



Strings in Java

As is the case in most other programming languages, in Java, a string is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type String. Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, String objects can be constructed in a number of ways, making it easy to obtain a string when needed.

Objectives

Upon completion of this topic, we will be able to:

- Explain various string operations in Java
- Handle strings in Java
- Discuss about string operations, character extractions and string comparison
- Discuss methods of searching strings and its modifications
- Explain StringBuffer.

Content_Heading

String Handling

When you create a String object, you are creating a string that cannot be changed. That is, once a String object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction. However, such is not the case. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new String object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, there is a companion class to String called StringBuffer, whose objects contain strings that can be modified after they are created.

Both the String and StringBuffer classes are defined in java.lang package. Thus, they are available to all programs automatically. Both are declared final, which means that neither of these classes may be sub-classed. This allows certain optimizations that increase performance to take place on common string operations.

When we say that the strings within objects of type String are unchangeable it means that the contents of the String instance cannot be changed after it has been created. However, a variable declared as a String reference can be changed to point at some other String object at any time.

The String class supports several constructors. To create an empty String, you call the default constructor. For example,



String s = new String();

It will create an instance of String with no characters in it.

Frequently, you may want to create strings that have initial values. The String class provides a variety of constructors to handle this. To create a String initialized by an array of characters, use the constructor shown here:

String(char chars[]);

Here is an example:

char chars[] = { 'a', 'b', 'c' };

String s = new String(chars);

This constructor initializes s with the string "abc".

You can specify a sub-range of a character array as an initializer using the following constructor:

String(char chars[], int startIndex, int numChars);

Here, startIndex specifies the index at which the sub-range begins, and numChars specifies the number of characters to use. Here is an example:

char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

String s = new String(chars, 2, 3);

This initializes s with the string "cde". Please remember that the array index starts with 0.

You can construct a String object that contains the same character sequence as another String object using this constructor:

String(String strObj);

Here, strObj is a String object. Consider this example:

// Construct one String from another.

```
class MakeString {  
    public static void main(String args[ ]) {  
        char c[ ] = { 'J', 'a', 'v', 'a' };  
        String s1 = new String(c);  
        String s2 = new String(s1);  
        System.out.println(s1);  
    }  
}
```



```
System.out.println(s2);
```

```
}
```

```
}
```

The output from this program is as follows:

Java

Java

As you can see, s1 and s2 contain the same string.

Even though Java's char type uses 16 bits to represent the Unicode character set, the typical format for strings on the Internet uses arrays of

8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array. Their forms are shown here:

```
String(byte asciiChars[ ]);
```

```
String(byte asciiChars[ ], int startIndex, int numChars);
```

Here, asciiChars specifies the array of bytes. The second form allows you to specify a sub-range. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform. The following program illustrates these constructors:

```
// Construct string from subset of char array.
```

```
class SubStringCons {
```

```
public static void main(String args[ ]) {
```

```
byte ascii[ ] = {65, 66, 67, 68, 69, 70 };
```

```
String s1 = new String(ascii);
```

```
System.out.println(s1);
```

```
String s2 = new String(ascii, 2, 3);
```

```
System.out.println(s2);
```

```
}
```

```
}
```



This program generates the following output:

ABCDEF

CDE

Extended versions of the byte-to-string constructors are also defined in which you can specify the character encoding that determines how bytes are converted to characters.

However, most of the time, you will require to use the default encoding provided by the platform.

The length of a string is the number of characters that it contains. To obtain this value, call the `length()` method, as shown here:

`int length();`

The following fragment prints "3", since there are three characters in the string `s`:

`char chars[] = { 'a', 'b', 'c' };`

`String s = new String(chars);`

`System.out.println(s.length());`

Special String Operations

Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language. These operations include the automatic creation of new `String` instances from string literals, concatenation of multiple `String` objects by use of the `+` operator, and the conversion of other data types to a string representation. There are explicit methods available to perform all of these functions, but Java does them automatically as a convenience for the programmer and to add clarity.

String Literals

The earlier examples showed how to explicitly create a `String` instance from an array of characters by using the `new` operator. However, there is an easier way to do this using a string literal. For each string literal in your program, Java automatically constructs a `String` object. Thus, you can use a string literal to initialize a `String` object. For example, the following code fragment creates two equivalent strings:

`char chars[] = { 'a', 'b', 'c' };`

`String s1 = new String(chars);`

`String s2 = "abc"; // use string literal`

Because a `String` object is created for every string literal, you can use a string literal any place you can use a `String` object. For example, you can call methods directly on a quoted string as if it were



an object reference, as the following statement shows. It calls the `length()` method on the string "abc". As expected, it prints "3".

```
System.out.println("abc".length());
```

String Concatenation

In general, Java does not allow operators to be applied to String objects. The one exception to this rule is the `+` operator, which concatenates two strings, producing a String object as the result. This allows you to chain together a series of `+` operations. For example, the following fragment concatenates three strings:

```
String age = "9";
```

```
String s = "He is " + age + " years old.";
```

```
System.out.println(s);
```

This displays the string "He is 9 years old."

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the `+` to concatenate them. Here is an example:

```
// Using concatenation to prevent long lines.
```

```
class ConCat {
```

```
    public static void main(String args[ ]) {
```

```
        String longStr = "This could have been " +
```

```
        "a very long line that would have " +
```

```
        "wrapped around. But string concatenation " +
```

```
        "prevents this.";
```

```
        System.out.println(longStr);
```

```
    }
```

```
}
```

You can concatenate strings with other types of data also. Be careful when you mix other types of operations with string concatenation expressions. You might get surprising results.

```
System.out.println(s);
```

This fragment displays four: 22 rather than the four: 4 that you probably expected. Here's why. Operator precedence causes the concatenation of "four" with the string equivalent of 2 to take place



first. This result is then concatenated with the string equivalent of 2 a second time. To complete the integer addition first, you must use parentheses, like this:

```
String s = "four: " + (2 + 2);
```

Now s contains the string "four: 4".

String Conversion and toString()

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method `valueOf()` defined by `String`. `valueOf()` is overloaded for all the simple types and for type `Object`. For the simple types, `valueOf()` returns a string that contains the human-readable equivalent of the value with which it is called. For objects, `valueOf()` calls the `toString()` method on the object. We will look more closely at `valueOf()` later in this chapter. Here, let's examine the `toString()` method, because it is the means by which you can determine the string representation for objects of classes that you create. Every class implements `toString()`, because it is defined by `Object`. However, the default implementation of `toString()` is seldom sufficient. For most important classes that you create, you may want to override `toString()` and provide your own string representations. Fortunately, this is easy to do. The `toString()` method has this general form:

String toString()

To implement `toString()`, simply return a `String` object that contains the human-readable string that appropriately describes an object of your class.

By overriding `toString()` for classes that you create, you allow the resulting strings to be fully integrated into Java's programming environment.

Character Extraction

The `String` class provides a number of ways in which characters can be extracted from a `String` object. Each is examined here. Although the characters that comprise a string within a `String` object cannot be indexed as if they were a character array, many of the `String` methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

charAt()

To extract a single character from a `String`, you can refer directly to an individual character via the `charAt()` method. It has this general form:

```
char charAt(int where);
```

Here, `where` is the index of the character that you want to obtain. The value of `where` must be nonnegative and specify a location within the string. `charAt()` returns the character at the specified location. For example,

```
char ch;
```



```
ch = "abc".charAt(1);
```

assigns the value "b" to ch.

getChars()

If you need to extract more than one character at a time, you can use the `getChars()` method. It has this general form:

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

Here, `sourceStart` specifies the index of the beginning of the substring, and `sourceEnd` specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from `sourceStart` through `sourceEnd-1`. The array that will receive the characters is specified by `target`. The index within `target` at which the substring will be copied is passed in `targetStart`. Care must be taken to assure that the `target` array is large enough to hold the number of characters in the specified substring.

getBytes()

There is an alternative to `getChars()` that stores the characters in an array of bytes. This method is called `getBytes()`, and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

```
byte[ ] getBytes( );
```

Other forms of `getBytes()` are also available. `getBytes()` is most useful when you are exporting a `String` value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray()

If you want to convert all the characters in a `String` object into a character array, the easiest way is to call `toCharArray()`. It returns an array of characters for the entire string.

It has this general form:

```
char[ ] toCharArray( );
```

This function is provided as a convenience, since it is possible to use `getChars()` to achieve the same result.

String Comparison

The `String` class includes several methods that compare strings or substrings within strings. Each is examined here.



`equals()` and `equalsIgnoreCase()`

To compare two strings for equality, use `equals()`. It has this general form:

`boolean equals(Object str);`

Here, `str` is the `String` object being compared with the invoking `String` object. It returns `true` if the strings contain the same characters in the same order, and `false` otherwise.

The comparison is case-sensitive. To perform a comparison that ignores case differences, call `equalsIgnoreCase()`. When it compares two strings, it considers A-Z to be the same as a-z. It has this general form:

`boolean equalsIgnoreCase(String str);`

Here, `str` is the `String` object being compared with the invoking `String` object. It, too, returns `true` if the strings contain the same characters in the same order, and `false` otherwise.

Here is an example that demonstrates `equals()` and `equalsIgnoreCase()`:

// Demonstrate `equals()` and `equalsIgnoreCase()`.

```
class equalsDemo {  
    public static void main(String args[ ]) {  
        String s1 = "Hello";  
        String s2 = "Hello";  
        String s3 = "Good-bye";  
        String s4 = "HELLO";  
        System.out.println(s1 + " equals " + s2 + " -> " +  
            s1.equals(s2));  
        System.out.println(s1 + " equals " + s3 + " -> " +  
            s1.equals(s3));  
        System.out.println(s1 + " equals " + s4 + " -> " +  
            s1.equals(s4));  
        System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +  
            s1.equalsIgnoreCase(s4));  
    }  
}
```




```
}
```

The output from the program is shown here:

Hello equals Hello -> true

Hello equals Good-bye -> false

Hello equals HELLO -> false

Hello equalsIgnoreCase HELLO -> true

regionMatches()

The `regionMatches()` method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

```
boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars);
```

```
boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars);
```

For both versions, `startIndex` specifies the index at which the region begins within the invoking `String` object. The `String` being compared is specified by `str2`. The index at which the comparison will start within `str2` is specified by `str2StartIndex`. The length of the substring being compared is passed in `numChars`. In the second version, if `ignoreCase` is true, the case of the characters is ignored. Otherwise, case is significant.

`startsWith()` and `endsWith()`

`String` defines two routines that are, more or less, specialized forms of `regionMatches()`. The `startsWith()` method determines whether a given `String` begins with a specified string. Conversely, `endsWith()` determines whether the `String` in question ends with a specified string. They have the following general forms:

```
boolean startsWith(String str);
```

```
boolean endsWith(String str);
```

Here, `str` is the `String` being tested. If the string matches, true is returned. Otherwise, false is returned. For example,

```
"Foobar".endsWith("bar");
```

```
"Foobar".startsWith("Foo");
```

are both true.

A second form of `startsWith()`, shown here, lets you specify the starting point:



```
boolean startsWith(String str, int startIndex);
```

Here, startIndex specifies the index into the invoking string at which point the search will begin. For example,

```
"Foobar".startsWith("bar", 3);
```

returns true.

`equals()` Versus `==`

It is important to understand that the `equals()` method and the `==` operator perform two different operations. As just explained, the `equals()` method compares the characters inside a String object. The `==` operator compares two object references to see whether they refer to the same instance. The following program shows how two different String objects can contain the same characters, but references to these objects will not compare as equal:

// equals() vs ==

```
class EqualsNotEqualTo {  
    public static void main(String args[] ) {  
        String s1 = "Hello";  
        String s2 = new String(s1);  
        System.out.println(s1 + " equals " + s2 + " -> " +  
            s1.equals(s2));  
        System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));  
    }  
}
```

The variable s1 refers to the String instance created by "Hello". The object referred to by s2 is created with s1 as an initializer. Thus, the contents of the two String objects are identical, but they are distinct objects. This means that s1 and s2 do not refer to the same objects and are, therefore, not `==`, as is shown here by the output of the preceding example:

Hello equals Hello -> true

Hello == Hello -> false



compareTo()

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is less than, equal to, or greater than the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The String method `compareTo()` serves this purpose. It has this general form:

```
int compareTo(String str);
```

Here, `str` is the String being compared with the invoking String. The result of the comparison is returned and is interpreted as shown here:

| Value | Meaning |
|-------------------|--|
| Less than zero | The invoking string is less than <code>str</code> . |
| Greater than zero | The invoking string is greater than <code>str</code> . |
| Zero | The two strings are equal. |

If you want to ignore case differences when comparing two strings, use

`compareToIgnoreCase()`, shown here:

```
int compareToIgnoreCase(String str)
```

This method returns the same results as `compareTo()`, except that case differences are ignored.

Searching Strings

The String class provides two methods that allow you to search a string for a specified character or substring:

- `indexOf()` Searches for the first occurrence of a character or substring.
- `lastIndexOf()` Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or `-1` on failure.

To search for the first occurrence of a character, use

```
int indexOf(int ch);
```

To search for the last occurrence of a character, use

```
int lastIndexOf(int ch);
```

Here, `ch` is the character being sought.

To search for the first or last occurrence of a substring, use

```
int indexOf(String str);
```



```
int lastIndexOf(String str);
```

Here, str specifies the substring.

You can specify a starting point for the search using these forms:

```
int indexOf(int ch, int startIndex);
```

```
int lastIndexOf(int ch, int startIndex);
```

```
int indexOf(String str, int startIndex);
```

```
int lastIndexOf(String str, int startIndex);
```

Here, startIndex specifies the index at which point the search begins. For indexOf(), the search runs from startIndex to the end of the string. For lastIndexOf(), the search runs from startIndex to zero.

String Modification

Because String objects are immutable, whenever you want to modify a String, you must either copy it into a StringBuffer or use one of the following String methods, which will construct a new copy of the string with your modifications complete.

```
substring( )
```

You can extract a substring using substring(). It has two forms. The first is

```
String substring(int startIndex);
```

Here, startIndex specifies the index at which the substring will begin. This form returns a copy of the substring that begins at startIndex and runs to the end of the invoking string.

The second form of substring() allows you to specify both the beginning and ending index of the substring:

```
String substring(int startIndex, int endIndex);
```

Here, startIndex specifies the beginning index, and endIndex specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

The following program uses substring() to replace all instances of one substring with another within a string:

// Substring replacement.

```
class StringReplace {
```

```
    public static void main(String args[ ]) {
```

```
        String org = "This is a test. This is, too.";
```



```
String search = "is";  
String sub = "was";  
String result = "";  
int i;  
do { // replace all matching substrings  
    System.out.println(org);  
    i = org.indexOf(search);  
    if(i != -1) {  
        result = org.substring(0, i);  
        result = result + sub;  
        result = result + org.substring(i + search.length());  
        org = result;  
    }  
} while(i != -1);  
}  
}
```

The output from this program is shown here:

This is a test. This is, too.

Thwas is a test. This is, too.

Thwas was a test. This is, too.

Thwas was a test. Thwas is, too.

Thwas was a test. Thwas was, too.

concat()

You can concatenate two strings using concat(), shown here:

String concat(String str);



This method creates a new object that contains the invoking string with the contents of str appended to the end. `concat()` performs the same function as `+`. For example,

```
String s1 = "one";
```

```
String s2 = s1.concat("two");
```

puts the string "onetwo" into s2. It generates the same result as the following sequence:

```
String s1 = "one";
```

```
String s2 = s1 + "two";
```

```
replace( )
```

The `replace()` method replaces all occurrences of one character in the invoking string with another character. It has the following general form:

```
String replace(char original, char replacement);
```

Here, original specifies the character to be replaced by the character specified by replacement. The resulting string is returned. For example,

```
String s = "Hello".replace('l', 'w');
```

puts the string "Hewwo" into s.

```
trim( )
```

The `trim()` method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

```
String trim( );
```

Here is an example:

```
String s = " Hello World ".trim();
```

This puts the string "Hello World" into s.

The `trim()` method is quite useful when you process user commands. For example, the following program prompts the user for the name of a state and then displays that state's capital. It uses `trim()` to remove any leading or trailing whitespace that may have inadvertently been entered by the user.

Data Conversion Using `valueOf()`

The `valueOf()` method converts data from its internal format into a human-readable form. It is a static method that is overloaded within `String` for all of Java's built-in types, so that each type can be converted properly into a string. `valueOf()` is also overloaded for type `Object`, so an object of any class type you create can also be used as an argument. (Recall that `Object` is a superclass for all classes.) Here are a few of its forms:



static String valueOf(double num);

static String valueOf(long num);

static String valueOf(Object ob);

static String valueOf(char chars[]);

As we discussed earlier, `valueOf()` is called when a string representation of some other type of data is needed – for example, during concatenation operations. You can call this method directly with any data type and get a reasonable String representation. All of the simple types are converted to their common String representation. Any object that you pass to `valueOf()` will return the result of a call to the object's `toString()` method. In fact, you could just call `toString()` directly and get the same result. For most arrays, `valueOf()` returns a rather cryptic string, which indicates that it is an array of some type. For arrays of `char`, however, a String object is created that contains the characters in the char array. There is a special version of `valueOf()` that allows you to specify a subset of a char array. It has this general form:

static String valueOf(char chars[], int startIndex, int numChars);

Here, `chars` is the array that holds the characters, `startIndex` is the index into the array of characters at which the desired substring begins, and `numChars` specifies the length of the substring.

Changing the Case of Characters within a String

The method `toLowerCase()` converts all the characters in a string from uppercase to lowercase. The `toUpperCase()` method converts all the characters in a string from lowercase to uppercase. Non-alphabetical characters, such as digits, are unaffected. Here are the general forms of these methods:

String toLowerCase();

String toUpperCase();

Both methods return a String object that contains the uppercase or lowercase equivalent to the invoking String.

StringBuffer

StringBuffer is a peer class of String that provides much of the functionality of strings. As you know, String represents fixed-length, immutable character sequences. In contrast, StringBuffer represents growable and writeable character sequences.

StringBuffer may have characters and substrings inserted in the middle or appended to the end. StringBuffer will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth. Java uses both classes heavily,



but many programmers deal only with String and let Java manipulate StringBuffer behind the scenes by using the overloaded + operator.

StringBuffer defines these three constructors:

StringBuffer();

StringBuffer(int size);

StringBuffer(String str);

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation. StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, StringBuffer reduces the number of reallocations that take place.

length() and capacity()

The current length of a StringBuffer can be found via the length() method, while the total allocated capacity can be found through the capacity() method. They have the following general forms:

int length();

int capacity();

The following program demonstrates the use of length() and capacity().

// StringBuffer length vs. capacity.

```
class StringBufferDemo {  
    public static void main(String args[ ]) {  
        StringBuffer sb = new StringBuffer("Hello");  
        System.out.println("buffer = " + sb);  
        System.out.println("length = " + sb.length());  
        System.out.println("capacity = " + sb.capacity());  
    }  
}
```




Here is the output of this program, which shows how StringBuffer reserves extra space for additional manipulations:

buffer = Hello

length = 5

capacity = 21

Since **sb** is initialized with the string "Hello" when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

ensureCapacity()

If you want to pre-allocate room for a certain number of characters after a StringBuffer has been constructed, you can use `ensureCapacity()` to set the size of the buffer. This is useful if you know in advance that you will be appending a large number of small strings to a StringBuffer.

`ensureCapacity()` has a general form.

void ensureCapacity(int capacity);

Here, **capacity** specifies the size of the buffer.

setLength()

To set the length of the buffer within a StringBuffer object, use `setLength()`. Its general form is shown here:

void setLength(int len);

Here, **len** specifies the length of the buffer. This value must be nonnegative.

When you increase the size of the buffer, null characters are added to the end of the existing buffer. If you call `setLength()` with a value less than the current value returned by `length()`, then the characters stored beyond the new length will be lost. The `setCharAtDemo` sample program in the following section uses `setLength()` to shorten a StringBuffer.

charAt() and setCharAt()

The value of a single character can be obtained from a StringBuffer via the `charAt()` method. You can set the value of a character within a StringBuffer using `setCharAt()`.

Their general forms are shown here:

char charAt(int where);

void setCharAt(int where, char ch);

For `charAt()`, '**where**' specifies the index of the character being obtained. For **setCharAt()**, '**where**' specifies the index of the character being set, and **ch** specifies the new value of that character.



For both methods, 'where' must be non-negative and must not specify a location beyond the end of the buffer.

getChars()

To copy a substring of a StringBuffer into an array, use the getChars() method. It has this general form:

void getChars(int sourceStart, int sourceEnd, char target[], int targetStart);

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from sourceStart through sourceEnd–1. The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

append()

The append() method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object. It has overloaded versions for all the **built-in** types and for **Object**. Here are a few of its forms:

StringBuffer append(String str);

StringBuffer append(int num);

StringBuffer append(Object obj);

String.valueOf() is called for each parameter to obtain its string representation. The result is appended to the current StringBuffer object. The buffer itself is returned by each version of append(). This allows subsequent calls to be chained together.

The append() method is most often called when the + operator is used on String objects. Java automatically changes modifications to a String instance into similar operations on a StringBuffer instance. Thus, a concatenation invokes append() on a StringBuffer object. After the concatenation has been performed, the compiler inserts a call to toString() to turn the modifiable StringBuffer back into a constant String. All of this may seem unreasonably complicated. Why not just have one string class and have it behave more or less like StringBuffer? The answer is performance. There are many optimizations that the Java run time can make knowing that String objects are immutable. Thankfully, Java hides most of the complexity of conversion between Strings and StringBuffers. Actually, many programmers will never feel the need to use StringBuffer directly and will be able to express most operations in terms of the + operator on String variables.

insert()

The insert() method inserts one string into another. It is overloaded to accept values of all the simple types, plus Strings and Objects. Like append(), it calls String.valueOf() to obtain the string



representation of the value it is called with. This string is then inserted into the invoking StringBuffer object. These are a few of its forms:

StringBuffer insert(int index, String str);

StringBuffer insert(int index, char ch);

StringBuffer insert(int index, Object obj);

Here, index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

reverse()

You can reverse the characters within a StringBuffer object using reverse(), as shown here:

```
StringBuffer reverse( );
```

This method returns the reversed object on which it was called.

delete() and deleteCharAt()

Java 2 adds to StringBuffer the ability to delete characters using the methods delete() and deleteCharAt(). These methods are shown here:

StringBuffer delete(int startIndex, int endIndex);

StringBuffer deleteCharAt(int loc);

The delete() method deletes a sequence of characters from the invoking object. Here, startIndex specifies the index of the first character to remove, and endIndex specifies an index one past the last character to remove. Thus, the substring deleted runs from startIndex to endIndex-1. The resulting StringBuffer object is returned. The deleteCharAt() method deletes the character at the index specified by loc. It returns the resulting StringBuffer object.

replace()

Another new method added to StringBuffer by Java 2 is replace(). It replaces one set of characters with another set inside a StringBuffer object. Its signature is shown here:

StringBuffer replace(int startIndex, int endIndex, String str);

The substring being replaced is specified by the indexes startIndex and endIndex. Thus, the substring at startIndex through endIndex-1 is replaced. The replacement string is passed in **str**. The resulting StringBuffer object is returned.

substring()



Java 2 also adds the `substring()` method, which returns a portion of a `StringBuffer`. It has the following two forms:

`String substring(int startIndex);`

`String substring(int startIndex, int endIndex);`

The first form returns the substring that starts at `startIndex` and runs to the end of the invoking `StringBuffer` object. The second form returns the substring that starts at `startIndex` and runs through `endIndex-1`. These methods work just like those defined for `String` that were described earlier.

Summary

Here are the key takeaways:

- Each time you need an altered version of an existing string, a new `String` object is created that contains the modifications.
- Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language.
- Both the `String` and `StringBuffer` classes are defined in `java.lang` package.