Programming in Java

String and StringBuilder

string

- Unlike many other languages that implement strings as character arrays, Java implements strings as objects of type String.
- Advantage of creating strings as objects is that String objects can be constructed a number of ways, making it easy to obtain a string when needed.
- Once a **String** object has been created, you cannot change the characters that comprise that string.
- You can still perform all types of string operations. The difference is that each time we 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.

string

- 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.

- The String, StringBuffer, and StringBuilder classes are defined in java.lang. Thus, they are available to all programs automatically.

- The String, StringBuffer, and StringBuilder classes are declared final, which means that none of these classes may be subclassed.

The String Constructors

- The String class supports several constructors. To create an empty String, you call the default constructor. For example,

  String s = new String();

  will create an instance of String with no characters in it.

- Frequently, we 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”.

The String Constructors

- We 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

- We 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);
}
}
```

The String Constructors

- Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array. Their forms are shown here:

String(byte asciiChars[ ])

String(byte asciiChars[ ], int startIndex, int numChars)

Here, asciiChars specifies the array of bytes. The second form allows us 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);
```

The String Constructors

- The contents of the array are copied whenever we create a String object from an array. If we modify the contents of the array after we have created the string, the String will be unchanged.

String Length

- The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method, shown here:

```
int length( )
```

- The following fragment prints “3”, since there are three characters in the string **s**:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

String Literals

- 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
```

It calls the **length( )** method on the string “abc”. As expected, it prints “3”.

```
System.out.println("abc".length());
```

String Concatenation with Other Data Types

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

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

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)
```

Here, where is the index of the character that you want to obtain. The value of here 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.

Character Extraction

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)

```
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);
                                }
                }
```

Character Extraction

getBytes( )

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

```
byte[ ] getBytes( )
```

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.

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. Its general form is:

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

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

Byte Extraction

```
class TestingByteArrayCharArray
   {
            public static void main(String[] args)
              {
                   String a=new String();
                a="testing get bytes";
                byte charArray[]=new byte[a.length()];
                charArray=a.getBytes();
                 for(byte i: charArray)
                     {
                     System.out.println(i);
                     }
                  }
   }
```

```
class TestingByteArrayCharArray
    {
            public static void main(String[] args)
              {
                  String a=new String();
                 a="testing get bytes";
                 byte byteArray[]=new byte[a.length()];
                 byteArray=a.getBytes();
                  for(byte i: byteArray)
                       {
                        System.out.println(i);
                       }
                //System.out.println(a[5]);
                 char charArray[]=new char[a.length()];
                 charArray=a.toCharArray();
                  for(int i=0; i<a.length(); i++)
                       {
                        System.out.println(charArray[i]);
                       }
                  }
   }
```

equals( ) and equalsIgnoreCase( )

- boolean equals(Object *str*)
- boolean equalsIgnoreCase(String *str*)

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

startsWith( ) and endsWith( )

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,

“student".endsWith(“dent")

and

“smith”.startsWith(“smi")

boolean startsWith(String str, int startIndex)

Here, startIndex specifies the index into the invoking string at which point the search will begin. For example, “Student".startsWith(“dent”, 3) returns true.

equals( ) Versus ==

```
// equals() vs ==
class EqualsNotEqualTo {
        public static void main(String args[]) {
                String s1 = "Hello";
                String s2 = new String(s1);
                System.out.println(s1 + " equals " + s2 + " -> " +s1.equals(s2));
                System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
                        }
                }
```

- The **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( )

- For sorting applications, we 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 String method compareTo( ) serves this purpose. 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.

compareTo( )

```
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 i = 0; i < arr.length; i++)
                        {
                        for(int j = i+ 1; j < arr.length; j++)
                                {
                                if(arr[i].compareTo(arr[j]) > 0)
                                        {
                                        String t = arr[j];
                                        arr[j] = arr[i];
                                        arr[i] = t;
                                }
                                }
                                System.out.println(arr[i]);
                        }
                }
        }
```

compareTo( )

- **compareTo( )** takes into account uppercase and lowercase letters. The word “Now” came out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.

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

  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.

Searching Strings

- To search for the first occurrence of a character, use

  int indexOf(char ch)

- To search for the last occurrence of a character, use

  int lastIndexOf(char 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(char ch, int startIndex)

  int lastIndexOf(char ch, int startIndex)

  int indexOf(String str, int startIndex)

  int lastIndexOf(String str, int startIndex)

Searching Strings

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

Searching Strings

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

- Because String objects are immutable, whenever we want to modify a String, you must either copy it into a StringBuffer or StringBuilder, or use one of the following String methods, which will construct a new copy of the string with your modifications complete.
- **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 us 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.

Modifying a String

```
// Substring replacement.
class StringReplace {
        public static void main(String args[]) {
                String org = "This is a test. This is, too.";
                String search = "is";
                String sub = "was";
                String result = "";
                int i;
                do {        // replace all matching substrings
                        System.out.println(org);
                        i = org.indexOf(search);
                        if(i != -1) {
                                result = org.substring(0, i);
                                result = result + sub;
                                result = result + org.substring(i + search.length());
                                org = result;
                        }
                } while(i != -1);
                }
        }
```

concat

We can concatenate two strings using **concat( )**, shown here:

String concat(String *str*)

This method creates a new object that contains the invoking string with the contents of *str* appended to the end. **concat( )** performs the same function as +. For example,

```
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string “onetwo” into **s2**. It generates the same result as the following sequence:

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

replace( )

The replace( ) method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

```
String replace(char original, char replacement)
```

Here, original specifies the character to be replaced by the character specified by replacement. The resulting string is returned. For example,

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

puts the string “Hewwo” into **s**. The second form of **replace( )** replaces one character sequence with another. It has this general form:

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

trim( )

The trim( ) method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

```
String trim( )
```

Here is an example:

```
String s = " Hello World ".trim();
```

This puts the string “Hello World” into s.

The trim( ) method is quite useful when you process user commands.

Changing the Case of Characters Within a String

String toLowerCase( )

String toUpperCase( )

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

Java String join

- The java string join() method returns a string joined with given delimiter. In string join method, delimiter is copied for each elements.

- In case of null element, "null" is added. The join() method is included in java string since JDK 1.8.

- public static String join(CharSequence delimiter, CharSequence... elements)

- Parameters

  delimiter : char value to be added with each element

  elements : char value to be attached with delimiter
- Returns

  joined string with delimiter
- Throws

  NullPointerException if element or delimiter is null.

Java String join

```
public class StringJoinExample
{
    public static void main(String args[])
        {
            String joinString1=String.join("-","welcome","to","javatpoint");
            System.out.println(joinString1);
        }
}
```

Output:

welcome-to-javatpoint

Java String format

- The java string format() method returns the formatted string by given locale, format and arguments.
- If you don't specify the locale in String.format() method, it uses default locale by calling Locale.getDefault() method.
- The format() method of java language is like sprintf() function in c language and printf() method of java language.

- Signature
- There are two type of string format() method:
- public static String format(String format, Object... args)
- public static String format(Locale locale, String format, Object... args)

- Parameters
- locale : specifies the locale to be applied on the format() method.
- format : format of the string.
- args : arguments for the format string. It may be zero or more.
- Returns
- formatted string
- Throws

  NullPointerException : if format is null.

  IllegalFormatException : if format is illegal or incompatible.

Java String format

```
public class FormatExample
 {
  public static void main(String args[])
    {
            String name=“Jasbeer";
            String sf1=String.format("name is %s",name);
            String sf2=String.format("value is %f",32.33434);
        String sf3=String.format("value is %32.12f",32.33434);//returns 12 char fractional part filling with 0
            System.out.println(sf1);
            System.out.println(sf2);
             System.out.println(sf3);
   }
}
```

- Output:
- name is Jasbeer
- value is 32.334340
- value is 32.334340000000

Java String intern

- The java string intern() method returns the interned string. It returns the canonical representation of string.
- It can be used to return string from pool memory, if it is created by new keyword.
- Signature
- The signature of intern method is given below:
- public String intern()
- Returns
- interned string

Java String intern

```
public class InternExample
 {
  public static void main(String args[])
     {
          String s1=new String("hello");
          String s2="hello";
  String s3=s1.intern();//returns string from pool, now it will be same as s2
          System.out.println(s1==s2);//false because reference is different
          System.out.println(s2==s3);//true because reference is same
  }}
```

- Output:

  false
  true

StringBuilder

- public final class StringBuilder extends Object implements Serializable, CharSequence

- A mutable sequence of characters. This class provides an API compatible with StringBuffer, but with no guarantee of synchronization. This class is designed for use as a drop-in replacement for StringBuffer in places where the string buffer was being used by a single thread (as is generally the case).
- Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations.

- The principal operations on a StringBuilder are the append and insert methods, which are overloaded so as to accept data of any type. Each effectively converts a given datum to a string and then appends or inserts the characters of that string to the string builder.

- The append method always adds these characters at the end of the builder; the insert method adds the characters at a specified point.

Constructors

- StringBuilder()

Constructs a string builder with no characters in it and an initial capacity of 16 characters.

- StringBuilder(CharSequence seq)

Constructs a string builder that contains the same characters as the specified CharSequence.

- StringBuilder(int capacity)

Constructs a string builder with no characters in it and an initial capacity specified by the capacity argument.

- StringBuilder(String str)

Constructs a string builder initialized to the contents of the specified string.

Methods

- public StringBuilder append(String s)

is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.

- public StringBuilder insert(int offset, String s)

is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.

- public StringBuilder replace(int startIndex, int endIndex, String str)

is used to replace the string from specified startIndex and endIndex.

- public StringBuilder delete(int startIndex, int endIndex)

is used to delete the string from specified startIndex and endIndex.

- public StringBuilder reverse()

is used to reverse the string.

- public int capacity()

is used to return the current capacity.

Methods

- public void ensureCapacity(int minimumCapacity)
  is used to ensure the capacity at least equal to the given minimum.
- public char charAt(int index)
  is used to return the character at the specified position.
- public int length()
  is used to return the length of the string i.e. total number of characters.
- public String substring(int beginIndex)
  is used to return the substring from the specified beginIndex.
- public String substring(int beginIndex, int endIndex)
  is used to return the substring from the specified beginIndex and endIndex.