3

Manipulating Strings

Much of what you do in any programming language involves the manipulation of strings. Other than numeric data, nearly all data is accessed as a string. Quite often, even numeric data is treated as a simple string. It is difficult to imagine being able to write a complete program without making use of strings.

The phrases in this chapter show you some common tasks involving strings. The Java language has strong built-in support for strings and string processing. Unlike the C language, strings are built-in types in the Java language. Java contains a `String` class that is used to hold string data. Strings in Java should not be thought of as an array of characters as they are in C. Whenever you want to represent a string in Java, you should use the `String` class, not an array.

An important property of the `String` class in Java is that once created, the string is immutable. This means that once created, a Java `String` object cannot be changed. You can reassign the name you've given a string to another string object, but you cannot change

the string's contents. Because of this, you will not find any set methods in the `String` class. If you want to create a string that you can add data to, such as you might in some routine that builds up a string, you should use the `StringBuilder` class if you are using JDK 1.5, or the `StringBuffer` class in older versions of Java, instead of the `String` class. The `StringBuilder` and `StringBuffer` classes are mutable; thus you are allowed to change their contents. It is very common to build strings using the `StringBuilder` or `StringBuffer` class and to pass or store strings using the `String` class.

Comparing Strings

```
boolean result = str1.equals(str2);
boolean result2 = str1.equalsIgnoreCase(str2);
```

The value of `result` and `result2` will be true if the strings contain the same content. If the strings contain different content, the value of `result` and `result2` will be false. The first method, `equals()`, is case sensitive. The second method, `equalsIgnoreCase()`, will ignore the case of the strings and return true if the content is the same regardless of case.

String comparison is a common source of bugs for novice Java programmers. A novice programmer will often attempt to compare strings using the comparison operator ==. When used with Strings, the comparison operator == compares object references, not the contents of the object. Because of this, two string objects that contain the same string data, but are physically distinct string object instances, will not compare as equal when using the comparison operator.

The `equals()` method on the `String` class compares a string's contents, rather than its object reference. This is the preferred string comparison behavior in most string comparison cases. See the following example:

```
String name1 = new String("Timmy");
String name2 = new String("Timmy");
if (name1 == name2) {
     System.out.println("The strings are equal.");
}
else {
     System.out.println("The strings are not
equal.");
}
```

The output from executing these statements will be

```
The strings are not equal.
```

Now use the `equals()` method and see the results:

```
String name1 = new String("Timmy");
String name2 = new String("Timmy");
if (name1.equals(name2)) {
     System.out.println("The strings are equal.");
}
else {
     System.out.println("The strings are not
equal.");
}
```

The output from executing these statements will be

```
The strings are equal.
```

Another related method on the `String` class is the `compareTo()` method. The `compareTo()` method compares two strings lexographically, returning an integer value—either positive, negative, or `0`. The value `0` is

returned only if the `equals()` method would evaluate to `true` for the two strings. A negative value is returned if the string on which the method is called alphabetically precedes the string that is passed as a parameter to the method. A positive value is returned if the string on which the method is called alphabetically comes after the string that is passed as a parameter. To be precise, the comparison is based on the Unicode value of each character in the strings being compared. The `compareTo()` method also has a corresponding `compareToIgnoreCase()` method that performs functionally the same with the exception that the characters' case is ignored. See the following example:

```
String name1="Camden";
String name2="Kerry";
int result = name1.compareTo(name2);
if (result == 0) {
     System.out.println("The names are equal.");
}
else if (result > 0) {
     System.out.println(
         "name2 comes before name1 alphabeti-
cally.");
}
else if (result < 0) {
     System.out.println(
        "name1 comes before name2 alphabetically.");
}
```

The output of this code will be

```
name1 comes before name2 alphabetically.
```

Searching For and Retrieving Substrings

```
int result = string1.indexOf(string2);
int result = string1.indexOf(string2, 5);
```

In the first method shown, the value of `result` will contain the index of the first occurrence of `string2` within `string1`. If `string2` is not contained within `string1`, `-1` will be returned.

In the second method shown, the value of `result` will contain the index of the first occurrence of `string2` within `string1` that occurs after the fifth character within `string1`. The second parameter can be any valid integer greater than `0`. If the value is greater than the length of `string1`, a result of `-1` will be returned.

Besides searching a string for a substring, there might be times when you know where a substring is that you are interested in and simply want to get at that substring. So, in either case, you now know where a substring is that you are interested in. Using the String's `substring()` method, you can now get that substring. The `substring()` method is overloaded, meaning that there are multiple ways of calling it. One way of calling it is to pass a start index. This will return a substring that begins at the start index and extends through the end of the string. The other way of using `substring()` is to call it with two parameters—a start index, and an end index.

```
String string1 = "My address is 555 Big Tree Lane";
String address = string1.substring(14);
System.out.println(address);
```

This code will print out

```
555 Big Tree Lane
```

The first `5` character is at position 14 in the string; thus it is the beginning of the substring. Note that strings are always zero-based indexed, and the last character of a string is at location (length of string) `-1`.

Processing a String One Character at a Time

```
for (int index = 0; index < string1.length();
index++) {
     char aChar = string1.charAt(index);
}
```

The `charAt()` method allows you to obtain a single character from the string at the specified index. The characters are indexed 0 based, from 0 to the length of the string-1. The phrase shown previously loops through each character contained in `string1`.

An alternative method would be to use the `StringReader` class, as follows:

```
StringReader reader = new StringReader(string1);
int singleChar = reader.read();
```

Using this mechanism, the `read()` method of the `StringReader` class returns one character at a time, as an integer. Each time the `read()` method is called, the next character of the string will be returned.

Reversing a String by Character

```
String letters = "ABCDEF";
StringBuffer lettersBuff = new StringBuffer(let-
ters);
String lettersRev =
lettersBuff.reverse().toString();
```

The `StringBuffer` class contains a `reverse()` method that returns a `StringBuffer` that contains the characters from the original `StringBuffer` reversed. A `StringBuffer` is easily converted into a `String` using the `toString()` method of the `StringBuffer`. So by temporarily making use of a `StringBuffer`, you are able to produce a second string with the characters of an original string in reverse order.

If you are using JDK 1.5, you can use the `StringBuilder` class instead of the `StringBuffer` class. The `StringBuilder` class has an API compatible with the `StringBuffer` class. The `StringBuilder` class will give you faster performance, but its methods are not synchronized; thus it is not thread-safe. In multithreaded situations, you should continue to use the `StringBuffer` class.

Reversing a String by Word

```
String test = "Reverse this string";
Stack  stack = new Stack();
StringTokenizer strTok = new StringTokenizer(test);

while(strTok.hasMoreTokens()) {
    stack.push(strTok.nextElement());
}
```

```
StringBuffer revStr = new StringBuffer();
while(!stack.empty()) {
    revStr.append(stack.pop());
    revStr.append(" ");
}
System.out.println("Original string: " + test);
System.out.println("\nReversed string: " + revStr);
```

The output of this code fragment will be

```
Original string: Reverse this string
Reversed string: string this Reverse
```

As you can see, reversing a string by word is more complex than reversing a string by character. This is because there was built-in support for reversing a string by character, but there is no such built-in support for reversing by word. To accomplish this task, we make use of the `StringTokenizer` and the `Stack` classes. Using `StringTokenizer`, we parse each word out of the string and push it onto our stack. After we've processed the entire string, we iterate through the stack, popping each word off and appending to a string buffer that holds the reversed string. A stack that has the property of last item in becomes the first item out. Because of this property, the stack is often referred to as a LIFO (last in, first out) queue. This makes the reverse successful.

See the phrase covered in the section, "Parsing a Comma-Separated String" in this chapter for more uses of the `StringTokenizer` class.

NOTE: I don't cover it here, but you may also be interested in checking out a new addition to JDK 1.5, the Scanner class. The Scanner class is a simple text scanner which can parse primitive types and strings using regular expressions.

Making a String All Uppercase or All Lowercase

```
String string = "Contains some Upper and some
Lower.";
String string2 = string.toUpperCase();
String string3 = string.toLowerCase();
```

These two methods transform a string into all uppercase or all lowercase letters. They both return the transformed result. These methods do not change the original string. The original string remains intact with mixed case.

A practical area in which these methods are useful is when storing information in a database. There might be certain fields that you always want to store as all uppercase or all lowercase. These methods make the conversion a snap.

Case conversion is also useful for processing user logins. The user ID field is normally considered to be a field that's not case sensitive, whereas the password field is case sensitive. So, when comparing the user ID, you should convert to a known case and then compare to a stored value. Alternatively, you can always use the `equalsIgnoreCase()` method of the `String` class, which performs a non case sensitive comparison.

Trimming Spaces from the Beginning or End of a String

```
String result = str.trim();
```

The `trim()` method will remove both leading and trailing whitespace from a string and return the result.

The original string will remain unchanged. If there is no leading or trailing whitespace to be removed, the original string is returned. Both spaces and tab characters will be removed.

This is very useful when comparing user input with existing data. A programmer often racks his brain for hours trying to figure out why what he enters is not the same as a stored string, only to find out that the difference is only a trailing space. Trimming data prior to comparison will eliminate this problem.

Parsing a Comma-Separated String

```
String str = "tim,kerry,timmy,camden";
String[] results = str.split(",");
```

The `split()` method on the `String` class accepts a regular expression as its only parameter, and will return an array of `String` objects split according to the rules of the passed-in regular expression. This makes parsing a comma-separated string an easy task. In this phrase, we simply pass a comma into the `split()` method, and we get back an array of strings containing the comma-separated data. So the results array in our phrase would contain the following content:

```
results[0] = tim
results[1] = kerry
results[2] = timmy
results[3] = camden
```

Another useful class for taking apart strings is the `StringTokenizer` class. We will repeat the phrase using

the `StringTokenizer` class instead of the `split()` method.

```
String str = "tim,kerry,timmy,Camden";
StringTokenizer st = new StringTokenizer(str, ",");
while (st.hasMoreTokens()) {
     System.out.println(st.nextToken());
}
```

This code example will print each of the names contained in the original string, `str`, on a separate line, as follows:

```
tim
kerry
timmy
camden
```

Notice that the commas are discarded and not output.

The `StringTokenizer` class can be constructed with one, two, or three parameters. If called with just one parameter, the parameter is the string that you want to tokenize, or split up. In this case, the delimiter is defaulted to natural word boundaries. The tokenizer uses the default delimiter set, which is `" \t\n\r\f"`: the space character, the tab character, the newline character, the carriage-return character, and the form-feed character.

The second way of constructing a `StringTokenizer` object is to pass two parameters to the constructor. The first parameter is the string to be tokenized, and the second parameter is a string containing the delimiters that you want to split the string on. This overrides the default delimiters and sets them to whatever you pass in the second argument.

Finally, you can pass a third argument to the `StringTokenizer` constructor that designates whether

delimiters should be returned as tokens or discarded. This is a Boolean parameter. A value of true passed here will cause the delimiters to be returned as tokens. False is the default value, which discards the delimiters and does not treat them as tokens.

You should also review the phrases in Chapter 6. With the addition of regular expression support to Java in JDK1.4, many of the uses of the `StringTokenizer` class can be replaced with regular expressions. The official JavaDoc states that the `StringTokenizer` class is a legacy class and its use should be discouraged in new code. Wherever possible, you should use the `split()` method of the `String` class or the regular expression package.