String Handling

String Constructors

- **String() -** default constructor creates empty string.
- **String(char chars[ ])** - String can be initialized with array of characters.
  e.g:
  — char chars[ ] = { 'a', 'b', 'c' };
  — String s = new String(chars);
- **String(char chars[ ], int startIndex, int numChars)** - specifying a subrange of a character array as an initializer.
  e.g
  — char chars[ ] = { 'a', 'b', 'c', 'd', 'e', 'f' };
  — String s = new String(chars, 2, 3);
- **String(String strObj)** - We can construct a String object that contains the same character sequence as another String object.
  e.g:
  — char c[ ] = {'J', 'a', 'v', 'a'};
  — String s1 = new String(c);
  — String s2 = new String(s1);
- String(byte asciiChars[ ])
- String(byte asciiChars[ ], int startIndex, int numChars)
  - Even though Java's char type uses 16 bits to represent the Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from theASCII character set.
  - Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array
  Example:

```
class SubStringCons {
public static void main(String[ ] args) {
  byte ascii[ ] = {65,66,67,68,69,70};
  String s1=new String(ascii);
  System.out.println(s1);
  String s2=new String(ascii, 2,3);
  System.out.println(s2);
  }
}
```

String Length

The length of a string is the number of characters that it contains. To obtain this value, call the length() method of String class, whose signature is:
- int length()
e.g:
char chars[ ] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
o/p:3

Special String Operations

String Literals
For each string literal in our program, Java automatically constructs a String object. Thus, we can use a string literal to initialize a String object.
Example:
String s2 = "abc" ;

String Concatenation
In general, Java does not allow operators to be applied to String objects. The one exception to this rule is the + operator, which concatenates two strings, producing a String object as the result. This allows us to chain together a series of + operations.
Example:
String age = "9" ;
String s = "He is" + age + " years old." ;
System.out.println(s);

String Concatenation with other data types
Example:
String s = "Four:" +2+2;
System.out.println(s);
Output: Four:22
String s = "Four:" +(2+2);
Output: Four:4

String Conversion and toString() method

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method valueOf() defined by String class. Every class implements toString() because it is defined by Object. However, the default implementation of toString() is seldom sufficient. For most important classes that we create, we will want to override toString() and provide our own string representations. The toString() method has this general form: String toString()
Example of valueOf() method:

```
class Stringtest {
public static void main(String[ ] args) {
   int i = 1;
  Double d = 2.5;
  String str_i = String.valueOf(i); //converting primitive int to String
  String str_d = String.valueOf(d); //converting double to String
  System.out.println(str_i);
  System.out.println(str_d);
  }
}
```

```
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

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

CharAt()

— To extract a single character from a String, we can refer directly to an individual character via the charAt( ) methods as shown.
char charAt(int where)

e.g:
char ch;
ch = "abc".charAt(1); **//assigns the value "b" to ch.**

getChars()

If we need to extract more than one character at a time, we can use the getChars() method.
General Form:
- void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart).

sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from sourceStart through sourceEnd – 1. The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart.

```
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);
}
}
```

Output of this program:
demo

getBytes()

- byte[] getBytes()-
- it uses the default character-to-byte conversions

toCharArray()

If we 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 Characterextract {
  public static void main(String[] args) {
    String s4 = "I am a disco dancer.";
    byte arr1[] = new byte[4];
    s4.getBytes(2, 6, arr1, 0);
    for (byte c : arr1) {
      System.out.println(c);
     }
   }
}
```

Output:
97
109
32
97

Example: toCharArray()

```
class Characterextract {
  public static void main(String[] args) {
    String s4 = "I am a disco daner.";
    char[] mychararr = s4.toCharArray();
    for(char c : mychararr) {
      System.out.print(c);
      }
  }
}
```

Output:

```
I am a disco dancer.
```

regionMatches()

The regionMatches() method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows to ignore case in such comparisons. The general forms for these two methods:

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

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

Example:

```
class Stringcompare {
   public static void main(String[] args) {
       String s5 = "he likes to go disco";
       String s6 = "I am a disco dancer";
       String s7 = "I am a Disco dancer";
       System.out.println(s5.regionMatches(15,s6,7,5));
       System.out.println(s5.regionMatches(15,s7,7,5));
       System.out.println(s5.regionMatches(true,15,s7,7,5)); // ignoring the case
}
```

Output:

```
true
false
true
```

startsWith() and endsWith()

String defines two routines 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 ends with a specified string. They have the following general forms:

boolean startsWith(String str)

- boolean endsWith(String str)

```
System.out.println("Foobar".startsWith("Foo"));
System.out.println("Foobar".startsWith("bar",3));
// The output will be true for both
```

equals() and equalsIgnoreCase()

To compare two strings for equality, use equals(). It has this general form: boolean equals(Object str)

- str is the String object being compared with the invoking String object. It returns true if the strings contain the same characters in the same order, and false otherwise. The comparison is case-sensitive.

To perform a comparison that ignores case differences, call equalsIgnoreCase(). When it compares two strings, it considers A-Z to be the same as a-z. It has this general form:
e.g:

- “a”.equals(“a”) //**true**

- "a".equals("A") //**false**

- "a".equalsIgnoreCase(“A”)

equals() vs = =

It is important to understand that the equals() method and the == operator perform two different operations. 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.

```
class Strcomp {
public static void main(String[] args) {
   String s1 = “hello”;
   String s2 = “hello”;
   String s3 = new String(“hello”);
   String s4 = “hElLo”;
   System.out.println(s1 = = s2); //true
   System.out.println(s1 = = s3); //false
   System.out.println(s1 = = s4); //false
   System.out.println(s1.equals(s2)); //true
   System.out.println(s1.equals(s3)); //true
   System.out.println(s1.equals(s4)); //false
   System.out.println(s1.equalsIgnoreCase(s4)); //true
  }
}
```

compareTo()

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is less than, equal to, or greater than the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order.
The String method compareTo() serves this purpose. It has this general form:

- int compareTo(String str)

Sorting string array using compareTo()

```
class StringSort {
  public static void main(String[] args) {
    String s[] = {"Ashesh", "Ram", "Hari", "Shyam", "Praveen"};
    for(int j=0; j<s.length; j++) {
      for(int i=0; i<s.length-j-1; i++) {
        int comparedvalue = s[i].toLowerCase().compareTo(s[i+1].toLowerCase());
        if(comparedvalue>0) {
          String temp = s[i];
          s[i] = s[i+1];
          s[i+1] = temp;
        }
      }
  }
  for(String string : s) {
    System.out.println(string);
    }
   }
}
```

Modifying a string

substring()

We can extract a substring using substring() method. It has two forms.

- The first is String substring(int startIndex)
  where, 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)

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 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('I','w');

puts the string "Hewwo" into s.

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.

Example modification of string

```
class StringModification {
  public static void main(String[] args) {
  String test = “hello java hello”;
  String trimmedtest = test.trim();
  System.out.println(trimmedtest);
  String replacedstring = test.replace(“ell”, “wow”);
  System.out.println(replacedstring);
  String testsubstr = test.substring(4);
  System.out.println(testsubstr);
  String testsubstr1 = test.substring(4,8);
  System.out.println(testsubstr1);
  }
}
```

Output of the above program

```
hello java hello
hwowo java hwowo
lo java hello
lo j
```