CS 1301 – Ch 8, Part A

Sections	Pages	Review Questions	Programming Exercises
8.1-8.8	264-291	1-30	2,4,6,8,10,12,14,16,18,24,28

This section of notes discusses the String class.

The String Class

1. A *String* is an object from the *String* class.

2. http://java.sun.com/javase/6/docs/api/java/lang/String.html

3. We have already seen that we can create a String with this:

```
String s1 = "Hello";
```

Which is equivalent to:

```
String s2 = new String( "Hello" );
```

You can also create a string from an array of char's:

```
char[] msg = { 'H', 'i', ' ', 't', 'h', 'e', 'r', 'e' };
String s3 = new String( msg );
```

4. A String object is immutable.

```
String s = "Hello";
s = "Goodbye";
```

The second statement creates a new string and changes the variable s to point to it. The old string, "Hello" still exists, but has no references to it so it will eventually be picked up by the garbage collector.

5. String literals are interned in Java. This means that only one copy is stored for string literals with the same value. For example, s4 and s5, below both refer to the same string in memory. Remember, strings are immutable, so if we change one, a new copy will be created. The disadvantage is that it takes more time to create and intern a string than to just create it. The advantage is that, in general, string processing (comparisons, etc.) is more efficient in time and/or space.

```
String s4 = "Hello";
String s5 = "Hello";
String s6 = new String( "Hello" );

System.out.println( s4==s5 );  // true
System.out.println( s4==s6 );  // false

String s5 = "Goodbye";
System.out.println( s4==s5 );  // false
```

Character Comparisons

1. Comparing two characters is based on the difference between the Unicode values of the two characters.

2. The standard Unicode values are:

```
char c = '!';

for (int i=1; i<=94; i++)
{
   System.out.printf( "[%3d] %s ", (int)c, c++ );

   if( (i+1)%10 == 0 ) System.out.println();
}
```

which displays:

```
[ 33] ! [ 34] " [ 35] # [ 36] $ [ 37] % [ 38] & [ 39] ' [ 40] ( [ 41] )
[ 42] * [ 43] + [ 44] , [ 45] - [ 46] . [ 47] / [ 48] 0 [ 49] 1 [ 50] 2 [ 51] 3
[ 52] 4 [ 53] 5 [ 54] 6 [ 55] 7 [ 56] 8 [ 57] 9 [ 58] : [ 59] ; [ 60] < [ 61] =
[ 62] > [ 63] ? [ 64] @ [ 65] A [ 66] B [ 67] C [ 68] D [ 69] E [ 70] F [ 71] G
[ 72] H [ 73] I [ 74] J [ 75] K [ 76] L [ 77] M [ 78] N [ 79] O [ 80] P [ 81] Q
[ 82] R [ 83] S [ 84] T [ 85] U [ 86] V [ 87] W [ 88] X [ 89] Y [ 90] Z [ 91] [
[ 92] \ [ 93] ] [ 94] ^ [ 95] _ [ 96] ` [ 97] a [ 98] b [ 99] c [100] d [101] e
[102] f [103] g [104] h [105] i [106] j [107] k [108] l [109] m [110] n [111] o
[112] p [113] q [114] r [115] s [116] t [117] u [118] v [119] w [120] x [121] y
[122] z [123] { [124] | [125] } [126] ~
```

3. For instance, the difference between ‘e’ and ‘b’ is:

```
char c1 = 'e';
char c2 = 'b';
char c3 = 'E';

System.out.printf( "%s-%s = %d\n", c1, c2, (int)c1-(int)c2 );
System.out.printf( "%s-%s = %d\n", c2, c1, (int)c2-(int)c1 );
System.out.printf( "%s-%s = %d\n", c1, c3, (int)c1-(int)c3 );
System.out.printf( "%s-%s = %d\n", c3, c1, (int)c3-(int)c1 );
```

which displays:

```
e-b = 3
b-e = -3
e-E = 32
E-e = -32
```

String Comparisons

1. To see if two strings have identical value, we use the equals() method, not ==.

```
String s4 = "Hello";
String s5 = "Hello";
String s6 = new String( "Hello" );
String s7 = "hello";

System.out.println( s4.equals(s5) );  // true
System.out.println( s4.equals(s6) );  // true
System.out.println( s4.equals(s7) );  // false
System.out.println( s4.equalsIgnoreCase(s7) );  // true
```

2. To determine if one string is “less than”, “equal to”, or “greater than” another string, we use the compareTo() method. This method returns a negative value if the first string is “less than” the second string, 0 if the two strings have the same content, and a positive value if the first string is greater than the second string.

```
s1 = "abc";
s2 = "abd";
s3 = "abc";
s4 = "abb";

System.out.println( s1.compareTo(s2) );  // -1
System.out.println( s1.compareTo(s3) );  // 0
System.out.println( s1.compareTo(s4) );  // 1
```

Thus, we can write statements like:

```
if( s1.compareTo(s2) < 0 )...
```

to handle the case when one string is “less than” another.

The actual return value of the compareTo() method is the difference in Unicode values for the first character where the two strings are different.

```
s1 = "abe";
s2 = "adx";
s3 = "abb";
s4 = "Aaa";

System.out.println( s1.compareTo(s2) );  // -2
System.out.println( s1.compareTo(s3) );  // 3
System.out.println( s1.compareTo(s4) );  // 32
```

Thus, we may have applications where need to know “how” much smaller one string is than another. Notice, however, that this does not tell you where the strings differ (which character).

3. Note, you cannot compare strings with the usual comparison operators, <, <=, >, >=. This will generate a compile error.

Accessing the Characters in a String

1. A string is represented internally as an array. However, you cannot use array notation to retrieve individual characters. You must use the charAt(index) method. The length of the string is found by using the length() method.

```
s = "CS is great";

System.out.println( s.length() );  // 11

for( int i=0; i<s.length(); i++ )
     System.out.print( s.charAt(i) + " " );
```

displays:

```
11
C S   i s   g r e a t
```

String Concatenation

1. You can append one string on to the end of another string in two ways. This is called string *concatenation*.

```
s1 = "Hello ";
s2 = "everyone";

s3 = s1.concat(s2);
s4 = s1 + s2;

System.out.println( s3 );
System.out.println( s4 );
```

Another example:

```
s1 = "How ";
s2 = "are ";
s3 = "you ";
s4 = "today?";

s5 = s1.concat(s2).concat(s3).concat(s4);
s6 = s1 + s2 + s3 + s4;

System.out.println( s5 );
System.out.println( s6 );
```

2. You can also use the + operator to concatenate a string and a number. In this case, the number is first converted to a string. At least one of the two operands must be a string.

```
s1 = "I am ";
int x = 20;

s2 = s1 + x;

System.out.println( s2 );
```

Obtaining Substrings

1. Suppose you wanted to copy a part of a string to another variable. Consider s1 below. Suppose that I wanted to build a new string with the (substring) value: Iron. You could do this by (a) noting the index for the "I" and the "n", (b) build a char array and then (c) use it to build a string. However, Java provides a method which does this for us, *substring(beg,end).* A new string is returned that begins at *beg* and includes all the characters up to and including index *end-1*. Thus, the length of the new string is *end-beg*.

```
s1 = "I am Iron Man";

s2 = s1.substring(0,1);
s3 = s1.substring(0,s1.length());
s4 = s1.substring(5,9);
// Last three characters.
s5 = s1.substring(s1.length()-3, s1.length());

System.out.println(s2);
System.out.println(s3);
System.out.println(s4);
System.out.println(s5);
```

Which displays:

```
I
I am Iron Man
Iron
Man
```

2. You can use the *substring(beg)* method to obtain a substring that begins at *beg* and goes to the end of the string.

```
s1 = "Have you ever?";

s2 = s1.substring(0);
s3 = s1.substring(9);
s4 = s1.substring( s1.length()-1 );

System.out.println(s2);
System.out.println(s3);
System.out.println(s4);
```

Which displays:

```
Have you ever?
ever?
?
```

Locating Substrings

1. These methods are useful for finding a the location of a character or substring in another string:

`indexOf( aChar )`	Returns the index of the first character that matches *aChar*.
`indexOf( aChar, fromIndex )`	Returns the index of the first character that matches *aChar* beginning at *fromIndex*.
`indexOf( aStr )`	Returns the index of the first character of the first substring that matches *aStr*.
`indexOf( aStr, fromIndex )`	Returns the index of the first character of the first substring that matches *aStr* beginning at *fromIndex*.
`lastIndexOf( aChar )`	Returns the index of the last character that matches *aChar*.
`lastIndexOf( aChar, endIndex )`	Returns the index of the last character that matches *aChar* ending at *endIndex*.
`lastIndexOf( aStr )`	Returns the index of the first character of the last substring that matches *aStr*.
`lastIndexOf( aStr, endIndex )`	Returns the index of the first character of the last substring that matches *aStr* ending at *endIndex*.

In all cases, the functions return -1 if the character or string is not found. Some examples:

```
System.out.println( "Welcome Home Melanie".indexOf( 'c' ) );  // 3
System.out.println( "Welcome Home Melanie".indexOf( 'C' ) );  // -1
System.out.println( "Welcome Home Melanie".indexOf( 'm', 6 ) );  // 10
System.out.println( "Welcome Home Melanie".indexOf( 'c', 6 ) );  // -1
System.out.println( "Welcome Home Melanie".indexOf( "ome" ) );  // 4
System.out.println( "Welcome Home Melanie".indexOf( "ome", 5 ) );  // 9
System.out.println( "Welcome Home Melanie".lastIndexOf( 'e' ) );  // 19
System.out.println( "Welcome Home Melanie".lastIndexOf( 'e', 16 ) );  // 14
System.out.println( "Welcome Home Melanie".lastIndexOf( "ome" ) );  // 9
System.out.println( "Welcome Home Melanie".lastIndexOf( "ome", 8 ) );  // 4
```

Examples 1

1. Write a program that finds all occurrences of the character 'p' and takes that character and the next two and builds a new string with these values. You can assume that there will always be at least two characters after any occurrences of 'p'. Example:

   The input string: "A pear is a proper fruit" results in the new string: "peaproper"

   The input string: "Apples in pipes" results in the new string: "pplplepippes"

```
public static void main(String[] args)
{

//        String s1 = "A pear is a proper fruit";
          String s1 = "Apples in pipes";

          String s2 = s1;
          int pos = -1;
          String partial = "";
          String result = "";

          while ( true )
          {
               pos = s2.indexOf( 'p' );

               if ( pos == -1 )
                    break;
               else
               {
                    partial = s2.substring( pos, pos+3 );
                    result += partial;
                    s2 = s2.substring( pos+1 );
               }
          }

          System.out.println( result );
     }
```

Another way to do this, if you didn't want to copy and/or destroy the string is to use this loop instead:

```
while ( true )
{
   pos = s2.indexOf( 'p', pos+1 );

   if ( pos == -1 )
      break;
   else
   {
      partial = s2.substring( pos, pos+3 );
      result += partial;
   }
}
```

Global Modification of Strings

1. These methods are useful for globally modifying a string:

`toLowerCase()`	All characters are made lower case
`toUpperCase()`	All characters are made upper case
`trim()`	Blank characters are trimmed from either side
`replace(oldChar, newChar)`	Replaces all occurrences of *oldChar* with *newChar*
`replaceFirst(oldString, newString)`	Replaces the first occurrence of *oldString* with *newString*
`replaceAll(oldString, newString)`	Replaces all occurrences of *oldString* with *newString*

Some examples:

```
s1 = "  John     ";

s2 = s1.trim();

System.out.println( s1.length() );  // 11
System.out.println( s2.length() );  // 4

s1 = "Comz hzar Unclz John's Band";

s2 = s1.replace( 'z', 'e' );

System.out.println( s2 );  // Come hear Uncle John's Band
```

```
s1 = "DanBrown";

s2 = s1.replaceFirst( "n", "n Q. " );

System.out.println( s2 );  // Dan Q. Brown

s1 = "AOXOMOXOA";

s2 = s1.replaceAll( "OX", " great " );

System.out.println( s2 );  // A great OM great OA
```

Examples 02

1. Write a program that removes all the spaces in a string. Example:

   The input string: "A pear is a proper fruit" results in the new string: "Apearisaproperfruit"

```
public static void main(String[] args)
{
     String s1 = "A pear is a proper fruit";

     s1 = s1.replaceAll( " ", "" );

     System.out.println( s1 );
}
```

2. Write a program that counts the number of occurrences of the string, "pea" and replaces each such occurrence with #'s where the number of #'s is the count. Example:

   The input string: `peach peal`

   results in the new string: `##ch ##l`

   The input string: `"peace for peas, peaches, pears appeases pearl"`

   results in the new string:

```
######ce for ######s, ######ches, ######rs ap######ses ######rl
```

Code:

```
    public static void main(String[] args)
    {
//     String s = "peace for peas, peaches, pears appeases pearl";
       String s = "peach peal";
       String searchFor = "pea";

       int count = numOccurrences( s, searchFor );

       String replWith = buildReplacementString( count );

       s = s.replaceAll( searchFor, replWith );

       System.out.println( s );
    }

public static int numOccurrences( String s, String s1 )
{
   int pos = 0;
   int count = 0;

   while( pos < s.length()-3 )
   {
      pos = s.indexOf( s1, pos );

      if ( pos == -1 )
         break;
      else
      {
         count++;
         pos += 3;
      }
   }
   return count;
}

public static String buildReplacementString( int count )
{
   String repl = "";
   String s = "#";

   for( int i=0; i<count; i++ )
      repl += s;

   return repl;
}
```

Converting Strings to chars

1. We saw earlier that we could obtain the individual chars in a string by using the charAt() method. For instance, we could create a char array from the characters in a string:

```
String s = "Brady Bunch";

char[] chars = new char[ s.length() ];

for( int i=0; i<s.length(); i++ )

     chars[i] = s.charAt(i);
```

2. However, the String class has a method to do this for you, toCharArray().

```
s1 = "In the strangest of places";

char[] chars = s1.toCharArray();

for( char c : chars )
     System.out.print( c );
```

Examples 03

1. Write a method that accepts a string and returns the string reversed.

```
public static String reverseString( String s )
{
   char c;
   char[] chars = s.toCharArray();;

   for( int i=0, j=s.length()-1; i<s.length()/2; i++, j-- )
   {
      c = chars[i];
      chars[i] = chars[j];
      chars[j] = c;
   }

   String revString = new String( chars );

   return revString;
}
```

Or, another way:

```
public static String reverseString2( String s )
{
   String revString = "";

   for( int j=s.length()-1; j>=0; j-- )
   {
      revString += s.charAt(j);
   }
   return revString;
}
```

Comparing chars

1. We can use the regular comparison operators (<,<=,>,>=) with characters:

```
int sum=0;
for( char c : chars )
     if ( c != ' ' )
          if ( c <= 'c' )
               sum++;
```

Converting Numbers and chars to String

1. The static method, valueOf(var) returns a String representation of the input, *var,* where *var* is char, char[], double, float, long, int, short, byte, boolean.

```
char[] chars = s1.toCharArray();
double y = 22.3343;

System.out.println( String.valueOf( chars ) );
System.out.println( String.valueOf( y ) );
```

The Character Class

1. The Character class is a *wrapper* class for the primitive data type, char. You can create a Character from a char like this:

```
char c = 'd';

Character ch = new Character( c );
```

2. These are some useful methods of the *Character* class. Notice that the most useful methods are the static methods.

Character
Character(value : char)
charValue() : char
compareTo(c : Character) : int
equals(c : Character) : boolean
isDigit(c : char) : boolean
isLetter(c : char) : boolean
isLetterOrDigit(c : char) : boolean
isLowerCase(c : char) : boolean
isUpperCase(c : char) : boolean
toLowerCase(c : char) : char
toUpperCase(c : char) : char

Examples 04

1. Write a method that determines whether an input string (a password) has length at least six, all alpha-numeric characters, and with at least three letters and at least three digits.

```
public static boolean isValidPassword( String password )
{
   int i = 0, numDigits = 0, numLetters = 0;
   char c;

   if ( password.length() < 6 ) return false;

   while ( i < password.length() )
   {
      c = password.charAt(i++);

      if ( Character.isDigit(c) )
        numDigits++;
      else if ( Character.isLetter(c) )
         numLetters++;
      else
         return false;
   }

   if ( ( numDigits > 2 ) && ( numLetters > 2 ) )
      return true;
   else
      return false;
}
```

2. Write a method that determines whether an input sequence has the form: xyxxyxxxyxxxxy..., where x is a letter and y is a digit. This problem can more neatly be solved by noting that the sequence must have length identically equal to $\frac{n(n+3)}{3}$, for some integer value of *n* (*e.g.* the length must be 2, 5, 9, 14, 20,..... If we ensure that the length is ok first, then this makes checking for the pattern simpler.

   Algorithm:

```
if ( !isCorrectLength(s) ) Return false

check for 1 letter,  then digit
check for 2 letters, then a digit
check for 3 letters, then a digit
etc.
```

```
public static boolean isValidSequence( String sequence )
{
   int i=0, numLetters = 1;
   char c;

   if( !isCorrectLength( sequence ) )
      return false;

   while ( i < sequence.length() )
   {
      for( int j=0; j<numLetters; j++ )
      {
         c = sequence.charAt(i++);

         if( !Character.isLetter(c) ) return false;
      }

      c = sequence.charAt(i++);

      if( !Character.isDigit(c) ) return false;

      numLetters++;
   }
   return true;
}

public static boolean isCorrectLength( String sequence )
{
   int sum = 0;
   int n=0;

   while( sum < sequence.length() )
   {
      sum = n*(n+3)/2;
      n++;
   }

   if( sum == sequence.length() )
         return true;

   return false;
}
```