The Class, String

There are five programs in the class code folder Set17. The first one, String1 is discussed below. The folder StringInput shows simple string input from the keyboard.

Processing is really Java. It is referred to as a "super set " of Java because it is all of Java and more. Much of the java aspects are hidden from view by the Processing IDE. We will look at some of the hidden stuff next class meeting.

The main organizing structure of Java - and Processing - is the class. One main reason for using a class structure is simply the size and complexity of the language. A way to consider this is to think about a library. It is composed of books - lots and lots of books. But, what if every book in the library was pasted together into a single large book? That's right - one book. Think of the ramifications:

- only one person could check it out at a time
- it would be difficult to find something specific
- its shear size and weight would make it very cumbersome and be difficult to work with in any way

This same list and more is one reason that Java is broken up into smaller units or segments. These segments are called classes. The class list in the Java API has over 2000 entries.

We have been programming inside a class since we wrote the first program on day 1. We will look at this next time.

As a first look at a class, we will look at the **String** class of Java. Here are the high points:

1. Anything inside quotation marks **(" ")** is a **String**. Think of a string of pearls, a string of beads, a string of characters. Here are some examples of **String**s:

   "" the empty string
   " " the space as a string
   "hello world" which is one of the most famous **String**

2. Strings are NOT arrays. They are **references** to pieces of memory called **objects** or in this discussion, **String** objects.

3. The characters in a **String** object are indexed like arrays beginning at **zero** (but they are not arrays).
4. There is a **length** field in the **String** object but it is **private** - we cannot directly access it. To find the length of a **String**, we must use the **length( )** function.
5. The last character in a **String** has an index of **length( )-1**

The processing API lists seven **String** functions (In Java, functions have a different name - they are methods. sigh… functions, methods, parameters, arguments, sigh again…). that may be useful to you. These are used in the **classcodeA** program . The JAVA API has over 2000 functions (methods) listed for the class. All of these can be used by you in a Processing program because Processing is really Java.

Here is a list of comparisons and contrasts between an array of **char** and a **String**;

Category	Array of char	String
declaration	char [ ] c;	String s;
initialization	char [ ] c = {'a', 'b'};	String s = "hello";
access	println(c[0]);	println(s.charAt(0));
Changing values	c[0] = 'x';	Illegal for Strings
Size	println(c.length);	println(s.length( ));
Index of beginning character	println(c[0]);	println(s.charAt(0));
Index of last character	println(c[c.length - 1]);	println(s.charAt(s.length( ) - 1));

Equality	==
	compiles, runs, and does not work properly
		==
		compiles, runs, and usually does not work properly
		
		if (s.equals(s1))
		{
		
		}
		compiles, runs and works properly

One question that is asked is how can the **toUpperCase( )** and **toLowerCase( )** functions be used. One way to use them is when we are dealing with user input (discussed later in these notes).

If we ask the use to type either **yes** or **no** as a reply to a question the user can type any of the following to correctly respond:

yes Yes YES yEs yeS YeS no No NO nO

Any of these are proper inputs. Yet we do not want to have to type:

```
if ( s1.equals( "YES" ) ||
     s1.equals( "yes" ) ||
     s1.equals( "yES" ) ||
     s1.equals( "yeS" ) ||
     s1.equals( "YeS" ) ||
     s1.equals( "yEs" )  )
   {
       . . .
   }
```

This is silly. What if we were asking for the capital of the state of Florida? We can use the case changing functions to simplify the code:

```
   if ( s.toUpperCase().equals("YES" ) )
   {
     . . .
   }
```

The function **toUpperCase()** returns a new **String** and we can use the **equals( )** function with the returned **String**. This does look a bit bizarre, but there is a simpler way to write this code:

```
   String temp = s.toUpperCase( );
   if ( temp.equals("YES" ) )
   {
     . . .
   }
```

Arrays of String References

We can build arrays of any type of data so we can build an array of **String**s. The syntax is the same:

```
   String [ ] labels = { "On", "Off", "In", "Out" };
```

String Input

We have used single character input for weeks. We have done this in the **keyPressed( )** function. The advantage of this is that the program does not pause and wait for the input. If we have no animation ongoing when we want input, a pause in the action is fine. But, if there is animation occurring, a pause for input is not good.

Processing provides no other direct way to get user input and we do not want to explore Java's ways of getting user input so we will keep using the **keyPressed()** function to do our work.

Here is part of the code in A_BoardNotes that gets input from the user:

Code	Comments
void keyPressed( ) {	

if (keyCode == DELETE ||
      keyCode == BACKSPACE)
	If we assume the user might not type things correctly and will want to correct their input, we have to account for the delete (Mac) or backspace (Windows) keys. These keys are CODED so we check the keyCode value first.

{
    if (s.length() >0)
    {
	If the user typed one of these coded keys, we must first test the String to insure it is not empty. Our code will crash is we try to delete a character from an empty String.

s = s.substring
          (0,
          s.length( )-1);
    }
  }
	We cannot alter a String so we have to make an new String without the last character. The substring( ) function does this. The details of the parameters are discussed below.

`else`
`{`
`   s = s + key;`
`}`
`}`
	If the user did not type the coded keys tested for above, this assumes they typed a character and a new String is made from the old String and the new letter using the concatenation operator +

The parameters for the call of the **substring()** *function need some explanation:*

```
   s = s.substring (0,  s.length( )-1);
```

The last character in the **String** *has an index of :*

```
        s.length( )-1
```

If we want a new **String** *created that does not have the last character, shouldn't we use the following for the second parameter:*

```
   s = s.substring (0,  s.length( )-2);
```

This is one time that our logic fails us. When we specify a range of values, there are many functions that do not include the second value in the range. For example, if we call the **random( )** *function like this:*

```
   float f = random( 10, 20 );
```

we get values between:

10.0 and 19.9999999

We do not get 20 in the range of returned values.

In fact when we draw a line like this:

```
   line( 0, 0, 400, 400 );
```

The line is drawn from pixel **(0, 0)** *to pixel (* **399, 399** *).*

In the parameter list for **substring( )**:

```
     s = s.substring (0,  s.length( )-1);
```

The new **String** *that is returned has character from index* **0** *up to* <u>*BUT NOT INCLUDING the character at the index of the second parameter.*</u>