Chapter 8

String Theory

A great deal of the data processed by computer programs is represented as text. Word processors, email programs, and chat clients are primarily concerned with manipulating text. Most web pages contain plenty of text in addition to media of other forms. In addition, we have seen that the communication protocols upon which many network applications rely are based on sending text messages back and forth between client and server computers.

Up to this point, we have seen only a small portion of the mechanisms Java provides for processing text. We have learned that pieces of text can be described in our programs using quoted literals and associated with variable names as objects of the type `String`. We have seen that such objects can be used as parameters in constructions and invocations associated with many other types. We have, however, seen very few operations designed to process the `String` values themselves. The notable exception is that we have seen that the concatenation operator, `+`, can be used to combine the contents of two `String`s to form a longer `String`.

In fact, Java provides an extensive collection of methods for manipulating `String`s. In this chapter we will introduce some of the most important of these methods. We will learn how to access sub-parts of `String`s, how to search the text of a `String` for particular subsequences, and how to make decisions by comparing `String` values with one another. We will deliberately avoid trying to cover all of the `String` operations that Java provides. There are far too many. We will limit our attention to the most essential methods. We will, however, explain how you can access the online documentation for the standard Java libraries so that you can learn about additional methods you can use with `String`s.

8.1 Cut and Paste

Anyone who has used a computer is probably familiar with "cut and paste." This phrase refers to the paradigm that is used by almost all programs that support text editing with a graphical user interface. Such programs let you extract a fragment of a larger text document (i.e., cut) and then insert a copy of the fragment elsewhere in the document (i.e., paste). Not surprisingly, two of the most important operations you can perform on text within a program are essentially "cutting" and "pasting."

The ability to paste text together within a Java program is provided by the concatenation operator, `+`. As we have seen, if `prefix` and `suffix` are two `String` variables, then the expression

```
prefix + suffix
```

describes the `String` obtained by pasting their contents together. For example, if we set

```
suffix = "ion field"
```

and

```
prefix = "enter the construct"
```

then the expression `prefix + suffix` will describe the `String`

```
"enter the construction field"
```

A fragment can be extracted (i.e., cut) from a `String` using a method named `substring`. Actually, the `substring` method is a bit more akin to an editor's "copy" operation than the standard "cut" operation. It allows a program to extract a copy of a subsection of a `String` but leaves the source `String` unchanged.

When you cut or copy text within a word processor or text editor, you select the desired text using the mouse. When you are using the `substring` method, however, you must specify the fragment of text you want to extract using the arguments you provide in the method invocation. This is done by providing an `int` parameter identifying the first character of the desired fragment and, optionally, an `int` identifying the first character following the fragment.

The parameter values provided to the `substring` method specify the number of characters that come before the character being identified. That is, a parameter value of 0 describes the first character in a `String` since there are 0 characters before this character and a value of 1 describes what you would normally call the second character. Therfore, if we declared and initialized a string variable

```
String bigword = "thermoregulation";
```

then the expression

```
bigword.substring( 1, 4 )
```

would produce the result `"her"`, while the expression

```
bigword.substring( 0, 3 )
```

would produce the result `"the"`.

It is important to note that the second parameter of the `substring` method indicates the position of the first character that should **not** be included in the result, rather than the position of the last character that should be included. Thus, assuming that the variable `bigword` was declared and initialized as shown above, the value described by the expression

```
bigword.substring( 1, 2 )
```

would be `"h"`, since this invocation asks for all the characters from position 1 in `bigword` up to but not including position 2. In general, an expression of the form

```
bigword.substring( p, p+n )
```

returns a `String` containing the n characters found starting at position `p` within `bigword`. In particular, an expression of the form

```
bigword.substring( p, p )
```

describes the `String` containing **zero** characters, `""`.

Java provides a method named `length` that returns the number of characters found within a `String`. For example,

```
bigword.length()
```

would return the value 16. As a result, an invocation of the form

```
bigword.substring( p, bigword.length() )
```

describes all the characters from position `p` within `bigword` through the last character in the `String`. Using `substring` to extract a suffix from a `String` in this way is common enough that Java provides an easier way to do it. The second parameter to the `substring` method is optional. When `substring` is invoked with just one parameter, it acts as if the length of the `String` was provided as the second parameter. That is,

```
bigword.substring( p )
```

is equivalent to

```
bigword.substring( p, bigword.length() )
```

For example,

```
bigword.substring( 13 )
```

would return the value `"ion"`.

The values provided as parameters to the `substring` method should always fall between 0 and the length of the `String` to which the method is being applied. If two parameters are provided, then the value of the first parameter must not be greater than that of the second. For example, the invocation

```
bigword.substring( 11, 20 )
```

would cause a run-time error since `bigword` only contains 16 characters. The invocation

```
bigword.substring( 10, 5 )
```

would produce an error since the starting position is greater than the ending position, and

```
bigword.substring( -1, 10 )
```

would cause an error since -1 is out of the allowed range for positions within a `String`.

The `substring` method does not modify the contents of the `String` to which it is applied. It merely returns a portion of that `String`. Thus, still assuming that `bigword` is associated with the `String` `"thermoregulation"`, if we were to declare a local variable

```
String smallword;
```

and then execute the statement

```
smallword = bigword.substring( 5, 8 );
```

the value associated with `smallword` would be `"ore"`, but the value associated with `bigword` would remain `"thermoregulation"`. While it would not make much sense to do so, we could even execute the statement several times in a row as in

```
smallword = bigword.substring( 5, 8 );
smallword = bigword.substring( 5, 8 );
smallword = bigword.substring( 5, 8 );
```

The final result would still be the same. Since the value associated with `bigword` is not changed, the final value associated with `smallword` would still be `"ore"`.

In fact, none of the methods presented in this chapter actually modify the `String` values to which they are applied. Instead, they return new, distinct values. The only way to change the `String` associated with a variable name is to assign a new value to the name. For example, if we wanted to make `bigword` refer to a substring of `"thermoregulation"` we might say

```
bigword = bigword.substring( 6 );
```

After this statement was executed, `bigword` would be associated with the `String` `"regulation"`. Unlike the assignment to `smallword` discussed in the preceding paragraph, the result of executing this assignment several times will not be the same as executing the assignment just once. Executing

```
bigword = bigword.substring( 6 );
```

a second time will leave `bigword` associated with the `String` `"tion"` since `t` appears in position 6 (i.e., it is the seventh letter) in `"regulation"`. A third execution would result in an error since there are less than 6 characters in the `String` `"tion"`.

In general, executing the statement

```
bigword = bigword.substring( p );
```

will effectively remove a prefix of length `p` from the value associated with the variable `bigword`. Similarly, executing the statement

```
bigword = bigword.substring( 0, p );
```

will remove a suffix from `bigword` so that only a prefix of length `p` remains. If, on the other hand, we want to remove a substring from the middle of a `String` value, the `substring` method alone is not enough. To do this, we have to use `substring` and concatenation together.

Suppose for example, that we want to turn `"thermoregulation"` into `"thermion"`.[1] The expression

```
bigword.substring( 0, 5 )
```

describes the `String` `"therm"`, and, as we showed above,

```
bigword.substring( 13 )
```

describes the `String` `"ion"`. Therefore, the assignment

```
bigword = bigword.substring( 0, 5 ) + bigword.substring( 13 );
```

will associate `bigword` with the value `"thermion"`. In general, executing an assignment of the form

```
bigword = bigword.substring( 0, start ) + bigword.substring( end );
```

will associate `bigword` with the `String` obtained from its previous value by removing the characters from position `start` up to but not including position `end`.

[1] "Thermion" is actually a word! It refers to an ion released by a material at a high temperature.

8.2 Search Options

The `substring` method can be very handy when writing programs that support communications through the Internet. Many of the Internet's protocols depend on `String`s. Web page addresses, email addresses, and IM screen names are all `String`s. To interpret these and other `String`s, programs often need the ability to access the subparts of a `String`. To apply the `substring` method effectively in such applications, however, we need a way to find the positions of the subparts of interest. In this section, we will introduce a method named `indexOf` which makes it possible to do this.

For example, if you want to visit a web page and know its address, you can enter this address in a form known as a URL (Uniform Resource Locator) or URI (Uniform Resource Identifier). For example, if you wanted to read the editorial section of the New York Times online, you could enter the URL

```
http://www.nytimes.com/pages/opinion/index.html
```

An address of this form has two main parts. The text between the pair of slashes and the first single slash, `"www.nytimes.com"`, is the name of a computer on the Internet that acts as the web server for the New York Times. The characters that appear after this machine name `"/pages/opinion/index.html"` describe a file on that server that contains a description of the contents of the editorial page. In particular, it indicates that the desired file is named `index.html` and that it can be found within a directory or folder named `opinion` within a directory named `pages`. Such a complete specification of the location of a file is called a *file path name*.

In order to display the contents of a web page, a web browser requests that the server named in the page's URL send it the contents of the file named by the text following the server name in the URL. The first step in making such a request, is to create a TCP connection to port 80 on the web server.

To imagine how the code within a web browser might actually accomplish this, suppose that the URL shown above was associated with a local variable declared as

```
String url = "http://www.nytimes.com/pages/opinion/index.html";
```

The `substring` method could then be used to extract either the server name or the file path name from the URL. For example, the expression

```
url.substring( 7, 22 )
```

would produce the server name `"www.nytimes.com"` as a result. The browser could therefore establish a TCP connection to the server and associate it with the name `toServer` using the declaration

```
NetConnection toServer = new NetConnection( url.substring( 7,22 ), WEB_PORT );
```

(assuming that the name `WEB_PORT` was associated with the port number 80).

The problem with such code is that while the numbers 7 and 22 will work for the New York Times site, they will not work for many other URLs. If you decided you needed to read the editorials published in the Wall Street Journal, you would need to enter the URL

```
http://www.wsj.com/public/page/opinion.html
```

If this `String` was associated with the variable named `url`, then the expression

```
url.substring( 7, 22 )
```

would return the value `"www.wsj.com/pub"`. If you tried to use this value as a parameter in a `NetConnection` construction, you would receive an error message because this `String` is not the name of any server on the network. If you want to connect to the server for the Wall Street Journal, you have to replace 22 with the position of the end of the server's name within its URL, 18. This would lead to a declaration of the form

```
NetConnection toServer = new NetConnection( url.substring( 7,18 ), WEB_PORT );
```

Of course, we cannot include separate code for every web server in the world within a browser. We need to find a way to write code that can extract the server's name from a URL correctly regardless of its length. The `indexOf` method makes this possible.

In its simplest form, the `indexOf` method takes a substring to look for as a parameter and returns an `int` value describing the position at which the first copy of the substring is found within the `String` to which the method is applied. For example, assuming we declare

```
String url = "http://www.nytimes.com/pages/opinion/index.html";
```

then the expression

```
url.indexOf( "www" )
```

would produce the value 7, and the expression

```
url.indexOf( "http" )
```

would return the value 0.

We could use such simple invocations of `indexOf` to write code to extract the server's name from a URL, but it would be a bit painful. The problem is that the best way to find the end of the server's name is to use `indexOf` to look for the `"/"` that comes after the name. If we try to use the expression

```
url.indexOf( "/" )
```

to do this, it won't return 22, the position of the `"/"` that comes after the server name. Instead, it will return 5, the position of the first `"/"` that appears in the prefix `http://`. By default, `indexOf` returns the position of the *first* occurrence of the `String` it is asked to search for. To dissect a URL, however, we need a way to make it find the third `"/"`.

Java makes this easy by allowing us to specify a position at which we want `indexOf` to start its search as a second, optional parameter. For example, while the expression

```
url.indexOf( "ht" )
```

would return the value 0 since the value associated with `url` begins with the letters `"ht"`, the expression

```
url.indexOf( "ht", 5 )
```

would return 43 because the first place that the letters `"ht"` appear after position 5 in the `String` is in the `"html"` at the very end.

It is important to remember that `indexOf` starts it search immediately at the position specified by the second parameter. Thus the expressions

```
url.indexOf( "/", 5 )
```

and

```
url.indexOf( "/", 6 )
```

would return 5 and 6, respectively, because there are slashes at positions 5 and 6 in `url`. The expression

```
url.indexOf( "/", 7 )
```

on the other hand, would return 22, the position of the first slash found after the `"http://"` prefix. In general, this expression will return the position of the slash that indicates the end of the server's name within the URL. This is exactly what we need to extract the server's name from the URL.

To make the code to extract the server name as clear as possible, we will associate names with the positions where the server's name begins and ends. We will therefore begin with two variable declarations

```
int nameStart = 7;
int nameEnd = url.indexOf( "/", nameStart );
```

Then, we can extract the server name and associate it with a local variable

```
String serverName = url.substring( nameStart, nameEnd );
```

Finally, we can use the server's name to create a `NetConnection` through which we can send a request for the desired web page.

```
NetConnection toServer = new NetConnection( serverName, WEB_PORT );
```

These instructions will work equally well for both of the URLs

```
http://www.nytimes.com/pages/opinion/index.html
```

and

```
http://www.wsj.com/public/page/opinion.html
```

as well as many others.

There is one remaining aspect of `indexOf` that we need to discuss. What should `indexOf` do if it cannot find the substring we ask it to search for? For example, consider what will happen if we try to execute the code in the preceding paragraph when the variable `url` is associated with the familiar address

```
http://www.google.com
```

```
int nameStart = 7;
int nameEnd = url.indexOf( "/", nameStart );
String serverName;
if ( nameEnd == -1 ) {
    serverName = url.substring( nameStart );
} else {
    serverName = url.substring( nameStart, nameEnd );
}
NetConnection toServer = new NetConnection( serverName, WEB_PORT );
```

Figure 8.1: Code fragment to connect to a server specified in a URL

The last slash in Google's URL appears at position 6. Therefore, if `indexOf` starts searching for a slash in position 7, it will not find one.

By definition, in any situation where `indexOf` cannot find a copy of the substring it is asked to search for, it returns the value -1. Therefore, when `indexOf` is used it is typical to include an `if` statement to check whether the value returned is -1 and to execute appropriate code when this occurs. For example, if `indexOf` cannot find a slash after position 7 in a URL, then we can assume that everything from position 7 until the end of the `String` should be treated as the host name. Based on this idea, code that will correctly extract the server name from a URL and connect to the server is shown in Figure 8.1.

There is one slightly subtle aspect of the code in Figure 8.1. Note that the declaration of the local variable `serverName` is placed before the `if` statement rather than being combined with the first assignment to the name as we have done for the local variables `nameStart` and `nameEnd`. This is essential. If we had attempted to declare `serverName` by replacing the first assignment within the `if` statement with an initialized declaration of the form

```
String serverName = url.substring( nameStart );
```

the program would produce an error message when compiled. The problem is that the first assignment within the `if` statement appears within a set of curly braces that form a separate scope. Any name declared within this scope can only be referenced within the scope. Therefore, if `serverName` was declared here it could only be used in the first half of the `if` statement.

8.3 Separate but `.equals()`

There are many situations where a program needs to determine whether the contents of an entire `String` match a certain word or code. For example, a program might need to determine whether a user entered `"yes"` or `"no"` in answer to a question in a dialog, or whether the code an SMTP server sent to a client included the code for success (`"250"`) or failure (`"500"`). Java includes a method named `equals` that can be used in such programs. As an example of the `equals` method, we will show how it can be used to perform the appropriate action when one of several menu items is selected.

Many programs include menus that are used to enter a time of day. There may be one menu to select the hour, one to select a number of minutes, and a final menu used to indicate whether

Figure 8.2: Selecting daytime from an AM/PM menu

Figure 8.3: Menu selection makes darkness fall

the time is `"AM"` or `"PM"`. To explore a very specific aspect of the code that such programs might contain, we will consider a somewhat silly program that contains just one menu used to select between `"AM"` or `"PM"`. This menu will appear alone in a window like the one shown in Figure 8.2.

During the day, it is bright outside. Accordingly, when the `"AM"` menu item in our program is selected, the background of its window will be white as in Figure 8.2. On the other hand, when the `"PM"` menu item is selected, the program's window will become pitch black as shown in Figure 8.3.

A program that implements this behavior is shown in Figure 8.4. The code that uses the `equals` method can be found in the `menuItemSelected` method. When the user selects either the `"AM"` or `"PM"` menu item, the first line in this method:

```
String chosen = menu.getSelectedItem().toString();
```

associates the local variable name `chosen` with the item that was selected. Then, an `if` statement with the header

```
if ( chosen.equals( "AM" ) ) {
```

is used to decide whether to make the background black or white. The `equals` method will return `true` only if the `String` passed as a parameter, `"AM"` in this example, is composed of exactly the same sequence of symbols as the `String` to which the method is applied, `chosen`.

This should all seem quite reasonable, but if you think hard about some of the conditions we have included in `if` statements in other examples, the existence of the `equals` method may strike you as a bit odd. Recall that in many `if` statements we have used relational operators like `<` and `>=`. One of the available relational operators is `==`, which we have used to test if two expressions describe the same value. If Java already has `==`, why do we also need an `equals` method? Why not simply replace the header of the `if` statement seen in Figure 8.4 with:

```
// A menu driven program that illustrates the use of .equals
public class NightAndDay extends GUIManager {
    // Dimensions of the programs's window
    private final int WINDOW_WIDTH = 300, WINDOW_HEIGHT = 100;

    // The menu that controls the background color
    private JComboBox menu = new JComboBox();

    // Place a menu in a window on the screen
    public NightAndDay() {
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

        menu.addItem( "AM" );
        menu.addItem( "PM" );
        contentPane.add( menu );

        changeBackground( Color.WHITE );
    }

    // Respond to menu selections by setting the background color
    public void menuItemSelected() {
        String chosen = menu.getSelectedItem().toString();

        if ( chosen.equals( "AM" ) ) {
            changeBackground( Color.WHITE );
        } else {
            changeBackground( Color.BLACK );
        }
    }

    // Set the background color of both the menu and contentPane
    private void changeBackground( Color shade ) {
        menu.setBackground( Color.WHITE );
        contentPane.setBackground( Color.WHITE );
    }
}
```

Figure 8.4: A trivial simulation of the cycle of night and day

```
if ( chosen == "AM" ) {
```

Wouldn't this version of the program do the same thing?

The short and somewhat embarrassing answer to the last question is that the behavior of the program in Figure 8.4 would not change if we used `==` instead of `equals`. In general, however, `==` and `equals` do not always produce the same answer, as we can see by making a very minor change in our program.

The difference between using `equals` and `==` is closely connected to the differences between primitive values and objects that we discussed in Section 7.2.2. Java views values of primitive types like `int` as abstract and unchanging. As we explained in that section, it would not make sense to say

```
new int( 3 )
```

in a Java program, because there is only one value called 3. It would make no sense to make another one. If several variables in a Java program are associated with the value 3 at the same time, then we think of them as all being associated with exactly the same thing rather than each with their own separate copy of 3.

With objects, on the other hand, it is possible to have two names refer to two objects that are identical but distinct. For example, if we say

```
JButton start = new JButton( "Start" );
JButton go = new JButton( "Start" );
```

the two variables declared will refer to distinct but identical objects.

`String`s are **not** primitive values in Java. `String`s are objects. Therefore, it is possible to have two `String`s that are identical, in the sense that they contain exactly the same characters, but still distinct. This usually happens when at least one of the `String` values involved is produced using a `String` method or the concatenation operator. For example, suppose that we replaced the code to construct the menu used in the `NightAndDay` program with the following code:

```
String commonSuffix = "M":
menu.addItem( "A" + commonSuffix );
menu.addItem( "P" + commonSuffix );
```

While this is a silly change to make, one would not expect this change to alter the way in which the program behaves. If the program used `==` to determine which menu item was selected, however, this change would make the program function incorrectly.

The issue is that when we tell Java to create a `String` using the `+` operator, it considers the new string to be distinct from all other `String`s. Even though we didn't explicitly perform a construction, the concatenation operator produces a `new` value. In particular, the `String` produced by the expression

```
"A" + commonSuffix
```

would be distinct from the `String` produced by the expression `"AM"`, even though both `String`s would have exactly the same contents.

When we ask `Java` if two `String`s (or in fact if any two objects) are `==`, it produces the result `true` only if the two operands we provide are actually identical. When we ask `Java` if two `String`s

are `equals`, on the other hand, it produces the result `true` only if the two operands are `String`s that contain exactly the same sequence of symbols.

In particular, if we change the `NightAndDay` program to use the expression

```
"A" + commonSuffix
```

to describe the first item to be placed in `menu`, then when the user selects this item, evaluating the condition

```
chosen.equals( "AM" )
```

will produce `true`, while the condition

```
chosen == "AM"
```

would produce `false`. As a result, the program would never recognize that a user selected the `"AM"` item and therefore never set the background back to white after it had become black.

As this example illustrates, it can be hard to predict when Java will consider two `String`s to be identical. The good news is that it is not hard to avoid the issue in your programs. Simply remember that when you want to check to see if two `String` values are the same you should use `equals`. You should almost never use `==` to compare the values of two `String`s.

8.4 Methods and More Methods

At this point, we have introduced just a handful of the methods Java provides for working with `String` values. The methods we have presented are all that you will really need for the vast majority of programs that manipulate `String`s. Many of the remaining methods, however, can be very convenient in certain situations.

For example, suppose that you need to test whether the word `"yes"` appears somewhere in a message entered by a user. You can do this using a technique involving the `indexOf` method. The code in an `if` statement of the form

```
if ( userMessage.indexOf( "yes" ) >= 0 ) {
    . . .
}
```

will only be executed if the substring `"yes"` appears somewhere within `userMessage`. (Can you see why?) Nevertheless, Java provides a different method that is a bit easier to use in such situations and leads to much clearer code. The method is named `contains`. It returns `true` only if the value provided to the method as a parameter appears within the `String` to which it is applied. Thus, we can replace the code shown above with

```
if ( userMessage.contains( "yes" ) ) {
    . . .
}
```

An important special case of containing a substring is starting with that substring. For example, we have seen that the messages an SMTP server sends to a client all start with a three digit code indicating success or failure. The success code used by SMTP is `"250"`. Therefore, an SMTP

client is likely to contain code to determine whether each message it receives from the server starts with "250". This could be done using the length, equals, and substring methods (and it would probably benefit the reader to take a moment to figure out exactly how), but Java provides a convenient alternative. There are String methods named startsWith and endsWith. Both of these methods take a single String as a parameter and return true or false depending on whether the parameter appears as a prefix (or suffix) of the String to which the method is applied. For example, an if statement of the form

```
if ( serverMessage.startsWith( "250" ) ) {
    . . .
}
```

will execute the code in its body only if "250" appears as a prefix of serverMessage.

Several String methods are designed to make it easy to deal with the difference between upper and lower case characters. In many programs, we simply want to ignore the difference between upper and lower case. For example, if we were trying to see if userMessage contained the word "yes", we probably would not care whether the user typed "yes", "Yes", "YES", or even "yeS". The two if statements shown earlier, however, would only recognize "yes".

There are two String methods named toUpperCase and toLowerCase that provide a simple way to deal with such issues. The toLowerCase method returns a String that is identical to the String to which it is applied except that all upper case letters have been replaced by the corresponding lower case letters. The toUpperCase method performs the opposite transformation. As an example, the body of an if statement of the form

```
if ( userMessage.toLowerCase().contains( "yes" ) ) {
    . . .
}
```

will be executed if userMessage contains "yes", "Yes", "YES", or even "yeS". The subexpression

```
userMessage.toLowerCase()
```

describes the result of replacing any upper case letters that appeared in userMessage with lower case letters. Therefore, if userMessage had contained "Yes" or "YES", the String that toLowerCase returned would contain "yes".

8.5 Online Method Documentation

We could continue describing additional String methods for quite a few more pages. There are methods named replace, equalsIgnoreCase, compareTo, trim, and many others. A better alternative, however, is to tell you how to find out about them yourself.

Sun Microsystems, the company that created and maintains the Java programming language, provides online documentation describing all the methods that can be applied to object types defined within the standard Java library including Strings.If you point your favorite web browser at

```
http://java.sun.com/j2se/1.5.0/docs/api/
```

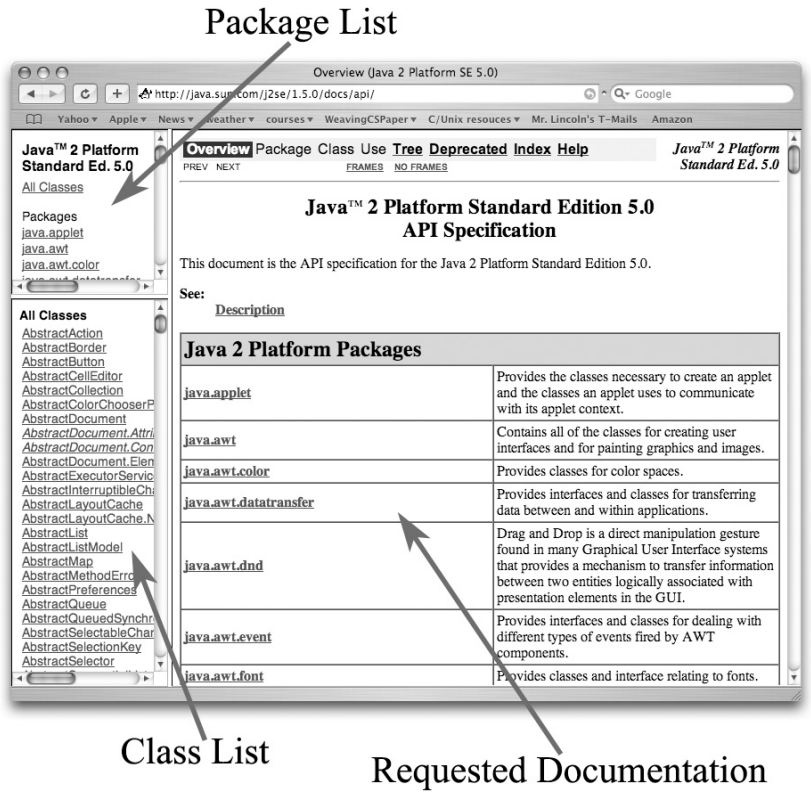

Figure 8.5: Initial page displayed when visiting Sun's Java documentation site

you should see a page that looks something like the one shown in Figure 8.5.[2]

Sun has organized its on-line Java documentation around the classes in the Java libraries. There is a page that describes the methods associated with the `String` class, a page that describes the `JTextField` class, and so on. Therefore, to navigate through the documentation, it helps to know what class you are trying to learn more about. Luckily, we know that we want to learn about the `String` class.

The windows in which Sun's documentation are displayed are divided into three panels. As shown in Figure 8.5, the panel on the right side of the window occupies most of the available space. This is used to display the documentation you have asked to see. In Figure 8.5 is is used to display brief summaries of the "packages" into which all of the classes in the Java libraries are divided. Once you select a particular class, the documentation for this class will be placed in this panel.

The left margin of the window is divided into two smaller panels. The upper panel holds a list of package names and the lower panel holds a list of class names. Initially, the lower panel contains the names of all the classes in the Java libraries. By clicking on a package name in the upper panel, you can reduce the list of classes displayed in the lower panel to just those classes in the selected package.

To examine the documentation of a particular class, you can simply scroll through the names shown in the lower left panel until you find the name of that class. If you then click on the name, the documentation for that class will be placed in the panel on the right. For example, in Figure 8.6 we show how the window might look immediately after we clicked on the name of the `String` class in the lower left panel.

We could use the scroll bar on the right side of the window in Figure 8.6 to read through the entire description of the `String` class, but if we are just trying to learn about more of the methods provided by the class, there is a shortcut we can use. Near the top of the class, there is a short list of links to summaries of the features of the class. One of these, which is identified with an arrow in Figure 8.6 appears under the word "METHOD". If you click on this link, it scrolls the contents of the page to display a collection of short summaries of the methods of the `String` class as shown in Figure 8.7.

In many cases, these short summaries provide enough information to enable you to use the methods. If not, you can easily get a more detailed description. Just click on the name of the method in which you are interested. For example, if you click on the name of the `contains` method, the contents of the window will scroll to display the complete description of the `contains` method as shown in Figure 8.8. As you can see, unfortunately, the complete description of a method sometimes provides little more than the summary.

Of course, by following similar instructions you can use Sun's online documentation to learn more about other types we have worked with. For example, if you search for `JButton` or `JComboBox` in the list found within the bottom left panel of the documentation window, you will be able to read about many additional methods associated with these GUI components.

[2]There is, of course, a risk in including a web site address in any published text. At some point in the future, Sun Microsystems is likely to reorganize their online documentation so that the address given above simply does not work. In this event, try going to `http://java.sun.com` and follow the links for "API documentation". With a bit of luck, you will still be able to find the pages described here.

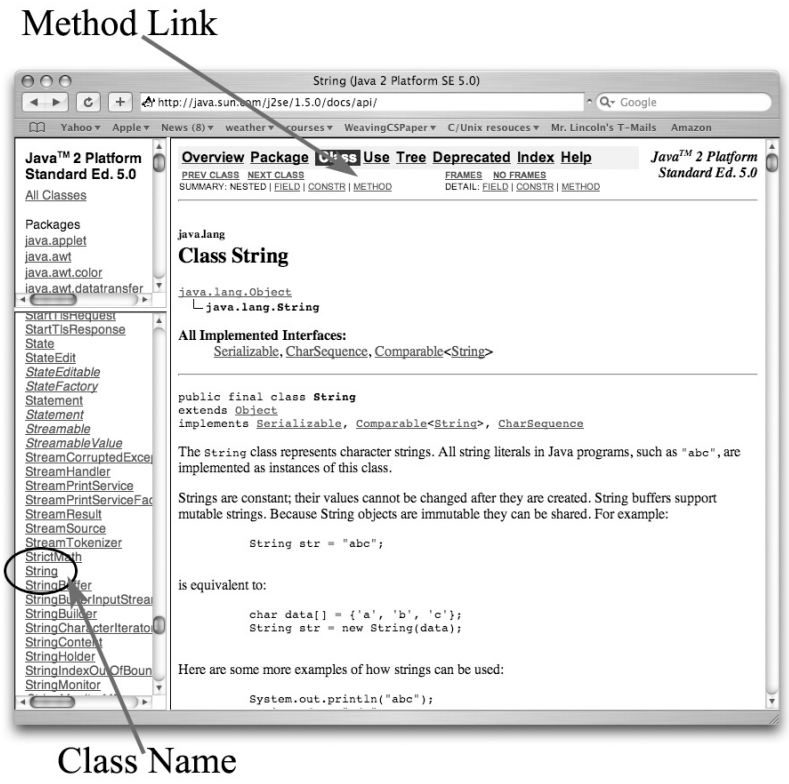

Figure 8.6: Page displaying the documentation for the `String` class

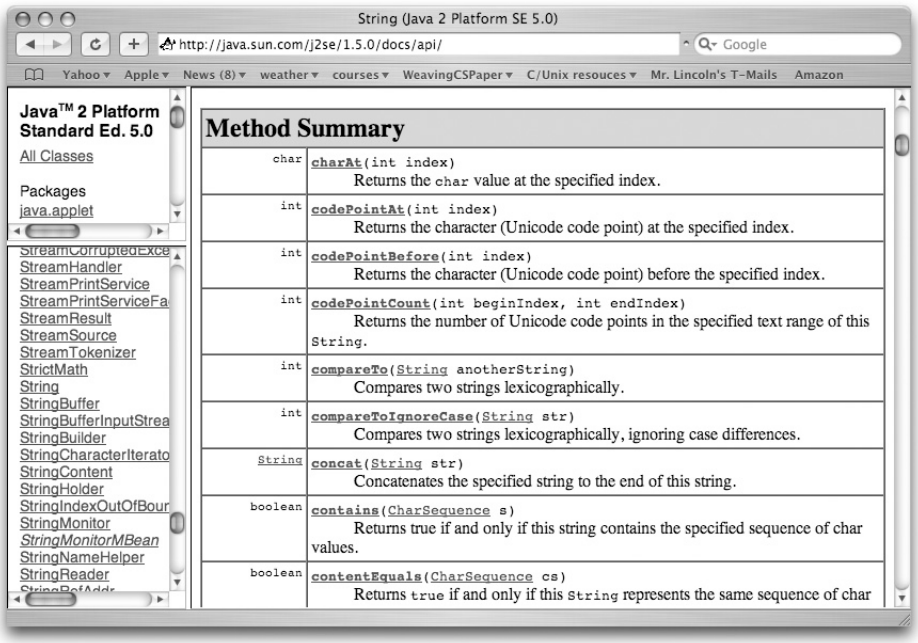

Figure 8.7: Summaries of the methods of the `String` class

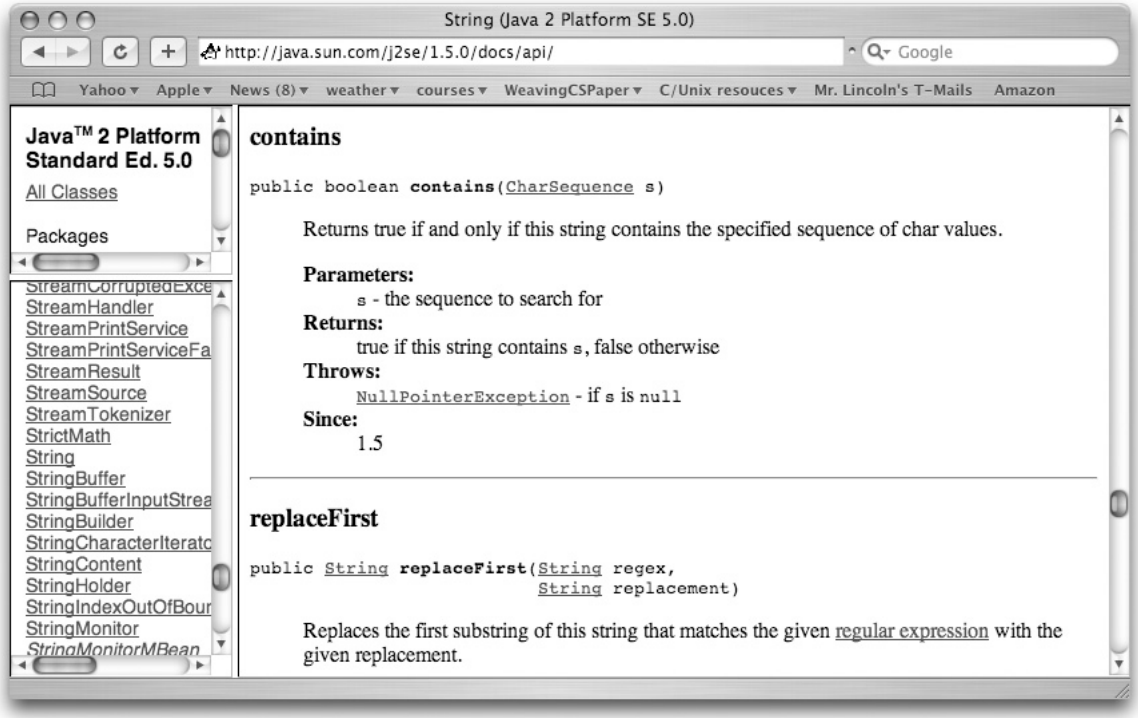

Figure 8.8: Online documentation for the `contains` method of the `String` class

8.6 What a `character`!

As we have noted, `String` is not a primitive type. There is, however, a primitive type within Java that is closely related to the `String` type. This type is named `char`. The values of the `char` type are individual characters.

The relationship between the `String` and `char` types is fairly obvious. A `String` is a sequence of `char` values, i.e., characters. A similar relationship exists in mathematics between a set and its elements. The set $\{2, 3, 5\}$ has the numbers 2, 3, and 5 as elements. In mathematics, the curly braces used to surround the members of a set make it easy to tell the difference between a set containing a single element and the element itself. The set containing the number 2, which we write as $\{2\}$ is clearly different from the number 2. Java provides a similar notational distinction between `String` and `char` literals.

In Java, a `char` literal is denoted by placing a single character or an escape sequence that describes a single character between single rather than double quotes. Thus, while `"E"` is a `String` containing a single character, `'E'` is a single character or `char` value. Similarly, `'7'`, `'?'`, `' '`, and `'\n'` are `char` literals describing the digit 7, the question mark, the space character, and the newline character.

While related, `char` and `String` are distinctly different types in Java. If we declare two variables

```
private char c;
private String s;
```

then the assignments

```
c = "E";
```

and

```
s = 'E' ;
```

would both be considered illegal as would the assignments

```
c = s;
```

and

```
s = c;
```

There is a method that makes it possible to extract individual characters from a `String`. This method is named `charAt`. It takes the position of the character within the `String` as a parameter, treating 0 as the position of the first character. Thus, if we executed the assignments

```
s = "characters";
c = s.charAt( 3 );
```

the variable `c` would become associated with the value `'r'`

None of the `String` methods described above can be applied to `char` values. For example, given the variable `c` declared as a `char`, it would be illegal to say

```
c.toLowerCase()          // This is incorrect code!
```

Actually, there are no methods that can be applied to `char` values. `char` is a primitive type like `int` and `boolean`. Like these other primitive types, there are operators that can be applied to `char` values, but no methods.

In fact, the `char` type is more closely related to the other primitive types than you might imagine. Internally, the computer uses a numeric code to represent each character value. For example, the number 65 is used to represent the character `'A'`, 95 is used to represent `'a'`, and 32 is use to represent the space (`' '`). The numbers assigned to various characters are in some sense arbitrary. `'A'` is clearly not the 65th letter of the alphabet. However, the numbers used to represent characters are not chosen at random. They are part of a widely accepted standard for representing text in a computer that is known as Unicode.

In many contexts, Java treats `char` values very much like the numeric type `int`. For example, if you evaluate the expression

```
'A' + 1
```

the answer will be 66, one more than the number used to represent an `'A'` internally. In Unicode, consecutive letters in the alphabet are represented using consecutive numeric values. Therefore, if you evaluate

```
'B' - 'A'
```

the answer will be 1, and

```
'Z' - 'A'
```

produces 25. On the other hand, the expression

```
'z' - 'A'
```

produces 57 as an answer. The numeric values used for upper and lower case letters are not as meaningfully related as for letters of a single case.

On the other hand, in some situations, Java treats `char` values as characters rather than numbers. For example, if you write

```
"ON" + 'E'
```

the answer produced is `"ONE"`, not `"ON69"`. On the other hand, if you write

```
"ON" + ( 'E' + 1 )
```

the result produced is `"ON70"`. This may seem confusing at first, but the rules at work here are fairly simple.

Java understands that `char` values are numbers. Therefore, Java allows you to perform arithmetic operations with `char` values. You can even say things like

```
'a' * 'b' + 'd'
```

Java also understands, however, that the number produced when it evaluates such an expression is unlikely to correspond to a code that can be meaningfully interpreted in Unicode. Therefore, Java treats the result of an arithmetic expression involving `char` values or a mixture of `char` and `int` values as an `int` value. For example, it would be illegal to say

```
char newChar = 'z' - 'a';
```

but fine to say

```
int charSeparation =  'z' - 'a';
```

This explains the fact that

```
"ON" + ( 'E' + 1 )
```

produces `"ON70"`. We already know that if you apply the `+` operator to a `String` value and an `int` value, Java appends the digits that represent the `int` to the characters found in the `String`. The character `'E'` is represented internally by the number 69. Therefore, Java interprets the expression `'E' + 1` as a rather complicated way to describe the `int` value 70.

On the other hand, when you apply the `+` operator to a `String` value and a `char` value, Java appends the character corresponding to the `char`'s numeric value to the characters found in the `String`. Basically, in this context, Java interprets the `char` as a character rather than as a number.

There is one way in which it is particularly handy that Java treats `char` values as numbers. This involves the use of relational operators. If we declare

```
private char c;
```

then we can say things like

```
if ( c > 'a' ) { . . .
```

When relational operators are applied to `char` values, Java simply compares the numeric codes used to represent the characters. This is useful because Unicode assigns consecutive codes to consecutive characters. As a result, numeric comparisons correspond to checking relationships involving alphabetical ordering.

For example, consider the `if` statement

```
if ( c >= 'a'  && c <= 'z') {
    . . .
} else if  ( c >= 'A'  && c <= 'Z') {
    . . .
}
```

The condition in the first `if` checks to see if the value associated with `c` is somewhere between the values used to represent lower case `'a'` and `'z'`. This will only happen if `c` is associated with a lower case letter. Therefore, the code placed in the first branch of the `if` will only be executed if `c` is a lower case letter. For similar reasons, the second branch will only be executed if `c` is upper case.

8.7 Summary

In this chapter, we have explored the facilities Java provides for manipulating text. We introduced one new type related to textual data, the `char` type and presented many methods of the `String` type that had not been discussed previously. We showed how the `substring` and `indexOf` methods together with the concatenation operator can be used to assemble and disassemble `String` values. We also introduced several additional methods including `contains`, `startsWith`, `toLowerCase` and

`toUpperCase`. More importantly, we outlined how you can use online documentation to explore even more methods of the `String` class and of other Java classes.

Since the bulk of this chapter was devoted to introducing new methods, we conclude this section with brief descriptions of the most useful `String` methods. We hope these summaries can serve as a reference.

8.7.1 Summary of `String` Methods

`someString.contains( otherString )`	The `contains` method produces a `boolean` indicating whether or not `otherString` appears as a substring of `someString`. If the string provided as the `otherString` argument appears within `someString` then the value produced will be `true`. Otherwise, the result will be `false`.
`someString.endsWith( otherString )`	The `endsWith` method produces a `boolean` indicating whether or not `otherString` appears as a suffix of `someString`. If the string provided as the `otherString` argument is a suffix of `someString` then the value produced will be `true`. If `otherString` is not a suffix of `someString`, the result will be `false`.
`someString.equals( otherString )`	The `equals` method produces a `boolean` indicating whether or not `otherString` contains the same sequence of characters as `someString`. If the string provided as the `otherString` argument is identical to `someString` then the value produced will be `true`. Otherwise, the result will be `false`.

```
someString.indexOf( otherString )
someString.indexOf( otherString, start )
```

The `indexOf` method searches a string looking for the first occurrence of another string provided as the first or only parameter to the method. If it finds an occurrence of the argument string, it returns the position of the first character where the match was found. If it cannot find an occurrence of the argument string, then the value -1 is produced. If a second actual parameter is included in an invocation of `indexOf`, it specifies the position within `someString` where the search should begin. The result returned will still be a position relative to the beginning of the entire string.

```
someString.length()
```

The `length` method returns an `int` equal to the number of characters in the string. Blanks, tabs, punctuation marks, and all other symbols are counted in the length.

```
someString.startsWith( otherString )
```

The `startsWith` method produces a `boolean` indicating whether or not `otherString` appears as a prefix of `someString`. If the string provided as the `otherString` argument is a prefix of `someString` then the value produced will be `true`. If `otherString` is not a prefix of `someString`, the result will be `false`.

```
someString.substring( start )
someString.substring( start, end )
```

The `substring` method returns a specified sub-section of the string to which it is applied. If a single argument is provided, the string returned will consist of all characters in the original string from the position `start` to the end of the string. If both a `start` and `end` position are provided as arguments, the string returned will consist of all characters from the position `start` in the original string up to but not including the character at position `end`. Positions within the string are numbered starting at 0. If `start` or `end` are negative or larger than the length of the string, an error will occur. An error will also occur if `end` is less than `start`. If `end` is equal to `start`, then the empty string (`""`) will be returned.

`someString.toLowerCase( )`	The `toLowerCase` method produces a `String` that is identical to `someString` except that any upper case letters are replaced with the corresponding lower case letters. `someString` itself is not changed.
`someString.toUpperCase( )`	The `toUpperCase` method produces a `String` that is identical to `someString` except that any lower case letters are replaced with the corresponding upper case letters. `someString` itself is not changed.