# *Course: Object Based Modeling*
# *Code: CS-33105*
# *Branch: MCA-3*

## *Lecture 14: Strings and String Handling*

*Dr. J Sathish Kumar (JSK)*
*(Faculty & Coordinator)*

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

# Introduction

- **String** is probably the most commonly used class in Java's class library.
- The first thing to understand about strings is that every string you create is actually an object of type **String**.
- Even string constants are actually **String** objects.
- For example, in the statement
  *System.out.println("This is a String, too");*
  the string "This is a String, too" is a **String** object.
- Once you have created a **String** object, you can use it anywhere that a string is allowed.
  - For example, this statement displays **myString**:
    System.out.println(myString);
- Java defines one operator for **String** objects: **+**. It is used to concatenate two strings.
  - For example, this statement
    String myString = "I" + " like " + "Java.";

# Example #1

```
// Demonstrating Strings.
class StringDemo {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1 + " and " + strOb2;

    System.out.println(strOb1);

    System.out.println(strOb2);
    System.out.println(strOb3);
  }
}
```

The output produced by this program

```
First String
Second String
First String and Second String
```

# Strings

- The **String** class contains several methods that you can use. Here are a few.
- You can test two strings for equality by using **equals( )**.
  - boolean equals(*secondStr*)
- You can obtain the length of a string by calling the **length( )** method.
  - int length( )
- You can obtain the character at a specified index within a string by calling **charAt( )**.
  - char charAt(*index*)

# Example #2

```
// Demonstrating some String methods.
class StringDemo2 {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1;

    System.out.println("Length of strOb1: " +
                       strOb1.length());

    System.out.println("Char at index 3 in strOb1: " +
                       strOb1.charAt(3));

    if(strOb1.equals(strOb2))
      System.out.println("strOb1 == strOb2");
    else
      System.out.println("strOb1 != strOb2");

    if(strOb1.equals(strOb3))
      System.out.println("strOb1 == strOb3");
    else
      System.out.println("strOb1 != strOb3");
  }
}
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

# Example #3

```
// Demonstrate String arrays.
class StringDemo3 {
  public static void main(String args[]) {
    String str[] = { "one", "two", "three" };

    for(int i=0; i<str.length; i++)
      System.out.println("str[" + i + "]: " +
                         str[i]);
  }
}
```

Here is the output from this program:

```
str[0]: one
str[1]: two
str[2]: three
```

# 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 shown here:

  *String(char chars[ ])*
- Here is an example:

  *char chars[] = { 'a', 'b', 'c' };*
  *String s = new String(chars);*
- 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);*

# The String Constructors

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

*String(String strObj)*

```
// 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

```
Java
Java
```

Example #4

**String** class provides constructors that initialize a string when given a **byte** array.

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

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

Example #5

This program generates the following output:

```
ABCDEF
CDE
```

# Special String Operations

- 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.

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

Example #6

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

# Special String Operations

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

Example #7

This fragment displays

```
four: 22
```

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

Now **s** contains the string "four: 4".

# Character Extraction

- 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)*
- 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)*
- getBytes( )
  - **getBytes( )**, uses the default character-to-byte conversions provided by the platform.
  - Here is its simplest form:

    *byte[ ] getBytes( )*
- 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( )*

```
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);
  }
}
```

Example #8

Here is the output of this program:

```
demo
```

# 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( )**. The comparison is case-sensitive.
  - It has this general form:
    *boolean equals(Object str)*
  - To perform a comparison that ignores case differences
    *boolean equalsIgnoreCase(String str)*
- regionMatches( )
  - The **regionMatches( )** method compares a specific region inside a string with another specific region in another string.

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

# String Comparison

- 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)*
  -

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

and

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

are both **true**.

```
// 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
```

Example #9

# String Comparison

- 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.
- compareTo( )
  - It is specified by the **Comparable<T>** interface, which **String** implements.
  - It has this general form:

    *int compareTo(String str)*

| 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. |

Example #10

```
// 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 c

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

int compareToIgnoreCase(String *str*)

# 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.
- 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.

# Searching Strings

- 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*)

Example #11

```
// 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*)
  - String substring(int *startIndex*, int *endIndex*)
- concat( )
  - You can concatenate two strings using **concat( )**, shown here:
    String concat(String *str*)
- replace( )
  - It has two forms.
  - String replace(char *original*, char *replacement*)
  - String replace(CharSequence *original*, CharSequence *replacement*)

# Modifying a String

- 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**.

```
// Using trim() to process commands.
import java.io.*;

class UseTrim {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    String str;

    System.out.println("Enter 'stop' to quit.");
    System.out.println("Enter State: ");
    do {
      str = br.readLine();
      str = str.trim(); // remove whitespace

      if(str.equals("Illinois"))
        System.out.println("Capital is Springfield.");
      else if(str.equals("Missouri"))
        System.out.println("Capital is Jefferson City.");
      else if(str.equals("California"))
        System.out.println("Capital is Sacramento.");
      else if(str.equals("Washington"))
        System.out.println("Capital is Olympia.");
      // ...
    } while(!str.equals("stop"));
  }
}
```

Example #12

# Other String Functions

- Data Conversion Using valueOf( )
  - The **valueOf( )** method converts data from its internal format into a human-readable form.
  - 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*[ ])
- Changing the Case of Characters Within a String
  - String toLowerCase( )
  - String toUpperCase( )
- Joining Strings
  - It is used to concatenate two or more strings, separating each string with a delimiter, such as a space or a comma
  - static String join(CharSequence *delim*, CharSequence . . . *strs*)

| 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*. |

# Additional String Methods

# StringBuffer

- **StringBuffer** represents growable and writable character sequences.
- StringBuffer Constructors
  - StringBuffer( )
    StringBuffer(int *size*)
    StringBuffer(String *str*)
    StringBuffer(CharSequence *chars*)
- For example,
  - append( ) : The **append( )** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object.
  - StringBuffer append(String *str*)
    StringBuffer append(int *num*)
    StringBuffer append(Object *obj*)

```
// Demonstrate append().
class appendDemo {
  public static void main(String args[]) {
    String s;
    int a = 42;
    StringBuffer sb = new StringBuffer(40);

    s = sb.append("a = ").append(a).append("!").toString();
    System.out.println(s);
  }
}
```

The output of this example is shown here:

```
a = 42!
```

Example #13

# StringBuffer

- 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.
- insert( )
  - The **insert( )** method inserts one string into another.
  - StringBuffer insert(int *index*, String *str*)
    StringBuffer insert(int *index*, char *ch*)
    StringBuffer insert(int *index*, Object *obj*)
- reverse( )
  - You can reverse the characters within a **StringBuffer** object using **reverse( )**, shown here:
    StringBuffer 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*)
- 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*)

| 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. |

# Additional StringBuffer Methods