STRING HANDLING

- in Java a string is a sequence of characters
- Java implements strings as objects of type **String**.

Implementing strings as built-in objects- Advantages

- Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string.
- **String** objects can be constructed using number of ways. The easiest is to use a statement like this. **String mystring="This is a test";**

Important Point

- **String** objects are immutable
- Java provides two options: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.
- The **String, StringBuffer**, and **StringBuilder** classes are defined in **java.lang**. Thus, they are available to all programs automatically.
- **String, StringBuffer**, and **StringBuilder** declared as **final**
- All three implement the **CharSequence** interface.

The String Constructors

- The **String** class supports several constructors. To create an empty **String**, call the default constructor.For example, **String s=new String();** will create an instance of **String** with no characters in it.
- To create a **String** initialized by an array of characters, use the constructor **String(char *chars*[ ])**

  Example

  char chars[]={'a','b','c','d'};
  String s=new String(chars);
  or
  String s=new String("This is a test string");

  This constructor initializes **s** with the string "abc".
- You can specify a subrange 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 subrange begins, and *numChars* specifies the number of characters to use.

  example:

  char chars[]={'a','b','c','d','e','f'};
  String s=new String(chars,2,3);

Program

```
public class StringTest
{
  public static void main(String[] args)   {
                char chars[]={'a','b','c','d','e','f'};
                String s=new String(chars,2,3);
                System.out.print(s);

  }
}
```

Output

```
F:\Javapgms>java StringTest
cde
```

- You can construct a **String** object using another **String** object using the constructor String(String *strObj*) Here, *strObj* is a **String** object.

Example Program

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

```
    {
            char chars[]={'a','b','c','d','e','f'};
            String s1=new String(chars,2,3);
            System.out.println("s1="+s1);
            String s2=new String(s1);
            System.out.println("s2="+s2);
    }
}
```

Output

```
F:\Javapgms>java StringTest
s1=cde
s2=cde
```

16 bit char array and 8 bit String

Java's **char** type uses 16 bits to represent the basic 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.

Two forms are shown here:

String(byte *chrs*[ ])

String(byte *chrs*[ ], int *startIndex*, int *numChars*)

Here, *chrs* 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.

Example

```
public class StringTest1
{
  public static void main(String[] args)
  {
            byte ascii[]={65,66,67,68,69,70};
            String s1=new String(ascii);
            System.out.println("s1="+s1);
            String s2=new String(ascii,2,3);
            System.out.println("s2="+s2);
  }
}
```

Output

```
F:\Javapgms>java StringTest1
s1=ABCDEF
s2=CDE
```

- You can construct a **String** from a **StringBuffer** by using the constructor
  String(StringBuffer *strBufObj*)
- You can construct a **String** from a **StringBuilder** by using this constructor
  String(StringBuilder *strBuildObj*)
- You can construct a **String** using a constructor supports the extended Unicode character set
  String(int *codePoints*[ ], int *startIndex*, int *numChars*)

String Length

The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method

```
int length( )
            char chars[]={'a','b','c','d','e','f'};
            String s1=new String(chars);
            System.out.println("s1="+s1);
            System.out.println("s1.length="+s1.length());
```

You will get the following output

s1=abcdef
s1.length=6

Special String Operations

String Literals

The statement like this.

String mystring=”This is a test”; is 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.

Concatenation

- **+** operator, which concatenates two strings, producing a **String** object as the result. This allows you to chain together a series of **+** operations.
- **Example**

```
String age="9";
  String s="He is "+age+" years old";
  System.out.println(s);
```

Output

He is 9 years old

- when you want to create very long strings use this series of + operations

  Example

```
String s="This is a very "+
“very long string”+
"in java concatenated using +";
```

String Concatenation with Other Data Types

You can concatenate strings with other types of data. For example

```
int age="9";
  String s="He is "+age+" years old";
  System.out.println(s);
```

Output

He is 9 years old

You will get the same output. This is because the **int** value in **age** is automatically converted into its string representation within a **String** object. This string is then concatenated as before. The compiler will convert an operand to its string equivalent whenever the other operand of the **+** is an instance of **String**.

Example

```
String s=”four”+2+2;
System.out.println(s);
```

What will be the output?

String Conversion and toString( )

toString( ) method

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 them to be fully integrated into Java’s programming environment. For example, they can be used in **print( )** and **println( )** statements and in concatenation expressions. The following program demonstrates this by overriding **toString( )** for the **Box** class.

```
// Override toString() for Box class.
class Box {
  double width;
  double height;
  double depth;

  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  public String toString() {
    return "Dimensions are " + width + " by " +
           depth + " by " + height + ".";
  }
}

class toStringDemo {
  public static void main(String args[]) {
    Box b = new Box(10, 12, 14);
    String s = "Box b: " + b; // concatenate Box object

    System.out.println(b); // convert Box to string
    System.out.println(s);
  }
}
```

As you can see, **Box**'s **toString( )** method is automatically invoked when a **Box** object is used in a concatenation expression or in a call to **println( )**.

```
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
```

Character Extraction

The **String** class provides a number of ways in which characters can be extracted from a **String** object. Like arrays, the string indexes begin at zero.

charAt( )

- returns the character at the specified location 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( )**

For example,

String s2="hello";

System.out.println(s2.charAt(1));

Output

e

Example

```
char ch;
ch="abc".charAt(1);
System.out.println(ch);
```

Output

b

getChars( )

- to extract more than one character at a time

general form:

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

- *sourceStart* specifies the index of the beginning of the substring
- *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*.

Example

```
String s3="hello how are you";
char buf[]=new char [10];
s3.getChars(6,10,buf,0);
System.out.println(buf);
```

Output

how

getBytes( )

- it uses the default character-to-byte conversions provided by the platform.

general form:

byte[ ] getBytes( )

- **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.

general form:

char[ ] toCharArray( )

String Comparison

The **String** class includes a number of methods that compare strings or substrings within strings.

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.

Example

```
String c1="hello";
String c2="hello";
String c3="HELLO";
if(c1.equals(c2))
System.out.println("c1==c2");
else
System.out.println("c1!=c2");
if(c1.equalsIgnoreCase(c3))
System.out.println("c1==c3");
else
System.out.println("c1!=c3");
```

Output

c1==c2

c1==c3

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.
- 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*.
- Themindex 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( )

- The **startsWith( )** method determines whether a given **String** begins with a specified string.
- **endsWith( )** determines whether the **String** in question ends with a specified string.
- 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”);
Output: true for both

A second form of **startsWith( )**,

- Have to specify a starting point:
- boolean startsWith(String *str*, int *startIndex*)
- Here, *startIndex* specifies the index into the invoking string at which point the search will begin.

Foobar.startsWith(“bar”,3);
Output: true

equals( ) Versus ==
the **equals( )** method compares thecharacters inside a **String** object.
The **==** operator compares two object references to see whether they refer to the same instance.
Example

```
// 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));
  }
}
```

```
Hello equals Hello -> true
Hello == Hello -> false
```

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.

compareTo( )

- to check string 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 method **compareTo( )** serves this purpose. It is specified by the **Comparable<T>** interface, which **String** implements.

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 *str*.
Greater than zero	The invoking string is greater than *str*.
Zero	The two strings are equal.

Here is a sample program that sorts an array of strings. The program uses **compareTo( )** to determine sort ordering for a bubble sort:

```
// A bubble sort for Strings.
class SortString {
  static String arr[] = {
    "Now", "is", "the", "time", "for", "all", "good", "men",
    "to", "come", "to", "the", "aid", "of", "their", "country"
  };
  public static void main(String args[]) {
    for(int j = 0; j < arr.length; j++) {
      for(int i = j + 1; i < arr.length; i++) {
        if(arr[i].compareTo(arr[j]) < 0) {
          String t = arr[j];
          arr[j] = arr[i];
          arr[i] = t;
        }
      }
      System.out.println(arr[j]);
    }
  }
}
```

The output of this program is the list of words:

```
Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to
```

- As you can see from the output of this example, **compareTo( )** takes into account uppercase and lowercase letters. The word "Now" came out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.
- If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase( )**

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(char *ch*)
To search for the last occurrence of a character, use
int lastIndexOf(char *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(char *ch*, int *startIndex*)
int lastIndexOf(char *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.
The following example shows how to use the various index methods to search inside of a **String**

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
  public static void main(String args[]) {
    String s = "Now is the time for all good men " +
               "to come to the aid of their country.";

    System.out.println(s);
    System.out.println("indexOf(t) = " +
                       s.indexOf('t'));
    System.out.println("lastIndexOf(t) = " +
                       s.lastIndexOf('t'));
    System.out.println("indexOf(the) = " +
                       s.indexOf("the"));
    System.out.println("lastIndexOf(the) = " +
                       s.lastIndexOf("the"));
    System.out.println("indexOf(t, 10) = " +
                       s.indexOf('t', 10));
    System.out.println("lastIndexOf(t, 60) = " +
                       s.lastIndexOf('t', 60));
    System.out.println("indexOf(the, 10) = " +
                       s.indexOf("the", 10));
    System.out.println("lastIndexOf(the, 60) = " +
                       s.lastIndexOf("the", 60));
  }
}
```

Here is the output of this program:

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

Modifying a String

substring( )

You can extract a substring using **substring( )**. It has two forms.

- 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.

- 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.

Example

```
String s="This is a test string";
String sub="is";
System.out.println(s.substring(5));
System.out.println(s.substring(5,10));
```

Output

is a test string

is a

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,

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

replace( )

The **replace( )** method has two forms. The first 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,

puts the string "Hewwo" into **s**.

The second form of **replace( )** replaces one character sequence with another. It has this general form:

String replace(CharSequence *original*, CharSequence *replacement*)

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:

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 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. Nonalphabetical characters, such as digits, are unaffected. Here are the simplest forms of these methods:
String toLowerCase( )
String toUpperCase( )
Both methods return a **String** object that contains the uppercase or lowercase equivalent of the invoking **String**. The default locale governs the conversion in both cases.
Here is an example that uses **toLowerCase( )** and **toUpperCase( )**:
The output produced by the program is shown here:
One other point: Overloaded versions of **toLowerCase( )** and **toUpperCase( )** that let you specify a **Locale** object to govern the conversion are also supplied. Specifying the locale can be quite important in some cases and can help internationalize your application.

Joining Strings

JDK 8 added a new method to **String** called **join( )**. It is used to concatenate two or more strings, separating each string with a delimiter, such as a space or a comma. It has two forms. Its first is shown here:
static String join(CharSequence *delim*, CharSequence … *strs*)
Here, *delim* specifies the delimiter used to separate the character sequences specified by *strs*. Because **String** implements the **CharSequence** interface, *strs* can be a list of strings. (See Chapter 18 for information on **CharSequence**.) The following program demonstrates this version of **join( )**:
The output is shown here:
In the first call to **join( )**, a space is inserted between each string. In the second call, the delimiter is a comma followed by a space. This illustrates that the delimiter need not be just a single character.
The second form of **join( )** lets you join a list of strings obtained from an object that implements the **Iterable** interface. **Iterable** is implemented by the Collections Framework classes described in Chapter 19, among others. See Chapter 18 for information on **Iterable**.

Additional String Methods

In addition to those methods discussed earlier, **String** has many other methods. Several are summarized in the following table:

Method	Description
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1.
boolean contains(CharSequence *str*)	Returns **true** if the invoking object contains the string specified by *str*. Returns **false** otherwise.
boolean contentEquals(CharSequence *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
boolean contentEquals(StringBuffer *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
static String format(String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. (See Chapter 19 for details on formatting.)
static String format(Locale *loc*, String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. Formatting is governed by the locale specified by *loc*. (See Chapter 19 for details on formatting.)

boolean isEmpty( )	Returns **true** if the invoking string contains no characters and has a length of zero.
boolean matches(string *regExp*)	Returns **true** if the invoking string matches the regular expression passed in *regExp*. Otherwise, returns **false**.
int offsetByCodePoints(int *start*, int *num*)	Returns the index within the invoking string that is *num* code points beyond the starting index specified by *start*.
String replaceFirst(String *regExp*, String *newStr*)	Returns a string in which the first substring that matches the regular expression specified by *regExp* is replaced by *newStr*.
String replaceAll(String *regExp*, String *newStr*)	Returns a string in which all substrings that match the regular expression specified by *regExp* are replaced by *newStr*.
String[ ] split(String *regExp*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*.

String[ ] split(String *regExp*, int *max*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*. The number of pieces is specified by *max*. If *max* is negative, then the invoking string is fully decomposed. Otherwise, if *max* contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If *max* is zero, the invoking string is fully decomposed, but no trailing empty strings will be included.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is implemented by **String**.

StringBuffer

StringBuffer supports a modifiable string. As you know, **String** represents fixedlength, immutable character sequences. In contrast, **StringBuffer** represents growable and writable 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 preallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

StringBuffer defines these four constructors:

StringBuffer( )

- The default constructor (the one with no parameters) reserves room for 16 characters without reallocation

StringBuffer(int *size*)

- accepts an integer argument that explicitly sets the size of the buffer

StringBuffer(String *str*)

- accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation

StringBuffer(CharSequence *chars*)

- constructor creates an object that contains the character sequence contained in *chars* and reserves room for 16 more characters.

Specilaity of StringBuffer

StringBuffer allocates room for 16 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. general forms

int length( )

int capacity( )

Example

```
public class StringBufferTest1
{
  public static void main(String[] args)
  {
                StringBuffer sb=new StringBuffer("Hello");
                System.out.println("sb="+sb);
```

```
                System.out.println("sb.lenght="+sb.length());
        System.out.println("sb.capacity="+sb.capacity());
    }
}
```

Output

```
F:\Javapgms>java StringBufferTest1
sb=Hello
sb.lenght=5
sb.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 preallocate 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 this general form:

void ensureCapacity(int *minCapacity*)

Here, *minCapacity* specifies the minimum size of the buffer. (A buffer larger than *minCapacity* may be allocated for reasons of efficiency.)

setLength( )

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

void setLength(int *len*)

Here, *len* specifies the length of the string. This value must be nonnegative. When you increase the size of the string, null characters are added to the end. 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 nonnegative and must not specify a location beyond the end of the string.

The following example demonstrates **charAt( )** and **setCharAt( )**:

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 several overloaded versions. Here are a few of its forms:

StringBuffer append(String *str*)
StringBuffer append(int *num*)
StringBuffer append(Object *obj*)
The string representation of each parameter is obtained, often by calling **String.valueOf( )**. 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, as shown in the following example:

insert( )

The **insert( )** method inserts one string into another. It is overloaded to accept values of all the primitive types, plus **String**S, **Object**S, and **CharSequence**S. Like **append( )**, it obtains 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.
The following sample program inserts "like" between "I" and "Java":

reverse( )

You can reverse the characters within a **StringBuffer** object using **reverse( )**, shown here:
StringBuffer reverse( )
This method returns the reverse of the object on which it was called. The following program demonstrates **reverse( )**:

delete( ) and deleteCharAt( )

You can delete characters within a **StringBuffer** by 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.
Here is a program that demonstrates the **delete( )** and **deleteCharAt( )** methods:

replace( )

You can replace one set of characters with another set inside a **StringBuffer** object by calling **replace( )**. 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.
The following program demonstrates **replace( )**:

substring( )

You can obtain a portion of a **StringBuffer** by calling **substring( )**. 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.

Additional StringBuffer Methods

In addition to those methods just described, **StringBuffer** supplies others. Several

are summarized in the following table:

Method	Description
StringBuffer appendCodePoint(int *ch*)	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned.
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1.
int indexOf(String *str*)	Searches the invoking **StringBuffer** for the first occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int indexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the first occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*)	Searches the invoking **StringBuffer** for the last occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the last occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int offsetByCodePoints(int *start*, int *num*)	Returns the index within the invoking string that is *num* code points beyond the starting index specified by *start*.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is implemented by **StringBuffer**.
void trimToSize( )	Requests that the size of the character buffer for the invoking object be reduced to better fit the current contents.

The following program demonstrates **indexOf( )** and **lastIndexOf( )**:

StringBuilder

StringBuilder is similar to **StringBuffer** except for one important difference: it is not synchronized, which means that it is not thread-safe. The advantage of **StringBuilder** is faster performance. However, in cases in which a mutable string will be accessed by multiple threads, and no external synchronization is employed, you must use **StringBuffer** rather than **StringBuilder**.