Java Workshop

DDUC ACM STUDENT CHAPTER
DEEN DAYAL UPADHYAYAYA COLLEGE
UNIVERSITY OF DELHI

Exploring the String Class

THE MOST COMMONLY USED CLASS IN JAVA'S CLASS LIBRARY

Things you keep in mind about String class

- Strings are a very important part of programming.
- Like 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**.
- Every string you create is actually an object of type **String**.
- String constants are actually **String** objects.
  - For example, the string “I am a string” is a **String** object.

Things you keep in mind about String class

- Objects of type **String** are *immutable*; once a **String** object is created, its contents cannot be altered.
- While this may seem like a serious restriction, it is not, for two reasons:
  - If you need to change a string, you can always create a new one that contains the modifications.
  - Java defines two peer classes of **String**, called **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.
- This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones.

Moving one step further

- The **String**, **StringBuffer** and **StringBuilder** classes are defined in **java.lang**.
- All three implement the **CharSequence** interface.
- All are declared **final**, which means that none of these classes may be subclassed.
- When we say that **String** class is immutable, it means that contents of the **String** instance cannot be changed but it can be changed to point to some other **String** instance.

String Constructors

- The **String** class supports several constructors.
- To create an empty **String**, call the default constructor.
  - String s1 = new String();
  - It will create an instance of **String** with no characters in it.
- *We generally need strings with initial values. The **String** class provides a variety of constructors to handle this.*
- To create a **String** initialized by an array of characters.
- Constructor used is : String(char *chars[]*).
  - char chars[]={ ‘a’, ‘b’, ‘c’ };
  - String s2 = new String(chars);
    - This constructor initializes **s2** with the string “abc”.

String Constructors

- A **String** can also be constructed after specifying a subrange of a character array as initializer.
- Constructor used is:
  - 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.
  - char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
  - String s3 = new String(chars, 2, 3);
    - This constructor initializes **s3** with the string "cde".

String Constructors

- A **String** can also be constructed with the same character sequence as another **String** object.
- Constructor used is : String(String *strObj*).
- Here, *strObj* is a **String** object.
  - char chars[] = { ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’ };
  - String s3 = new String(chars, 2, 3);
    - This constructor initializes **s3** with the string “cde”.
  - String s4 = new String(s3);
    - This constructor will initializes **s4** with the string “cde”, same as **s3**.

String Constructors

- As Java's **char** type uses 16 bits, **String** can also be created using 8-bit ASCII characters.
- Constructor used are : String(byte *asciiChars[]*) and String(byte *asciiChars[]*, int *startIndex*, int *numChars*).
- *asciiChars* in both constructors specifies the array of bytes. Second constructor allows as to use subrange.
  - byte asciiChars[] = { 65, 66, 67, 68, 69, 70 };
  - String s5 = new String(asciiChars);
    - This constructor initializes **s5** with the string "ABCDEF".
  - String s6 = new String(asciiChars, 2, 3);
    - This constructor will initializes **s6** with the string "CDE".

Things to keep in mind

- The contents of array are copied whenever a **String** object is created from an array. If contents of array are modified after creating the string, the **String** will remain unchanged.

String Constructors

- A **String** can be created from a **StringBuffer**.
  - Constructor used is : String(StringBuffer *strBufferObj*).
  - Here, *strBufferObj* is a **StringBuffer** object.
- A **String** can be created from a **StringBuilder**.
  - Constructor used is : String(StringBuilder *strBuilderObj*).
  - Here, *strBuilderObj* is a **StringBuilder** object.

String Literals

- Till now, we created **String** objects explicitly from array of characters using **new** operator. However, we can create **String** objects using string literals.
- For each string literal, Java automatically constructs a **String** object. Thus we can use string literal to initialize a **String** object.
  - String s7 = “abc”;
- Because a **String** object is created for every string literal, a string literal can be used any place where a **String** object can be used.
- Methods can be directly called on a quoted string literal as if it were an object reference.
  - int len = “abc”.length();

Methods of String Class

- int length()
- char charAt(int *where*)
- void getChars(int *sourceStart*, int *sourceEnd*, char *target[]*, int *targetStart*)
- byte[] getBytes()
- char[] toCharArray()
- boolean equals(Object *str*)
- boolean equalsIgnoreCase(String *str*)
- boolean regionMatches(int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)
- boolean regionMatches(boolean *ignoreCase*, int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)
- boolean startsWith(String *str*)
- boolean endsWith(String *str*)

Methods of String Class

- static String valueOf(Object *object*)
- int compareTo(String *str*)
- int compareToIgnoreCase(String *str*)
- int indexOf(int *ch*)
- int indexOf(int *ch*, int *startIndex*)
- int lastIndexOf(int *ch*)
- int lastIndexOf(int *ch*, int *startIndex*)
- int indexOf(String *str*)
- int indexOf(String *str*, int *startIndex*)
- int lastIndexOf(String *str*)
- int lastIndexOf(String *str*, int *startIndex*)
- String substring(int *startIndex*)
- String substring(int *startIndex*, int *endIndex*)

Methods of String Class

- String concat(String *str*)
- String replace(char *original*, char *replacement*)
- String replace(CharSequence *original*, CharSequence *replacement*)
- String trim()
- String toLowerCase()
- String toUpperCase()
- boolean contains(CharSequence *str*)
- boolean contentEquals(CharSequence *str*)
- boolean contentEquals(StringBuffer *str*)
- boolean isEmpty()
- CharSequence subSequence(int *startIndex*, int *stopIndex*)

String Length

- The length of a string is the number of character it contains.
- To obtain the value, function used is : int *length()*.
  - char chars[] = { ‘a’, ‘b’, ‘c’ };
  - String s = new String(chars);
  - int len = s.length();
    - len now contains 3, as there are three characters in **s**.

String Concatenation

- In general, Java does not allow operators to be applied to **String** class.
- 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.”;
    - **s** now contains “He is 9 years old.”.

String Concatenation with Other Data Types

- Strings can also be concatenated with other data types.
- Values of other data types are automatically converted into its string representation within a **String** object and then concatenation takes place.
- Be careful while mixing other types of operations with string concatenation expressions. You might get surprising results.
  - String s = “four: ” + 2 + 2;
    - **s** now contains “four: 22” rather than “four: 4”.

String Conversion using valueOf() and toString()

- Java converts data into string representation during concatenation by calling one of the overloaded versions of the string conversion method **valueOf()**.
- **valueOf()** is overloaded for all the primitive types and for type **Object**.
- For primitive types, **valueOf()** returns a string that contains the human-readable equivalent of the value with which it is called.
- For objects, **valueOf()** calls **toString()** method on the object.
- **toString()** is the means by which you can determine the string representation for objects of classes that you will create.

valueof()

- 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.
- It is also overloaded for type **Object**. *(Keep in mind that **Object** is superclass for all classes.)*
- Few forms of valueOf() are :
  - static String valueOf(double num)
  - static String valueOf(long num)
  - static String valueOf(Object obj)
  - static String valueOf(char chars[])

valueOf()

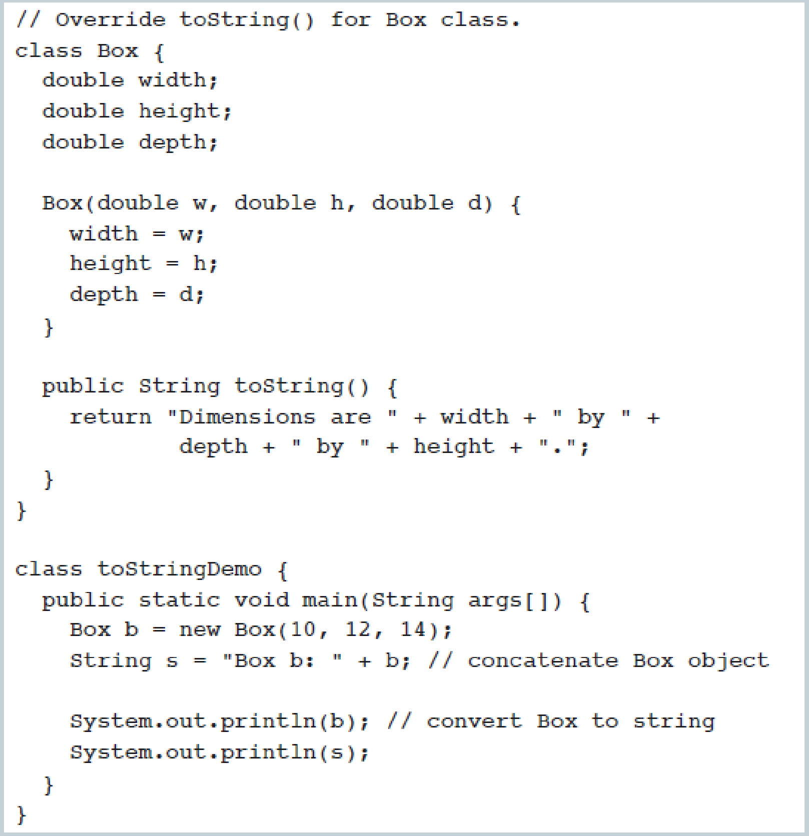
- All of the simple types are converted to their common **String** representation.
- Any object that is passed to **valueOf()** will return the result of a call to the object's **toString()** method.
- In fact, same result can be get by directly calling **toString()** of that object.
- For most arrays, **valueOf()** returns a rather cryptic string, which indicates that it is an array of some type, but for array of **char**, a **String** object is created that contains the characters in that array.
- A special version of **valueOf()** allows to specify a subset of **char** array.
  - static String valueOf(char chars[], int startindex, int numChars)

toString()

- Every class implements **toString()** because it is defined by **Object**.
- Default implementation of **toString()** may not be sufficient, so it is generally overridden to provided needed string representation.
- It is very easy to do because **toString()** has very simple form :
  - String toString()
- By implementing **toString()** for classes that you create, you allow them to be fully integrated into Java's environment.
- They can be then used in printing statements and concatenating expressions.

toString()

- Creating a class and overriding its **toString()** method :



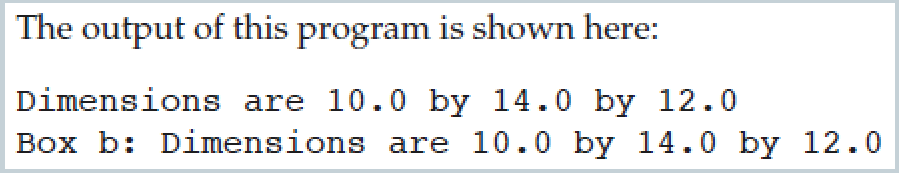
```
// Override toString() for Box class.
class Box {
  double width;
  double height;
  double depth;

  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
```

Character Extraction

- **String** class provides several ways in which characters can be extracted from a **String** object.
- Although characters that comprise a string within a **String** object can not 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 also begin at zero.

charAt()

- To extract a single character from a **String**, you can refer directly to an individual character via the **charAt()** method.
  - General form is char charAt(int *where*).
  - Here, *where* is the index of the character that you can want to obtain.
  - Value of *where* must be nonnegative and specify a location within the string. An **StringIndexOutOfBoundsException** is thrown otherwise.
    - char ch;
    - ch= “abc”.charAt(1);
      - Value of **ch** is now ‘b’.

getChars()

- If more than one characters are to be extracted at a time, **getChars()** method can be used.
  - General form is void getChars(int *sourceStart*, int *sourceEnd*, char *target[]*, int *targetStart*).
  - Here, *sourceStart* specifies the index of 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* and index within *target* at which the substring will be compied is specified by *targetStart*.
  - *target* array must be large enough to hold the number of characters in the specified substring.
    - char buf[] = new char[4];
    - "This is a demo of getChars".getChars (10, 14, buf, 0);
      - **buf** now contains "demo".

getChars()

- **IndexOutOfBoundsException** is thrown if any of following condition is true :
  - *sourceStart* is negative or/and
  - *sourceStart* is greater than *sourceEnd* or/and
  - *sourceEnd* is greater than the length of this string or/and
  - *targetStart* is negative or/and
  - *targetStart+(sourceEnd-sourceStart)* is larger than *target*.length.

getBytes()

- An alternative of **getChars()** is **getBytes()**, which stores characters in an array of bytes.
- It uses the default character-to-byte conversions provided by the platform.
  - Simplest form is byte[] getBytes().
- Other variants of **getBytes()** are also available.
- **getBytes()** is most useful while exporting a **String** value into an environment that does not support 16-bit Unicode characters.

toCharArray()

- If all the characters in a **String** object are to be converted into a character array, Java provides a method **toCharArray()** for this purpose also.
  - General form is char[] toCharArray().
  - This function is provided as a convenience, since **getChars()** can also be used to achieve the same result.

String Comparison

- **String** class includes a number of methods to :
  - compare strings keeping case in mind
  - compare strings ignoring case
  - compare substrings within strings
  - compare string with CharSequence
  - compare string with StringBuffer
  - check whether a string matches a regular expression

equals()

- To compare two strings for equality, **equals()** method can be used.
  - General form is boolean equals(String *str*).
  - Here, *str* is the **String** object to be compared with the string calling the method.
  - It returns **true** if the strings are same or say contains same characters in the same order, and **false** otherwise.
  - The comparison which takes place here is case sensitive.
    - "Hello".equals("Hell") gives **false**.
    - "Hello".equals("Hello") gives **true**.
    - "Hello".equals("hello") gives **false**.

equals() Versus ==

- As specified for all objects, equals() and == perform two different operations for **Strings** also.
- **equals()** method compares characters inside a **String** object but == operator compares two object references to see whether they refer to same instance or not.
  - String str1="hello"; String str2=str1;
    - str1.equals(str2) will give **true** and so will str1==str2.
  - String str1="hello"; String str2=new String(str1);
    - Str1.equals(str2) will give **true** but str1==str2 will return **false**.

equalsIgnoreCase()

- Another method to compare two strings for equality **equalsIgnoreCase()**.
  - General form is boolean equalsIgnoreCase(String *str*).
  - Here, *str* is the **String** object to be compared with the string calling the method.
  - It returns **true** if the strings are same or say contains same characters in the same order, and **false** otherwise.
  - The comparison which takes place here is case insensitive.
    - "Hello".equalsIgnoreCase("Hell") gives **false**.
    - "Hello".equalsIgnoreCase("Hello") gives **true**.
    - "Hello".equalsIgnoreCase("hello") gives **true**.

regionMatches()

- To compare a specific region inside a string with another specific region in another string, we can use the **regionMathches()** method.
  - General form is boolean regionMatches(int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*).
    - Here, *strartIndex* is the index at which the region begins within the invoking **String** object.
    - *str2* is the **String** object to be compared.
    - The index at which the region begins in *str2* is specified by *str2StartIndex*.
    - *numChars* specifies the length of the region.
  - It returns **true** if these regions represent identical character sequences, **false** otherwise.
  - The comparison which takes place here is case insensitive.

regionMatches()

- An overloaded version of **regionMatches()** also exists which allows us compare a specific region inside a string with another specific region in another string with chhosing whether to ignore case of characters or not.
  - General form is boolean regionMatches(boolean ignoreCase, int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*).
    - Here, everything is same as previous version but if *ignoreCase* is **true** then case of characters is ignored, otherwise its significant.
  - It returns **true** if these regions represent identical character sequences, **false** otherwise. (Keeping value of *ignoreCase* in mind.)

regionMatches()

- The result of **regionMatches()** is false if and only if at least one of the following is true:
  - *startIndex* is negative
  - *str2StartIndex* is negative
  - *startIndex+numChars* is greater than the length of the invoking **String** object
  - *str2StartIndex+numChars* is greater than the length of *str2*
  - There is some nonnegative integer *k* less than *numChars* such that:
    this.charAt(*startIndex+k*) != *str2*.charAt(*str2StartIndex+k*)
    (Keep overloaded version in mind.)

compareTo()

- Sometimes its not enough to check whether two strings are same or not. In those cases we might need to know that which string is lesser and which one is greater or may be equal.
- A string is less than another if it comes before the other in dictionary order.
- For this purpose **compareTo()** method can be used. This method is specified by the **Comparable<T>** interface which is implemented by **String** class.
  - General form is int compareTo(String *str)*.
  - Here *str* is the string to be compared by the invoking **String** object.
  - This function returns:
    - Zero if both strings are equal.
    - Less than zero if invoking string is less then *str*.
    - Greater than zero if invoking string is greater than *str*.

contains()

- To check whether a **String** object contains an instance of **CharSequence** or not, Java provides a method.
- **contains()** is the method used for this purpose.
  - General form is boolean contains(CharSequence *str*).
    - Here, *str* is the **CharSequence** to be checked, whether contained in invoking **String** object.
    - If *str* is contained in the invoking **String**, method returns **true**, otheerwise **false**.
    - If *str* or invoking string is **null** then **NullPointerException** is thrown.
      - “Hello”.contains(“He”) returns **true**.
      - “Hello”.contains(“he”) returns **false**.
      - “Hello”.contains(“”) returns **true**.
      - “Hello”.contains(null) throws **NullPointerException**.

isEmpty()

- A method, which at times becomes very useful to check whether a **String** object contains any character or not is **isEmpty()**.
  - General form is boolean isEmpty().
  - It return **true** if the invoking **String** object contains no character and its length is zero, otherwise returns **false**.
    - String str1=“”;
      - str1.isEmpty() will return **true**.
    - String str2=“Hello”;
      - str2.isEmpty() will return **false**.

Searching Strings

- **String** class provides two methods **indexOf()** and **lastIndexOf()**, that allow you to search a string for a specified character or a substring.
- These two methods are overloaded in several different ways.
- In all cases methods return -1 on failure, i.e. when the character or substring is not found in the invoking **String**.

indexOf()

- **indexOf()** method is overloaded in four ways.
  - int indexOf(int *ch*) returns first occurrence of *ch* in invoking **String** object if found, else returns -1.
  - int indexOf(String *str*) returns first occurrence of *str* in invoking **String** object if found, else returns -1.
  - int indexOf(int *ch*, int *startIndex*) returns first occurrence of *ch* in invoking **String** object after starting search from *startIndex* if found, else returns -1.
    - Search runs from *startIndex* to the end of string.
  - int indexOf(int *str*, int *startIndex*) returns first occurrence of *str* in invoking **String** object after starting search from *startIndex* if found, else returns -1.
    - Search runs from *startIndex* to the end of string.

indexOf()

- For code fragement:
  - String str="runner runs for victory.";
  - System.out.println("indexOf(run) : "+str.indexOf("run"));
  - System.out.println("indexOf(r) : "+str.indexOf('r'));
  - System.out.println("indexOf(run,3) : "+str.indexOf("run",3));
  - System.out.println("indexOf(r,8) : "+str.indexOf('r',8));
- Output is:
  - indexOf(run) : 0
  - indexOf(r) : 0
  - indexOf(run,3) : 7
  - indexOf(r,8) : 14

lastIndexOf()

- **lastIndexOf()** method is overloaded in four ways.
  - int lastIndexOf(int *ch*) return last occurrence of *ch* in invoking **String** object if found, else returns -1.
  - int lastIndexOf(String *str*) return last occurrence of *str* in invoking **String** object if found, else returns -1.
  - int lastIndexOf(int *ch*, int *startIndex*) return last occurrence of *ch* in invoking **String** object after starting search from *startIndex* if found, else returns -1.
    - Search runs from *startIndex* to zero.
  - int lastIndexOf(int *str*, int *startIndex*) return last occurrence of *str* in invoking **String** object after starting search from *startIndex* if found, else returns -1.
    - Search runs from *startIndex* to zero.

lastIndexOf()

- For code fragement:
  - String str="runner runs for victory.";
  - System.out.println(“lastIndexOf(run): "+str. lastIndexOf("run"));
  - System.out.println("lastIndexOf(r): "+str. lastIndexOf('r'));
  - System.out.println("lastIndexOf(run,9): "+str. lastIndexOf("run",9));
  - System.out.println("lastIndexOf(r,8): "+str. lastIndexOf('r',8));
- Output is:
  - lastIndexOf(run): 7
  - lastIndexOf(r): 21
  - lastIndexOf(run,9): 7
  - lastIndexOf(r,8): 7

Modifying a String

- As we know that **String** objects are *immutable*, whenever a modification is needed in a string:
  - it must be either copied into a **StringBuffer** or **StringBuilder**
  - or we should use a **String** method that constructs a new copy of the string with modification done into it.
- Many methods are there which might seem to be changing a string but in real are creating new copies of modified strings and then returning them.

substring()

- To extract a substring from a string, **substring()** method is used.
  - This is overloaded in two ways.
  - String substring(int *startIndex*) returns a substring consisting of characters of invoking string from *startIndex*.
    - "Hello".substring(3) returns "lo".
    - It throws **StringIndexOutOfBoundsException** if *startIndex* is negative or greater then length of invoking string.
  - String substring(int *startIndex*, int *endIndex*) returns a substring consisting of charaters of invoking string from *startIndex* upto, but not including the *endIndex*.
    - "Hello".substring(2,5) returns "llo".
    - It throws **StringIndexOutOfBoundsException** if the *startIndex* is negative, or *endIndex* is larger than the length of the invoking string, or *startIndex* is larger than *endIndex*.

concat()

- Two strings can be concatenated using **concat()** method.
  - General form is 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 +.
    - String s1="One";
    - String s2="Two";
    - String s3=s1.concat(s2);
      - **s3** now contains "OneTwo". Same result can be achieved by s3=s1+s2.

replace()

- The **replace()** method is overloaded in two forms.
- String replace(char *original*, char *replacement*) replaces all occurrences of *original* from invoking string by *replacement*. The modified string is returned.
  - String s="Hello".replace('l', 's');
    - **s** now contains "Hesso".
- String replace(CharSequence *original*, CharSequence *replacement*) replaces one character sequence with another.
  - String s="Hello".replace("el","as");
    - **s** now contains "Haslo".
  - It throws **NullPointerException** if any of *original* or *replacement* is null.

trim()

- The **trim()** method return a copy of the invoking string after removing leading and trailing whitespaces.
  - General form is String trim().
    - String s=“ Hello world ”.trim();
      - **s** now contains “Hello world”.

Changing the Case of Characters Within a String

- The method **toLowerCase()** converts all characters in a string from uppercase to lowercase and returns the modified string.
  - General form is String toLowerCase().
  - String s="Hello".toLowerCase();
    - **s** now contains "hello".
- The method **toUpperCase()** converts all characters in a string from lowercase to uppercase and returns the modified string.
  - General form is String toUpperCase().
  - String s="Hello".toUpperCase();
    - **s** now contains "HELLO".

StringBuffer

A PEER CLASS OF STRING

StringBuffer

- **StringBuffer** is a peer class of **String** that provides much of the functionality of strings.
- Where **String** represents fixed-length, immutable character sequences, and in contrast, **StringBuffer** represents growable and writable character sequences.
- It may have characters and substrings inserted in middle or appended to the end.
- It automatically grows to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth.

StringBuffer Constructors

- **StringBuffer** class has four constructors.
- StringBuffer() is the default constructor with no arguments. It reserves room for **16** characters without reallocation.
- StringBuffer(int *size*) is the constructor where *size* is now set as the size of the buffer.
  - It throws **NegativeArraySizeException** if size is less than zero.
- StringBuffer(String *str*) is the constructor accepting the **String** argument and sets initial contents of **StringBuffer** object and reserves room for **16** more characters without reallocation.
  - It throws **NullPointerException** if str is null.

StringBuffer Constructors

- StringBuffer(CharSequence *chars*) creates an object that contains the character sequence *chars* and reserves room for **16** more characters.
  - It throws **NullPointerException** if chars is null.
- **StringBuffer** allocates room for **16** additional characters when no specific buffer length is requested as reallocation is a costly process in terms of time. Also frequent allocation can fragment memory.

length() and capacity()

- The current length of **StringBuffer** can be found with help of **length()** method.
  - General form is int length().
- Total allocated capacity of **StringBuffer** can be found using **capacity()** method.
  - General form is int capacity().
- Here is an example:
  - StringBuffer strBuf=new StringBuffer("Hello");
  - int len=strBuf.length();
  - int cap=strBuf.capacity();
    - len now contains 5 and cap contains 21 i.e. (5+16).

ensureCapacity()

- If preallocation of room for a certain number of characters is needed after creating a **StringBuffer** object, **ensureCapacity()** can help in doing so.
- This is useful when we know in advance that we will be adding a specific number of characters in **StringBuffer** very soon.
    - General form is void ensureCapacity(int *minCapacity*).
    - *minCapacity* is the minimum size of buffer. A buffer larger than *minCapacity* may be allocated for reasons of efficiency.
    - It ensures that the capacity is at least equal to the specified minimum. If the current capacity is less than the argument, then a new internal array is allocated with greater capacity. The new capacity is the larger of:
        - The *minCapacity* argument.
        - Twice the old capacity, plus 2.
    - If the minCapacity argument is nonpositive, this method takes no action and simply returns.

setLength()

- To set the length of a string within a **StringBuffer** object, **setLength()** method is used.
  - General form is void setLength(int *len*).
  - Here, *len* specifies the length of the string and must be nonnegative.
  - It throws **IndexOutOfBoundsException** if len is negative.
  - If *len* is less than the current length, the length is changed to the specified length.
  - If *len* is greater than or equal to the current length, sufficient null characters are appended so that length becomes the *len*.

charAt() and setCharAt()

- The value of a single character can be obtained from a **StringBuffer** by using **charAt()** method.
  - General form is char charAt(int *where*).
  - Here, *where* specifies the index of the character being obtained.
  - It throws **IndexOutOfBoundsException** if *where* is negative or not less than length() of invoking **StringBuffer** object.
- The value of a single character of **StringBuffer** can be set by using **setCharAt()** method.
  - General form is void setCharAt(int *where*, char *ch*).
  - Here, *where* specifies the index of the character to be set and *ch* is new value of the character.
  - It throws **IndexOutOfBoundsException** if *where* is negative or not less than length() of invoking **StringBuffer** object.

getChars()

- To copy a substring of a **StringBuffer** into an array, **getChars()** method can be used.
  - General form is 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, one past the end of desired substring.
  - *target* is the array which will receive characters and the index within *target* at which the substring will be copied is specified by *targetStart*.
  - *target* array must be large enough to hold the number of characters in the specified substring.

getChars()

- Things to keep in mind for **getChars()** method are:
  - It throws **NullPointerException** if *target* is null.
  - It throws **IndexOutOfBoundsException** if any of the following is true:
    - *sourceStart* is negative
    - *targetStart* is negative
    - *sourceStart* is greater than *sourceEnd*
    - *sourceEnd* is greater than length() of invoking **StringBuffer** object
    - *targetStart+sourceEnd-sourceBegin* is greater than *target*.length

append()

- The **append()** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object.
- After modification, buffer itself is returned to allow subsequent calls to be chained together.
- It has several overloaded versions. Here are a few of its forms:
  - StringBuffer append(String *str*)
  - StringBuffer append(int *num*)
  - StringBuffer append(Object *obj*)
- When null is passed as argument to append then four character string “null” is appended to the end of buffer.

insert()

- **insert()** method inserts one string into buffer.
- It is overloaded to accept values of all the primitive data types, plus **String**s, **Object**s, and **CharSequence**s.
- Like **append()**, it obtains the string representation of the value it is called with. The string is then inserted into the invoking **StringBuffer** object at the index specified.
- Here are a few of its forms:
  - StringBuffer insert(int *index*, String *str*)
  - StringBuffer insert(int *index*, int *num*)
  - StringBuffer insert(int *index*, Object *obj*)
    - If index is less than zero or greater than length() of invoking **StringBuffer** object then **IndexOutOfBoundsException** is thrown.
- If null is passed as second argument to **insert()** then four character string "null" is inserted at that *index*.

reverse()

- Characters within a **StringBuffer** object can be reversed using **reverse()** method.
  - General form is StringBuffer reverse().
  - It returns the reverse of object on which it was called.
    - StringBuffer strBuf=new StringBuffer(“Hello”);
    - strBuf.reverse();
      - Now strBuf contains “olleH”.

delete()

- Characters within a **StringBuffer** can be deleted by using the methods **delete()** and **deleteCharAt()**.
- General form of delete() is:
  - StringBuffer delete(int *startIndex*, int *endIndex*)
  - Thus substring deleted runs from *startIndex* to *endIndex-1*.
  - The resulting **StringBuffer** is returned.
  - It throws **StringIndexOutOfBoundsException** if
    - *startIndex* is negative
    - *startIndex is* greater than length() of invoking **StringBuffer** object
    - *startIndex* is greater than *endIndex*.

deleteCharAt()

- General form of deleteCharAt() is:
  - StringBuffer deleteCharAt(int *location*)
  - It deletes a character from the invoking object from the index specified by *location*.
  - The resulting **StringBuffer** is returned.
  - It throws **StringIndexOutOfBoundsException** if
    - *location* is negative
    - *location* is greater than or equal to length() of invoking **StringBuffer** object.

replace()

- One set of characters of a **StringBuffer** object can be replaced by another set using **replace()** method.
- General form is StringBuffer replace(int *startIndex*, int *endIndex*, String *str*)
  - *startIndex* specifies the index of first character of substring and *endIndex* specifies an index one past the last character of substring to be replaced. *str* is the replacement string.
  - Thus substring replaced runs from *startIndex* to *endIndex-1*.
  - The resulting **StringBuffer** is returned.
  - It throws **StringIndexOutOfBoundsException** if
    - *startIndex* is negative
    - *startIndex is* greater than length() of invoking **StringBuffer** object
    - *startIndex* is greater than *endIndex*.

substring()

- A portion of a **StringBuffer** can be obtained using **substring()** method.
- It has two forms:
  - String substring(int *startIndex*)
  - String substring(int *startIndex*, int *endIndex*)
- First form returns the substring that starts at *startIndex* and runs to the end of the invoking **StringBuffer** object.
  - Throws **StringIndexOutOfBoundsException** if *startIndex* is less than zero, or greater than the length of the invoking object.
- Second form returns the substring that starts at *startIndex* and runs through *endIndex-1*.
  - Throws **StringIndexOutOfBoundsException** if *startIndex* or *endIndex* are negative or greater than length(), or *startIndex* is greater than *endIndex*.

Additional StringBuffer Methods

- int indexOf(String *str*)
- int indexOf(String *str*, int *startIndex*)
- int lastIndexOf(String *str*)
- int lastIndexOf(String *str*, int *startIndex*)
- void trimToSize()

StringBuilder

- Introduced by JDK 5
- Identical to **StringBuffer** but not synchronized like **StrigBuffer**
- This limitation makes **StringBuilder** thread unsafe but its faster than **StringBuffer**.