# Unit 13: String handling

***Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.***

- As is the case in most other programming languages, in Java a string is a sequence of characters. But, unlike some 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 a number of ways, making it easy to obtain a string when needed.
- Somewhat unexpectedly, 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, Java provides two options: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.

***Note: The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes.***

## 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.

- 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);
```

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. 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);
  }
}
```

- 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. 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);
  }
}
```

### 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());
```

### 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);
```

- 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);
  }
}
```

### String Concatenation with Other Data Types

You can concatenate strings with other types of data. For example, consider this slightly different version of the earlier example:

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

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);
```

### 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 primitive types and for type **Object**. For the primitive 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.
- 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. 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);
  }
}
```

Output :

```
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( ).**

## Character Extraction

The **String** class provides a number of ways in which characters can be extracted from a **String** object. Several are 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**.

//Example

```
public class CharAtExample {

    public static void main(String[] args) {

        String str = "Where are you going";

        int count = 0;

        for (int i=0; i<str.length(); i++) {

            if(str.charAt(i) == 'e') {

                count++;

            }

        }

        System.out.println("Frequency of e is: "+count);

    }

}
```

**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.

The following program demonstrates **getChars( ):**

```
class getCharsDemo {
  public static void main(String args[]) {
    String s = "This is a demo of the getChars method.";
    int start = 10;
    int end = 14;
    char buf[] = new char[end - start];

    s.getChars(start, end, buf, 0);
    System.out.println(buf);
  }
}
```

**getBytes( )**

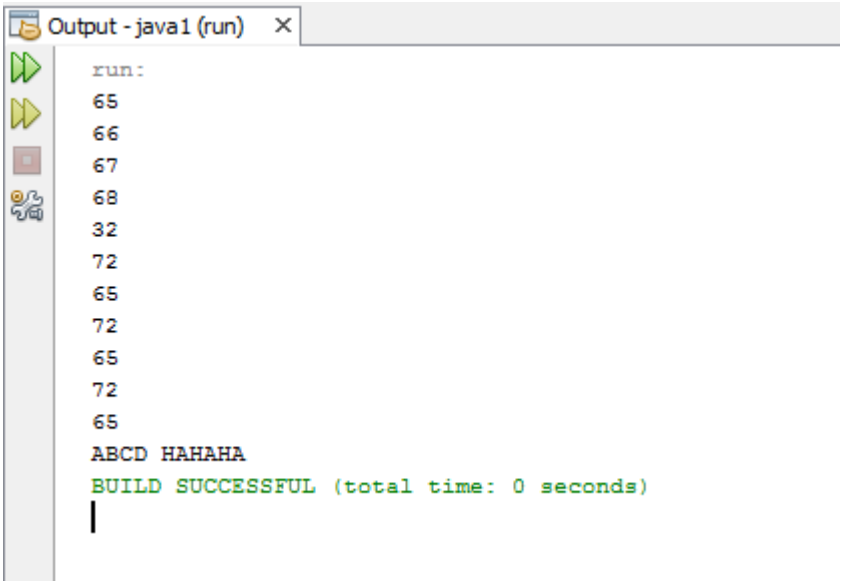
There is an alternative to **getChars( )** that stores the characters in an array of bytes. This method is called **getBytes( ),** and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

```
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.

```
package stringhandlingdemo;

public class GetBytesExample {

    public static void main(String[] args) {

        String s1 = "ABCD HAHAHA";

        byte[] barr = s1.getBytes();

        for(int i=0;i<barr.length;i++){

            System.out.println(barr[i]);

        }

        // Getting string back
```

```
        String s2 = new String(barr);
        System.out.println(s2);
    }  }
```




**toCharArray( )**

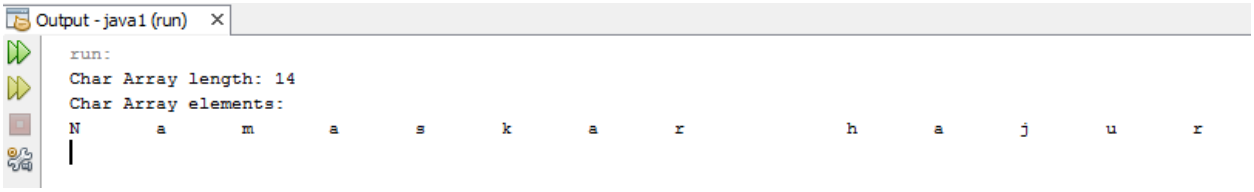
If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( ).** It returns an array of characters for the entire string. It has this general form:

**`char[ ] toCharArray( )`**

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

```
package stringhandlingdemo;

public class StringToCharArrayDemo{
    public static void main(String[] args) {
        String s1 = "Namaskar hajur";
        char[] ch = s1.toCharArray();
        int len = ch.length;
        System.out.println("Char Array length: " + len);
        System.out.println("Char Array elements: ");
```

```
        for (int i = 0; i < len; i++) {
            System.out.print(ch[i]);
            System.out.print("\t");
        }
    }
}
```




## 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:

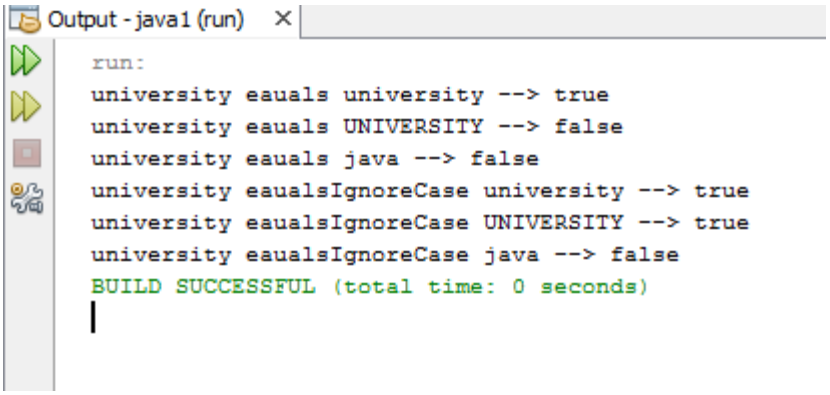
  **`boolean equalsIgnoreCase(String str)`**

  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.

**`//to demonstrate equals() and equalsIgnoreCase( ) methods`**

```
package stringhandlingdemo;
public class EqualsDemo{
```

```
public static void main(String args[]){
String s1="university";
String s2="university";
String s3="UNIVERSITY";
String s4="java";
// this prints true because content and case is same
System.out.println(s1+" eauals "+ s2+ " --> "+ s1.equals(s2));
// this prints false because case is not same
System.out.println(s1+" eauals "+ s3+ " --> "+ s1.equals(s3));
// this prints false because content is not same
System.out.println(s1+" eauals "+ s4+ " --> "+ s1.equals(s4));
// this prints true because content and case both are same
System.out.println(s1+" eaualsIgnoreCase "+ s2+ " -->
"+s1.equalsIgnoreCase(s2));
// this prints true because case is ignored
System.out.println(s1+" eaualsIgnoreCase "+ s3+ " --> "+
s1.equalsIgnoreCase(s3));
// this prints false because content is not same
System.out.println(s1+" eaualsIgnoreCase "+ s4+ " -->
"+s1.equalsIgnoreCase(s4));
}
}
```



.

**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:

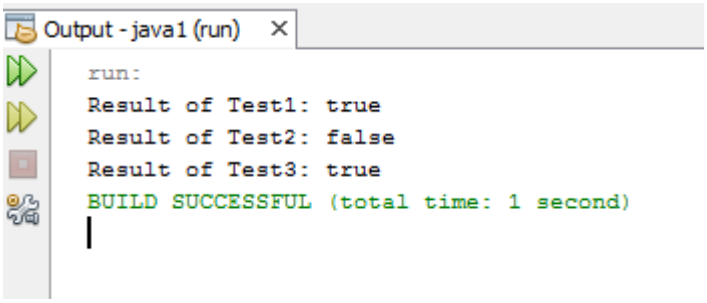
boolean regionMatches(int *startIndex*, String *str2*,
int *str2StartIndex*, int *numChars*)

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

```
//example program

package stringhandlingdemo;

public class RegionMatchesExample {

 public static void main(String args[]){

       String str1 = new String("Hello, How are you");

       String str2 = new String("How");

       String str3 = new String("HOW");


       System.out.print("Result of Test1: " );
```

```
        System.out.println(str1.regionMatches(7, str2, 0, 3));

        System.out.print("Result of Test2: " );
        System.out.println(str1.regionMatches(7, str3, 0, 3));

        System.out.print("Result of Test3: " );
    System.out.println(str1.regionMatches(true, 7, str3, 0, 3));
    }
}
```



**startsWith( ) and endsWith( )**

- **String** defines two methods 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,

```
"Lalitpur".endsWith("pur");
```

and

```
"Lalitpur".startsWith("Lalit");
```

are both **true**

A second form of **startsWith( ),** shown here, lets you 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. For example,

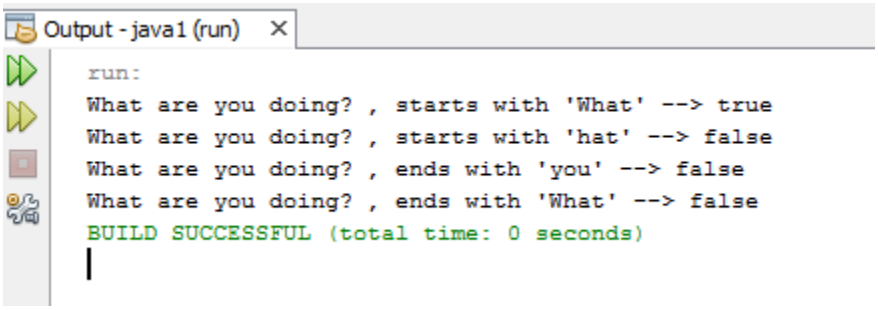
```
"Foobar".startsWith("bar", 3)
```

returns **true**

**//Example Program**

```
public class StartsWithEndsWith {

 public static void main(String[] args) {

   String str = "What are you doing?";

        System.out.println(str+ " , starts with 'What' --> "+
str.startsWith("What"));

        System.out.println(str+ " , starts with 'hat' --> "+
str.startsWith("hat"));
```

```
        System.out.println(str+ " , ends with 'you' --> "+
str.startsWith("you"));

        System.out.println(str+ " , ends with 'What' --> "+
str.startsWith("doing?"));

    }

}
```

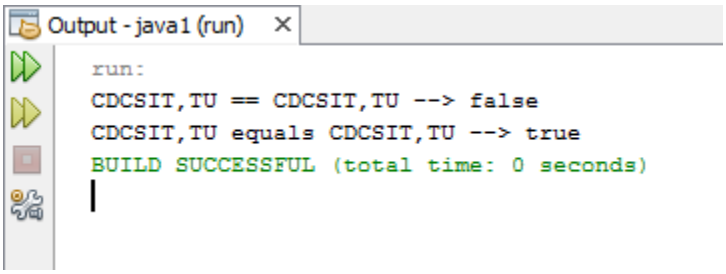



**equals( ) Versus ==**

- 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

```
//Example
package stringhandlingdemo;
public class EqualsNotEqualTo {
   public static void main(String []args) {
      String s1 = "CDCSIT,TU";
      String s2 = new String (s1);
      System.out.println(s1 + " == "+ s2 + " --> "+(s1==s2));
      System.out.println(s1 + " equals " +s2+ " --> "
+s1.equals(s2));
   }
}
```

The variable **s1** refers to the String instance created by **"CDCSIT,TU".** 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




**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:

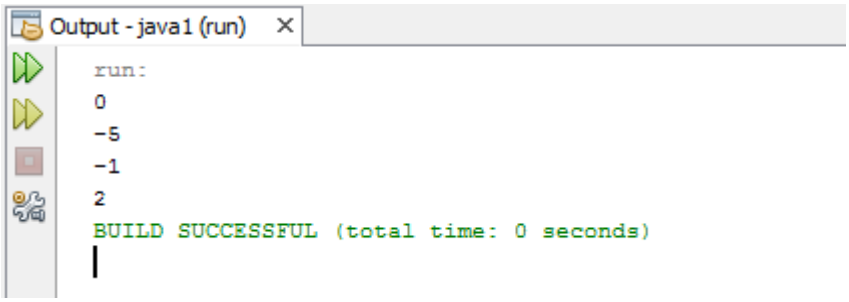
| Value | Meaning |
|---|---|
| Less than zero | The invoking string is less than *str*. |
| Greater than zero | The invoking string is greater than *str*. |
| Zero | The two strings are equal. |

→ If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase( ),** as shown here:

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

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

```
package stringhandlingdemo;
public class CompareToExample{
public static void main(String args[]){
String s1="hello";
String s2="hello";
String s3="meklo";
String s4="hemlo";
String s5="flag";
System.out.println(s1.compareTo(s2));//0 because both are equal
//-5 because "h" is 5 less than "m"
System.out.println(s1.compareTo(s3));
//-1 because "l" is less than "m"
System.out.println(s1.compareTo(s4));
//2 because "h" is 2  more than "f"
System.out.println(s1.compareTo(s5));
}
}
```




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

**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 searched.

- 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)**

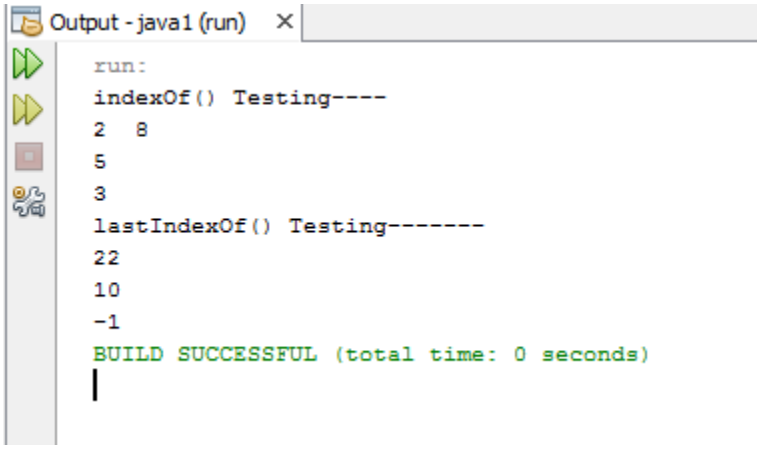
**int lastIndexOf(String str, int startIndex)**

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**.

//Example

```
package stringhandlingdemo;


public class IndexOfExample{
public static void main(String args[]){
String s1="this is index of example";
String s2 = "This is last index of example";
    System.out.println("indexOf() Testing----");
//passing substring
int index1=s1.indexOf("is");//returns the index of is substring
int index2=s1.indexOf("index");//returns the index of index
//substring
System.out.println(index1+"  "+index2);//2 8
```

```
//passing substring with from index
int index3=s1.indexOf("is",4);//returns the index of is
//substring after 4th index
System.out.println(index3);//5 i.e. the index of another is


//passing char value
int index4=s1.indexOf('s');//returns the index of s char value
System.out.println(index4);//3


    System.out.println("lastIndexOf() Testing-------");
int index5 = s2.lastIndexOf("ex");
System.out.println(index5);  //22
int index6=s2.lastIndexOf('s');//returns last index of 's' char
value
System.out.println(index6);//10
int index7 = s2.lastIndexOf("ple",10);
System.out.println(index7);//-1 if not found
}
}
```



## Modifying a String

- 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)
```

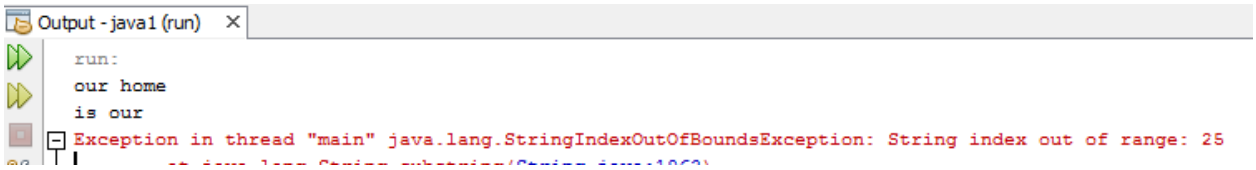
Here, **startIndex** specifies the index at which the substring will begin. This form returns a copy of the substring that begins at **startIndex** and runs to the end of the invoking string.

  - 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.

```
public class SubStringDemo {

    public static void main(String[] args) {

        String s1="Earth is our home";

          // Starts with 9 and goes to end

        String substr1 = s1.substring(9);
        System.out.println(substr1);

     // Starts from 6 and goes to 12

      String substr2 = s1.substring(6,12);
      System.out.println(substr2);

       String substr3 = s1.substring(13,25);//  Exception occurs

        System.out.println(substr3);

    }

}
```




**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");
```

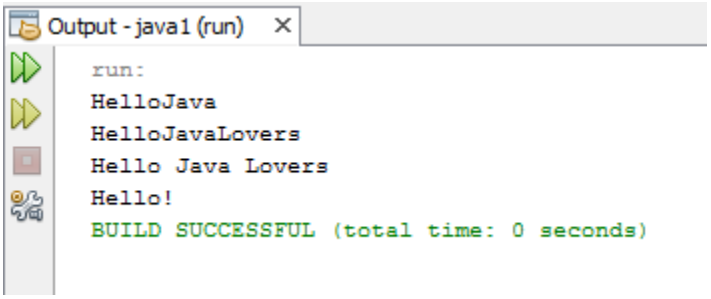
  puts the string "onetwo" into **s2.** It generates the same result as the following sequence

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

  //Example program

```
package stringhandlingdemo;

public class ConcatExample {

    public static void main(String[] args) {

        String str1 = "Hello";

        String str2 = "Java";

        String str3 = "Lovers";

        // Concatenating one string

        String str4 = str1.concat(str2);

        System.out.println(str4);

        // Concatenating multiple strings

        String str5 = str1.concat(str2).concat(str3);

        System.out.println(str5);

        //Concatenating Space among strings
```

```
            String str6 = str1.concat(" ").concat(str2).concat("
").concat(str3);
             System.out.println(str6);
             // Concatenating Special Chars
             String str7 = str1.concat("!");
             System.out.println(str7);
         }
     }
```




**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');
  ```

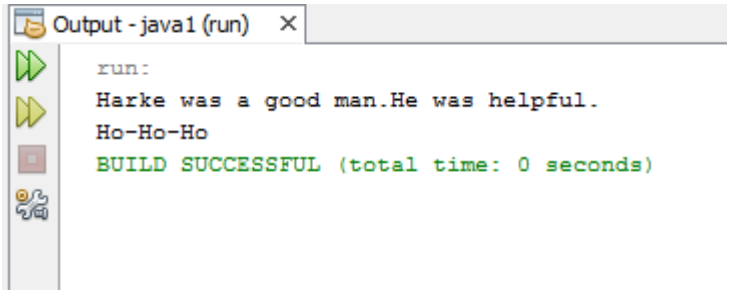
  puts the string "Hewwo" into **s**.

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

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

```
//Example Program


package stringhandlingdemo;
public class ReplaceExample {
public static void main(String args[]){
String s1="Harke is a good man.He is helpful.";
```

```
String s2 ="Ha-Ha-Ha";
//replaces all occurrences of substring "is" to "was"
String rs1=s1.replace("is","was");
//replaces all occurrences of char 'a' to 'o'
String rs2=s2.replace('a','o');
System.out.println(rs1);
System.out.println(rs2);
}
}
```




**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.

```
//Example
public class StringTrimDemo {
    public static void main(String[] args) {
        String s1 ="  hello guys how are you?     ";
        System.out.println("Length of string before trim =
"+s1.length());
        System.out.println(s1); //Without trim()
        String tr = s1.trim();
```

```
        System.out.println("Length of string after trim =
"+tr.length());
        System.out.println(tr); //With trim()
    }
}
```