RV College of Engineering®

Go, change the world

String Handling

Department of Computer Science,
R V College of Engineering,
Bengaluru-59.

Contents

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.

Frequently, you will 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);
```

String Constructors

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. Here is an example:

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

This initializes **s** with the characters **cde**.

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.

String Constructors

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.

String Constructors

Even though 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 subrange.

In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform.

String Constructors

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, you will often want to use the default encoding provided by the platform.

String Length

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

String Literals

```
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."

String Concatenation with Other Data Types

Be careful when you mix other types of operations with string concatenation expressions, however. You might get surprising results. Consider the following:

```
String s = "four: " + 2 + 2;
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".

toString() Method

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 will 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 them to be fully integrated into Java’s programming environment.
or example, they can be used in **print( )** and **println( )** statements and in concatenation expressions.

toString() Method

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);
} }
```

The output of this program is shown here:

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

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( )**.

equals()

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().

```
// 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
```

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
```

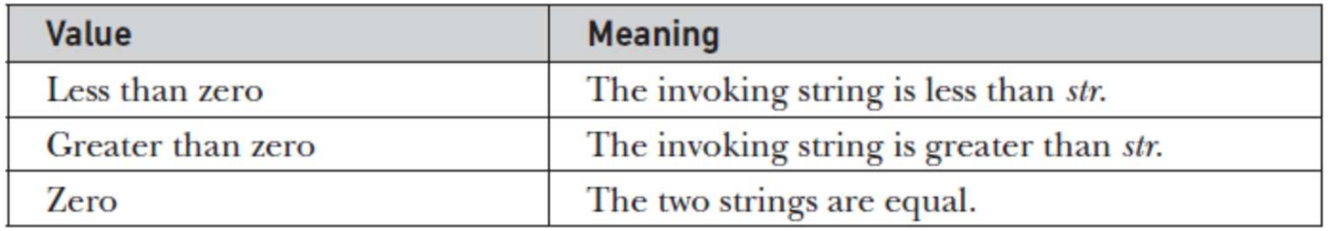
compareTo( )

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 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.

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

Searching Strings

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.

Searching Strings

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
```

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( )**:

```
// Demonstrate toUpperCase() and toLowerCase().
class ChangeCase {
public static void main(String args[])
{
String s = "This is a test.";
System.out.println("Original: " + s);
String upper = s.toUpperCase();
String lower = s.toLowerCase();

System.out.println("Uppercase: " + upper);
System.out.println("Lowercase: " + lower);
}
}
```

The output produced by the program is shown here:

```
Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.
```

Modifying a String (substring() method)

Because **String** objects are immutable, whenever you want to modify a **String**, you must either copy it into a **StringBuffer** or **StringBuilder**, or use a **String** method that constructs a new copy of the string with your modifications complete.
A sampling of these methods are described here.

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.

Concat() and replace() methods

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 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,

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

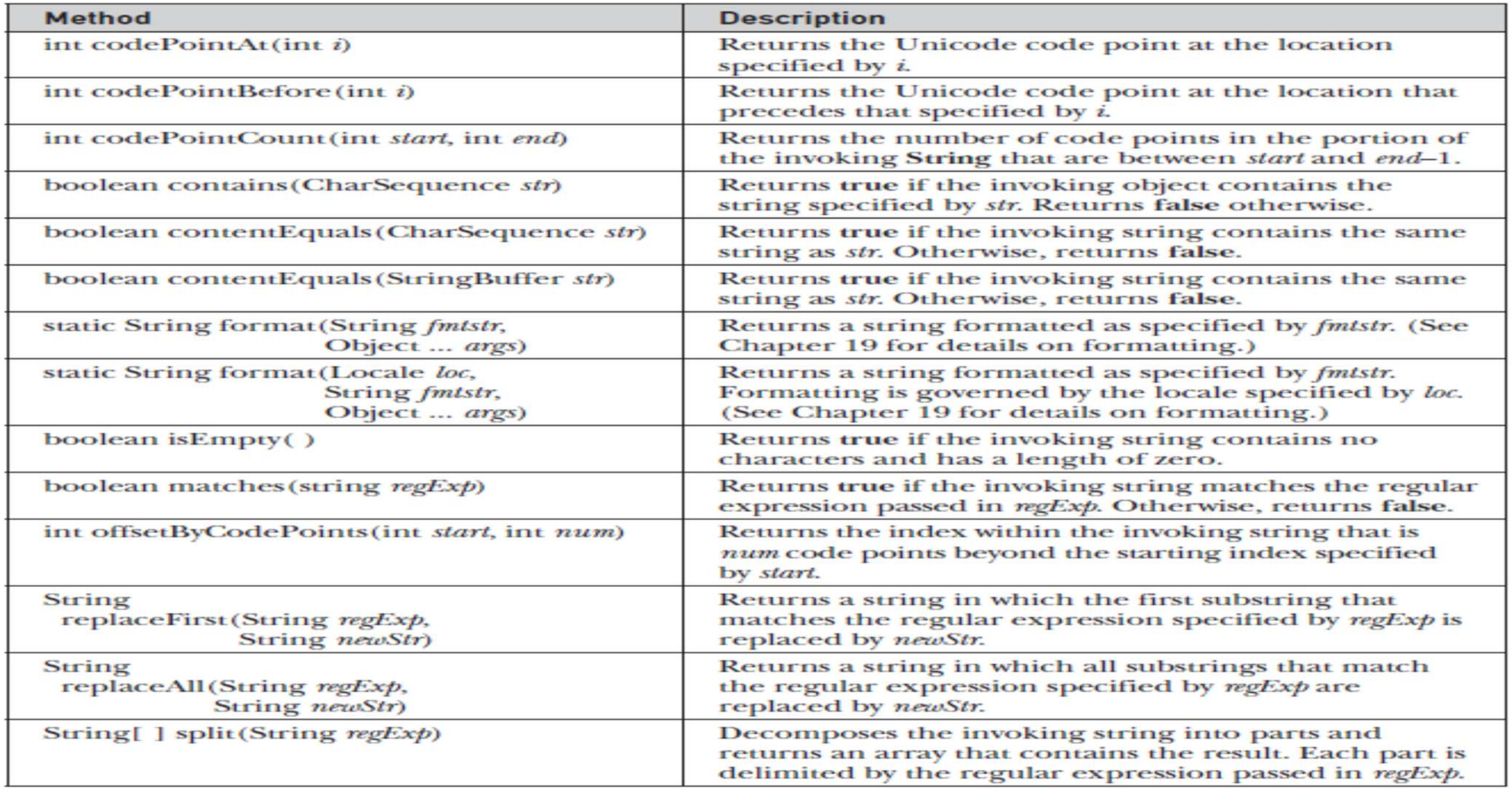
Trim()

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:
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.

Additional String Methods



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*.

THANK YOU