Exploring the String Class

By
M. BABY ANUSHA,
ASST.PROF IN CSE DEPT.,
RGUKT,NUZVID

STRINGS

- Strings represent a sequence of characters.
- The easiest way to represent a sequence of characters in JAVA is by using a character array.

```
char Array[ ] = new char [5];
```

Character arrays are not good enough to support the range of operations we want to perform on strings.

In JAVA strings are class objects and implemented using two classes:-

- ✓String
- ✓StringBuffer.

In JAVA strings are not a character array and is not NULL terminated.

DECLARING & INITIALISING

- Normally, objects in Java are created with the *new* keyword.

```
String name;
  name = new String("Craig");
```

OR

```
String name= new String("Craig");
```

- However, String objects can be created "implicitly":

```
String name;
name = "Craig";
```

The String Class

- String objects are handled specially by the compiler.
  - String is the only class which has "implicit" instantiation.
- The String class is defined in the **java.lang package.**
- Strings are immutable.
- The value of a String object can never be changed.
  - For mutable Strings, use the StringBuffer class.

DYNAMIC INITIALIZATION OF STRINGS

```
BufferedReader br =  new BufferedReader(new
  InputStreamReader(System.in));

    String city = br.readLine();
```

Give throws IOException beside function name

```
Scanner sc=new Scanner(System.in);
    String state=sc.nextLine();
   String state1=sc.next();
```

STRING CONCATENATION

- JAVA string can be concatenated using + operator.

```
String name="Ankita";
String surname="Karia";
System.out.println(name+"  "+surname);
```

STRING Arrays

- An array of strings can also be created

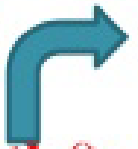
```
String cities [ ] = new String[5];
```

- Will create an array of CITIES of size 5 to hold string constants

String Methods

- The **String** class contains many useful methods for string-processing applications.
  - A **String** method is called by writing a **String** object, a dot, the name of the method, and a pair of parentheses to enclose any arguments
  - If a **String** method returns a value, then it can be placed anywhere that a value of its type can be used

```
String greeting = "Hello";
int count = greeting.length();
System.out.println("Length is " + greeting.length());
```



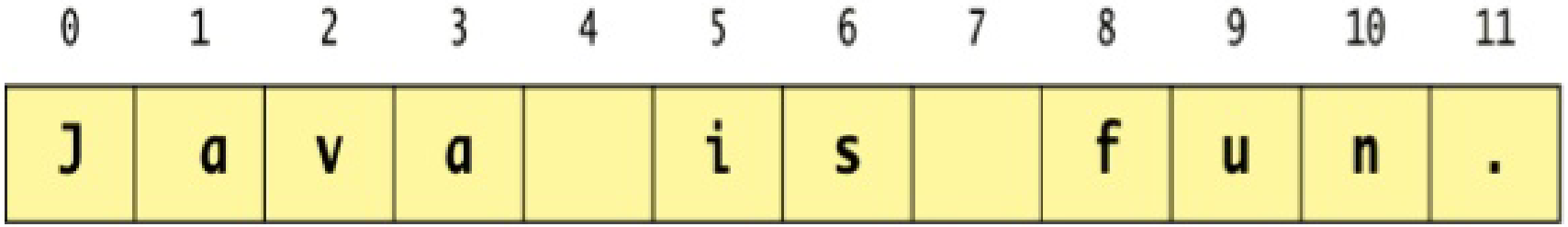
String method

  - Always count from zero when referring to the *position* or *index* of a character in a string

String Indexes

Display 1.5 String Indexes

The 12 characters in the string `"Java is fun."` have indexes 0 through 11.



Notice that the blanks and the period count as characters in the string.

Some Methods in the Class `String` (Part 1 of 8)

Display 1.4 Some Methods in the Class `String`

```
int length()
```

Returns the length of the calling object (which is a string) as a value of type `int`.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.length()` returns 6.

```
boolean equals(Other_String)
```

Returns `true` if the calling object string and the *Other_String* are equal. Otherwise, returns `false`.

EXAMPLE

After program executes `String greeting = "Hello";`
`greeting.equals("Hello")` returns `true`
`greeting.equals("Good-Bye")` returns `false`
`greeting.equals("hello")` returns `false`

Note that case matters. `"Hello"` and `"hello"` are not equal because one starts with an uppercase letter and the other starts with a lowercase letter.

(continued)

Some Methods in the Class `String` (Part 2 of 8)

Display 1.4 Some Methods in the Class `String`

`boolean equalsIgnoreCase(`*Other_String*`)`

Returns `true` if the calling object string and the *Other_String* are equal, considering uppercase and lowercase versions of a letter to be the same. Otherwise, returns `false`.

EXAMPLE

After program executes `String name = "mary!";`
`greeting.equalsIgnoreCase("Mary!")` returns `true`

`String toLowerCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to lowercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toLowerCase()` returns `"hi mary!"`.

(continued)

Some Methods in the Class `String` (Part 3 of 8)

Display 1.4 Some Methods in the Class `String`

`String toUpperCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to uppercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toUpperCase()` returns `"HI MARY!"`.

`String trim()`

Returns a string with the same characters as the calling object string, but with leading and trailing white space removed. Whitespace characters are the characters that print as white space on paper, such as the blank (space) character, the tab character, and the new-line character `'\n'`.

EXAMPLE

After program executes `String pause = "  Hmm  ";`
`pause.trim()` returns `"Hmm"`.

(continued)

Some Methods in the Class `String` (Part 4 of 8)

Display 1.4 Some Methods in the Class `String`

`char charAt(`*Position*`)`

Returns the character in the calling object string at the *Position*. Positions are counted 0, 1, 2, etc.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.charAt(0)` returns `'H'`, and
`greeting.charAt(1)` returns `'e'`.

`String substring(`*Start*`)`

Returns the substring of the calling object string starting from *Start* through to the end of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2)` returns `"cdefG"`.

(continued)

Some Methods in the Class `String` (Part 5 o

Display 1.4 **Some Methods in the Class `String`**

`String substring(`*Start*`, `*End*`)`

Returns the substring of the calling object string starting from position *Start* through, but not including, position *End* of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned, but the character at position *End* is not included.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2, 5)` returns `"cde"`.

`int indexOf(`*A_String*`)`

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns `-1` if *A_String* is not found.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.indexOf("Mary")` returns 3, and
`greeting.indexOf("Sally")` returns `-1`.

(continued)

Some Methods in the Class `String` (Part 8 of 8)

Display 1.4 Some Methods in the Class `String`

```
int indexOf(A_String, Start)
```

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string that occurs at or after position *Start*. Positions are counted 0, 1, 2, etc. Returns −1 if *A_String* is not found.

EXAMPLE

After program executes `String name = "Mary, Mary quite contrary";`
`name.indexOf("Mary", 1)` returns 6.
The same value is returned if 1 is replaced by any number up to and including 6.
`name.indexOf("Mary", 0)` returns 0.
`name.indexOf("Mary", 8)` returns −1.

```
int lastIndexOf(A_String)
```

Returns the index (position) of the last occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns −1, if *A_String* is not found.

EXAMPLE

After program executes `String name = "Mary, Mary, Mary quite so";`
`greeting.indexOf("Mary")` returns 0, and
`name.lastIndexOf("Mary")` returns 12.

(continued)

Display 1.4 **Some Methods in the Class `String`**

`int compareTo(A_String)`

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering. Lexicographic order is the same as alphabetical order but with the characters ordered as in Appendix 3. Note that in Appendix 3 all the uppercase letters are in regular alphabetical order and all the lowercase letters are in alphabetical order, but all the uppercase letters precede all the lowercase letters. So, lexicographic ordering is the same as alphabetical ordering provided both strings are either all uppercase letters or both strings are all lowercase letters. If the calling string is first, it returns a negative value. If the two strings are equal, it returns zero. If the argument is first, it returns a positive number.

EXAMPLE

After program executes `String entry = "adventure";`
`entry.compareTo("zoo")` returns a negative number,
`entry.compareTo("adventure")` returns `0`, and
`entry.compareTo("above")` returns a positive number.

(continu

```
int compareToIgnoreCase(A_String)
```

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering, treating uppercase and lowercase letters as being the same. (To be precise, all uppercase letters are treated as if they were their lowercase versions in doing the comparison.) Thus, if both strings consist entirely of letters, the comparison is for ordinary alphabetical order. If the calling string is first, it returns a negative value. If the two strings are equal ignoring case, it returns zero. If the argument is first, it returns a positive number.

EXAMPLE

After program executes `String entry = "adventure";`
`entry.compareToIgnoreCase("Zoo")` returns a negative number,
`entry.compareToIgnoreCase("Adventure")` returns `0`, and
`"Zoo".compareToIgnoreCase(entry)` returns a positive number.

STRING BUFFER CLASS

- STRINGBUFFER class creates strings flexible length that can be modified in terms of both length and content.
- STRINGBUFFER may have characters and substrings inserted in the middle or appended to the end.
- STRINGBUFFER automatically grows to make room for such additions

Actually STRINGBUFFER has more characters pre allocated than are actually needed, to allow room for growth

STRING BUFFER CONSTRUCTORS

- **String Buffer():-** Reserves room fro 16 characters without reallocation

- **StringBuffer(int size):-** Accepts an integer argunent that explicilty sets the size of the buffer

- **StringBuffer(String str):-** Accepts STRING argument that sets the initial contents of the STRINGBUFFER and allocated room for 16 additional characters.

STRING BUFFER FUNCTIONS

- **length():-**Returns the current length of the string.
- **capacity():-** Returns the total allocated capacity.
- **void ensureCapacity():-** Preallocates room for a certain number of characters.
- **void setLength(int *len*):-** Sets the length of the string s1 to len.
  If len<s1.length(), s1 is truncated.
  If len>s1.length(), zeros are added to s1.
- **charAt(int *where*):-** Extracts value of a single character.
- **setCharAt(int *where*, char *ch*):-** Sets the value of character at specified position.

StringBuffer capacity() method

The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

StringBuffer ensureCapacity() method

The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

STRING BUFFER FUNCTIONS

- **append*(s2)*:-** Appends string s2 to s1 at the end.
- **insert*(n,s2)*:-** Inserts the string s2 at the position n of the string s1
- **reverse()**:- Returns the reversed object on when it is called.
- **delete*(int n1,int n2)*:-** Deletes a sequence of characters from the invoking object.

n1 → Specifies index of first character to remove

n2 → Specifies index one past the lastcharacter to remove

- **deleteCharAt*(int loc)*:-** Deletes the character at the index

STRING BUFFER FUNCTIONS

- *replace(int n1,int n2,String s1):-* Replaces one set of characters with another set.
- substring(*int startIndex*):- Returns the substring that starts at starts at *startIndex* and runs to the end.
- substring(*int startIndex, int endIndex*):- Returns the substring that starts at starts at *startIndex* and runs to the *endIndex-1*

Difference between String and StringBuffer :

No	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you concat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.


Thank you!
shutterstock.com · 567687052