. In a similar manner, because `lastIndexOf()` searches backwards, `lastIndexOf('a', 100)` will search the string, but `indexOf('a', 0)` or `indexOf('a', -100)` won't.
- Methods `indexOf()` and `lastIndexOf()` don't throw a compilation error or runtime exception if the search position is negative or greater than the length of this string. If no match is found, they return `-1`.
- Method `contains()` searches for an exact match in this string. Because `contains()` accepts a method parameter of interface `CharSequence`, you can pass to it both a `String` and a `StringBuilder` object.
- Methods `subSequence` (uppercase *S*) and `substring` (no uppercase letter) accept `int` parameters and return a substring of the target string.
- Method `substring()` defines overloaded versions, which accept one or two `int` method parameters to specify the start and end positions.
- Method `subSequence()` defines only one variant, the one that accepts two `int` method parameters for the start and end positions.
- Methods `subSequence()` and `substring()` don't include the character at the end position in the result `String`. Also, unlike methods `indexOf()` and `lastIndexOf()`, they throw the runtime exception `StringIndexOutOfBoundsException` for invalid start and end positions.
- The name of methods `subSequence()` and `substring()` can be used to determine their return type. `subSequence()` returns `CharSequence` and `substring()` returns `String`.
- Methods `split(String regex)` and `split(String regex, int limit)` in class `String` search for a matching regex pattern and split a `String` into an array of string values.
- The `String` array returned by `split()` doesn't contain the values that it matches to split the target string.

- You can limit the maximum number of tokens that you want to retrieve by using `split(String regex, int limit)`.
- `replace(char oldChar, char newChar)` returns a new `String` resulting from finding and replacing all occurrences of the `old` character with the `new` character.
- `replace(CharSequence oldVal, CharSequence newVal)` returns a new `String` resulting from finding and replacing each substring of the string that matches the `old` target sequence with the specified `new` replacement sequence.
- `replaceAll(String regex, String replacement)` replaces each substring of the string that matches the given regular expression with the given replacement.
- `replaceFirst(String regex, String replacement)` replaces the first substring of the string that matches the given regular expression with the given replacement.
- Unlike `replace()`, `replaceAll()` doesn't accept method parameters of type `CharSequence`. Watch out for the passing of objects of class `StringBuilder` to `replaceAll()`.
- The combination of the `replace`, `replaceAll`, and `replaceFirst` overloaded methods can be confusing on the exam. Take note of the method parameters that can be passed to each of these methods.
- `Scanner` can be used to parse and tokenize strings.
  - If no delimiter is specified, a pattern that matches whitespace is used by default for a `Scanner` object.
  - You can specify a custom delimiter by calling its method `useDelimiter()` with a regex.
  - Method `next()` returns an object of type `String`.
  - `Scanner` also defines multiple `nextXXX` methods, where `XXX` refers to a primitive data type. These methods return the value as the corresponding primitive type.
  - Methods `hasNext()` and `hasNextXxx()` only return `true` or `false` but don't advance. Only methods `next()` and `nextXxx()` advance in the input.
  - Method `findInLine()` matches the specified pattern with no regard to delimiters in the input.

Formatting strings

- Class `java.util.Formatter` is an interpreter for `printf`-style format strings.
- A formatter provides support for layout justification and alignment; common formats for numeric, string, and date/time data; and locale-specific output.
- `Formatter`'s `format()` is used to format data.
- To use `format()`, you need to define a format string that defines how to format text and an object argument list that defines what to format.
- You can define a combination of fixed text and one or more embedded format specifiers, to be passed to the method's `format()` first argument.

- The format specifier takes the following form:

```
%[argument_index$][flags][width][.precision]conversion
```

- A format specification must start with a % sign and end with a conversion character:
  - `b` for `boolean`
  - `c` for `char`
  - `d` for `int`, `byte`, `short`, and `long`
  - `f` for `float` and `double`
  - `s` for `String`
- If the number of arguments exceeds the required count, the extra variables are quietly ignored by the compiler and JVM. But if the number of required arguments falls short, the JVM throws a runtime exception.
- The `-` indicates to left-justify this argument; you must specify width as well. Number flags (only applicable for numbers, conversion chars `d` and `f`) are as follows:
  - The `+` indicates to include a sign (`+` or `-`) with this argument.
  - `0` indicates to pad this argument with zeros. Must specify width as well.
  - `,` indicates to use locale-specific grouping separators (for example, the comma in 123,456).
  - `(` is used to enclose negative numbers in parentheses.
- The flags `+`, `0`, `(`, and `,` can be specified only with the numeric specifiers `%d` and `%f`. If you try to use them with any other format specifier (`%b`, `%s`, or `%c`), you'll get a runtime exception.
- Format specifier `%b`
  - You can pass any type of primitive variable or object reference to specifier `%b`.
  - If the target argument `arg` is `null`, then `%b` outputs the result as `false`. If `arg` is `boolean` or `Boolean`, the result is the `String` returned by `String .valueOf()`. Otherwise, the result is `true`.
- Format specifier `%c`
  - `%c` outputs the result as a Unicode character.
  - You can pass only literals and variables that can be converted to a Unicode character (`char`, `byte`, `short`, `int`, `Character`, `Byte`, `Short`, and `Integer`) to the `%c` specifier. Passing variables of type `boolean`, `long`, `float`, `Boolean`, `Long`, `Float`, or any other class will throw `IllegalFormatConversionException`.
- Format specifier `%f`
  - You can format decimal numbers (`float`, `Float`, `double`, and `Double`) by using the format specifier `%f`.
  - By default, `%f` prints six digits after the decimal. It also rounds off the last digit.
- Format specifier `%d`
  - You can format integers (`byte`, `short`, `int`, `long`, `Byte`, `Short`, `Integer`, `Long`) by using the format specifier `%d`.
  - If you pass literal values or variables of type `float`, `double`, `Float`, or `Double` to the format specifier `%d`, the code will throw a runtime exception.

- Format specifier `%s`
  - `%s` outputs the value for a primitive variable. For reference variables, it calls `toString()` on objects that are not `null` and outputs `null` for `null` values.
  - You can pass any type of primitive variable or object reference to specifier `%s`.

SAMPLE EXAM QUESTIONS

Q 5-1. What is the output of the following code?

```
class Format1 {
    public static void main(String... args) {
        double num1 = 7.12345678;
        int num2 = (int)8.12345678;
        System.out.printf("num1=%f, num2=%2d, %b", num1, num2, num2);
    }
}
```

a. `num1=7.123456, num2= 8, true`
b. `num1=7.123456, num2=8, true`
c. `num1=7.123457, num2= 8, true`
d. `num1=7.123457, num2=8 , true`
e. `num1=7.1234, num2=8, false`
f. `num1=7.1234, num2=8.1234, true`
g. Compilation error
h. Runtime exception

Q 5-2. Given the following command line

```
java Regex1 \d\d 761cars8 5dogs-total846
```

what is the output of the following code?

```
class Regex1 {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile(args[0]);
        Matcher matcher = pattern.matcher(args[1]);
        boolean found = false;
        while(found = matcher.find()) {
            System.out.println(matcher.group());
        }
    }
}
```

a.
```
76
61
```
b.
```
76
```

c
```
76
61
84
46
```

d
```
76
84
```

e No output

Q 5-3. Given the following variables, which options will throw exceptions at runtime?

```
String eJava = "Guru";
Integer start = 100;
boolean win = true;
Float duration = new Float(-1099.9999);
```

a `System.out.format("%d", eJava);`

b `System.out.printf("%s", start);`

c `System.out.printf("[%-12b]", win);`

d `System.out.format("%s12", eJava);`

e `System.out.format("%d", duration);`

f `System.out.format("[%+,-(20f]", duration);`

Q 5-4. What is the output of the following code?

```
Scanner scanner = new Scanner("ThemeXtheirXcarpet77");
scanner.useDelimiter("t.*e");
while (scanner.hasNext())
    System.out.println (scanner.next());
```

a
```
ThemeX
the
t77
```

b
```
ThemeX
t77
```

c
```
The
the
```

d
```
t77
```

e Compilation error

f Runtime exception

Q 5-5. Which options will output the following code?

```
Hello true 123456
```

a
```
System.out.print("%s %b %d", new StringBuilder("Hello"), "false",
123456);
```

b `System.out.printf("%s %b %d", new String("Hello"), "false", 123456);`
c `System.out.format("%s %b %d", new StringBuilder("Hello"), "false", 123456);`
d `System.out.println("%s %b %d", new StringBuilder("Hello"), "false", 123456.70);`
e `System.out.printf("%s %b %d", new StringTokenizer("Hello"), "false", 123456);`
f `System.out.format("%s %b %d", ("Hello"), "FALSE", new Integer(123456));`

Q 5-6. What is the output of the following code?

```
class StringCompare {
    public static void main(String[] args) {
        String mgr1 = "Paul & Harry's new office";
        StringBuilder emp = new StringBuilder("Harry");
        System.out.println(mgr1.contains(emp));
    }
}
```

a `true`
b `false`
c Compilation error
d Runtime exception

NOTE This question includes reading from a file that's covered in chapter 7. In the exam it's usual to see questions that are based on multiple exam topics.

Q 5-7. Given that the content of the file data.txt is

```
Harry;8765,Per[fect
```

which options are correct for the following code?

```
public class StrToken {
    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new FileReader("data.txt"));
        String line;
        StringTokenizer st;
        while ((line = br.readLine()) != null) {
            st = new StringTokenizer(line, "[,;]");          //line1
            while (st.hasMoreElements())
            System.out.println(st.nextElement());
        }
        br.close();
    }
}
```

a The code prints

```
Harry
8765
Per
fect
```

b The code prints

```
Harry
8765
Per[fect
```

c The code prints

```
Harry;8765
Per[fect
```

d If the code on line `//line1` is changed to the following, it'll output the same results:

```
st = new StringTokenizer(line, "[,;$]");          //line1
```

e Code fails to compile.

f Code throws a runtime exception.

Q 5-8. What is the output of the following code?

```
String eJava = "e Java Guru";
if(eJava.matches("u.u")){
    String[] tokens = eJava.split("\\Bu");
    for (String token : tokens) System.out.println(token);
}
else {
    System.out.println(eJava.replace(
                          eJava.subSequence(3, 4), eJava.substring(11)));
}
```

a `e Java G`
`r`

b `e Java Guru`

c `e Jv Guru`

d `e Juvu Guru`

e `java.lang.StringIndexOutOfBoundsException: String index out of range: 11`

f Code fails to compile.

Q 5-9. Which option, when used as a format specifier, will format the decimal literal value 14562975.6543 to use at least the locale-specific grouping separator and include a sign with it (`+` or `-`)? (Choose all that apply.)

a `%+,3f`

b `%,+20f`

c `%+,f`
d `%,+f`
e `%()f`
f `%-,f`
g `%0.+f`

Q 5-10. Given the following string, which code options use the correct regex to replace the first letter of all words with *A*, leave the remaining string unchanged, and print the replaced string value?

```
String target = "amita, matinda,shreya,mike, and anthony are arrogant";
```

a `System.out.println(target.subSequence("[\\b]\\w", "A"));`
b `System.out.println(target.replace("\\b\\w", "A"));`
c `System.out.println(target.replaceAll("\\s\\b\\w", "A"));`
d `System.out.println(target.replace("\\s\\b\\w", "A"));`
e `System.out.println(target.modify("\\b\\w", "A"));`
f `System.out.println(target.replaceAll("\\b\\w", "A"));`
g None of the above

ANSWERS TO SAMPLE EXAM QUESTIONS

A 5-1. c

[5.3] Format strings using the formatting parameters: %b, %c, %d, %f, and %s in format strings

Explanation: By default, `%f` prints out six digits after the decimal number. It also rounds off the last digit. So `num1=%f` outputs `7.123457` and not `7.123456`.

Because the double literal 8.12345678 is explicitly casted to an `int` value, `num2` contains the integer part of the double literal 8.12345678, that is, 8. `%2d` sets the total width of the output to 2 digits, padded with spaces and right-aligned by default. It outputs a space preceding the digit 8.

For all non-Boolean primitive values, `%b` outputs `true`.

A 5-2. b

[5.2] Search, parse, and replace strings by using regular expressions, using expression patterns for matching limited to: . (dot), * (star), + (plus), ?, \d, \D, \s, \S, \w, \W, \b, \B, [], ()

Explanation: The last argument (`5dogs-total846`) is ignored when you use the following command line because a space precedes it.

```
java -ea Regex1 \d\d 761cars8 5dogs-total846
```

When you pass the regex by using the command-line arguments, you don't need to escape the backslashes. It's required only for literal string values.

`\d\d` will match two adjacent digits in the literal `761cars8`—that is, `76`. It won't match 61 because the digit 6 was already consumed in finding 76. By default, Java's regex engine won't use characters that have already been consumed.

A 5-3. a, e

[5.3] Format strings using the formatting parameters: %b, %c, %d, %f, and %s in format strings

Explanation: You'll get runtime exceptions if you use either of the following for a format specifier:

- An invalid data type
- An invalid combination of flags

Options (a) and (e) throw runtime exceptions because they use the format specifier `%d`, which must be used only with integer literal values or variables, and not with `String` or decimal numbers.

Option (b) won't throw any exception because you can pass any type of primitive variable or object reference to the format specifier `%s`. Also, this option doesn't use any format flag that's invalid to be used with `%s`.

Option (c) won't throw any exceptions. You can specify the alignment and width element `-12` with `%b`.

Option (d) won't throw any exceptions. The value `12` that follows `%s` is not part of the width element, but a literal value following `%s`, so it's a completely valid format string.

A 5-4. b

[5.1] Search, parse, and build strings (including Scanner, StringTokenizer, StringBuilder, String, and Formatter)
[5.2] Search, parse, and replace strings by using regular expressions, using expression patterns for matching limited to: . (dot), * (star), + (plus), ?, \d, \D, \s, \S, \w, \W, \b, \B, [], ().

Explanation: The code compiles successfully and throws no exceptions at runtime. So options (e) and (f) are incorrect.

In the regex `t.*e`, the metacharacter `.` matches any character, and `*` is a greedy quantifier. `t.*e` searches for the letter *t* and scans all the remaining letters to find the last occurrence of the letter *e*. The string `theirXcarpe` matches this regex pattern. So `theirXcarpe` is used as a delimiter to the tokenizer `ThemeXtheirXcarpet77`. Text preceding and following this delimiter is returned as tokens.

A 5-5. b, c, f

[5.3] Format strings using the formatting parameters: %b, %c, %d, %f, and %s in format strings.

Explanation: Options (a) and (d) won't compile. The type of the variable `out` in class `System` is `PrintStream`. `PrintStream`'s methods `print()` and `println()` accept just one method parameter. They're overloaded to accept parameters of all the primitive types and object references. But, they don't accept format strings and arguments.

Option (b) is correct. The format specifier `%b` returns the value `true` for non-null values for object references other than `Boolean` or `boolean`.

Option (c) is correct. The format specifier `%s` calls method `toString()` on an object reference to get its `String` representation. Method `toString()` of class `StringBuilder` creates and returns a new `String` by using the sequence stored by the `StringBuilder`. So printing a `StringBuilder` doesn't print a value similar to `StringBuilder@135d`.

Option (e) is incorrect. `%s` calls `StringTokenizer`'s method `toString()` to access its `String` representation. This option outputs a value similar to this:

```
java.util.StringTokenizer@750159 true 123456
```

Option (f) is correct. Methods `printf()` and `format()` provide exactly the same functionality. Behind the scenes, `printf()` calls `format()`.

A 5-6. a

[5.1] Search, parse, and build strings (including Scanner, StringTokenizer, StringBuilder, String, and Formatter)

Explanation: Method `contains()` accepts a method parameter of type `CharSequence`, and so an object of class `StringBuilder` is acceptable (class `StringBuilder` implements the interface `CharSequence`). Method `contains()` returns `true`, if the specified `String` equivalent can be found in the `String` object on which this method is called.

A 5-7. a, d

[5.1] Search, parse, and build strings (including Scanner, StringTokenizer, StringBuilder, String, and Formatter)
[5.2] Search, parse, and replace strings by using regular expressions, using expression patterns for matching limited to: . (dot), * (star), + (plus), ?, \d, \D, \s, \S, \w, \W, \b, \B, [], ().

Explanation: On the exam, you're likely to see questions based on multiple exam objectives, just as this question covers file handling and string processing.

Class `StringTokenizer` doesn't accept the delimiter as a regex pattern. When `[,;]` is passed as a delimiter to class `StringTokenizer`, the occurrence of either of these characters acts as a delimiter. So the line read from file data.txt is delimited using `,` `;` and `[`.

Because the text in file data.txt doesn't include `$`, changing the delimiter text from `[,;]` to `[,;$]` won't affect the output.

A 5-8. c

[5.1] Search, parse, and build strings (including Scanner, StringTokenizer, StringBuilder, String, and Formatter)
[5.2] Search, parse, and replace strings by using regular expressions, using expression patterns for matching limited to: . (dot), * (star), + (plus), ?, \d, \D, \s, \S, \w, \W, \b, \B, [], ().

Explanation: Method `matches()` doesn't look for a matching subsequence. It matches the complete string value against the given regex pattern. Because the regex pattern `u.u` matches a subsequence and not the entire string value `e Java Guru`, it returns `false`.

The length of string `eJava` is 11, so `eJava.substring(11)` doesn't throw `StringIndexOutOfBoundsException`. It returns an empty string, which replaces string `"a` in string value `e Java Guru`, resulting in `e Jv Guru`.

A 5-9. a, b, c, d

[5.3] Format strings using the formatting parameters: %b, %c, %d, %f, and %s in format strings.

Explanation: You must use the following for the required format:

- `%f`
- Flag `+` to include a sign
- Flag `,` to use locale-specific grouping

Options (a), (b), (c), and (d) use the correct format specifier, `%f`. The flags `+` and `,` can appear in any order. It's acceptable to specify the width of the numeric literal.

If the total width specified with the format specifier is less than the width of the argument passed to it, it's ignored. If the width specified after the decimal is less than the number of digits stored by the value to be formatted, it's rounded off.

Options (e), (f), and (g) are incorrect. They either include an invalid combination of flags or don't include the required flags.

A 5-10. f

[5.1] Search, parse, and build strings (including Scanner, StringTokenizer, StringBuilder, String, and Formatter)
[5.2] Search, parse, and replace strings by using regular expressions, using expression patterns for matching limited to: . (dot), * (star), + (plus), ?, \d, \D, \s, \S, \w, \W, \b, \B, [], ().

Explanation: Option (a) is incorrect because it doesn't compile. You can only pass integer values as parameters, not strings. Method `subSequence()` doesn't replace string values. It extracts and returns a matching subsequence of a string.

Options (b) and (d) are incorrect because `replace()` doesn't accept a regex pattern. It replaces exact matches of a string value.

Option (c) is incorrect—not all strings have a space in front of the first letter, so the `replace()` option isn't performed on every word in the string. `\\s` will also match and replace the spaces (or whitespace, but not a combination of both).

Option (e) is incorrect because `modify()` doesn't exist.

Option (f) is correct. `replaceAll()` can be passed a regex and its replacement. All matching regex values in the string are replaced with the specified replacement string. The regex pattern `\b\w` matches a word boundary (`\b`) followed by a word character (`\w`)— that is, `a-zA-Z_0-9`.