AP Computer Science Name: _____
Chapter 9 Notes (1)

Strings

In Java, a string of characters is represented by an object of the `String` type. String objects are treated pretty much like any other type of objects: they have constructors and methods, and they can be passed to other methods (always as references) or returned from methods. But, they are different in two respects: the Java compiler knows how to deal with *literal strings* (repress-ented by text in double quotes), and the `+` and `+=` operators can be used to concatenate a string with another string, a number, or an object.

Literal Strings:

Literal strings are written as text in double quotes. The text may contain escape characters. Recall that the backslash character `'\'` is used as the "escape" character: inside a literal string `\n` stands for "newline," `\'` represents a single quote, `\"` represents a double quote, and `\\` represents a backslash. For example:

```
String pathname = "C:\\Ch10\\funny.txt";
                                    // meaning C:\Ch10\funny.txt
```

A literal string can be empty, too, if there is nothing between the quotes.

```
String s = "";      // empty string
```

Literal strings act as `String` objects, but they do not have to be created – they are "just there" when you need them. The compiler basically treats a literal string as a reference to a `String` object with the specified value that is stored somewhere in memory. If you want, you can actually call that object's methods (for example, `"Internet".length()` returns 8). A declaration

```
String city = "Boston";
```

sets the reference `city` to a `String` object `"Boston"`. Note that `Boston` here is not the <u>name</u> of the variable (its name is city) but its <u>value</u>.

`String` Constructors and Immutability:

The `String` class has over a dozen constructors, but it is less common to use constructors for strings than for other types of objects. Instead, we can initialize `String` variables either to literal strings or to strings returned from `String`'s methods.

One of the constructors, `String()`, takes no parameters and builds an empty string; rather than invoking this constructor with the new operator, we can simply write:

```
String str = "";    // str is initialized to an empty string
```

Another constructor is a copy constructor `String(String s)`, which builds a copy of a string `s`. But in most cases we do not need to make copies of strings because, as we'll explain shortly, strings, once created, never change; so instead of copying a string we can just copy a reference. For example:

```
String str = "Foo Fighters";
```

This is not exactly the same as

```
String str = new String("Foo Fighters");
```

but, as far as your program is concerned, these two declarations of `str` act identically.

Other constructors create strings from character and byte arrays. They are potentially useful, but not before we learn about arrays.

There is a big difference between an empty string and an uninitialized `String` reference.

Empty strings are initialized to "" or created with the no-args constructor, as in

```
String s1 = "";               // s1 is set to an empty string
String s2 = new String();     // s2 is set to an empty string
```

A field of the `String` type iset to `null` by default:

```
private String s3;       // instance variable s3 is set to null
```

You can call methods for an empty string, and they will return the appropriate values. For example, `s1.length()` returns 0, and `s2.equals("")` returns `true`. But if a method is called for a reference that is equal to `null`, the program throws a `NullPointerException` and quits.

◇ ◇ ◇

Once a string is constructed, it cannot be changed! If you look carefully at `String`'s methods, summarized down below, you will notice that none of these methods changes the content of a string.

A string is an *immutable* object: none of its methods can change the content of a `String` object.

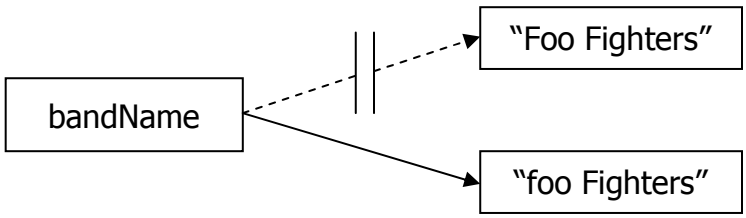
For example, you can get the value of a character at a given position in the string using the `charAt` method. But there is no method to set or replace one character in a string. If you want to change, say, the first character of a string from upper to lower case, you have to build a whole new string with a different first character. For example:

```
String bandName = "Foo Fighters";
char c = bandName.charAt(0);
bandName = Character.toLowerCase(c) + bandName.substring(1);
     // bandName now refers to a new string
     //   with the value "foo Fighters"
```

This code changes the reference – `bandName` now refers to your new string with the value "`foo Fighters`".

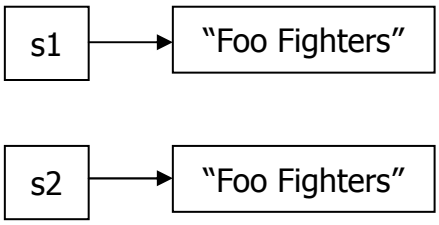
The old string is thrown away (unless some other variable refers to it). Java's automatic garbage collector releases the memory from the old string and returns it to the free memory pool. This is a little wasteful – like pouring your coffee into a new mug and throwing away the old mug each time you add a spoonful of sugar or take a sip.

```
String bandName = "Foo Fighters";
char C = bandName.charAt(0);
bandName = Character.toLower(c) + bandName.substring(1);
```



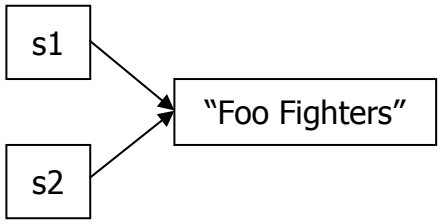
However, the immutability of strings makes it easier to avoid bugs. It allows us to have two `String` variables refer to the same string without the danger of changing the string contents through one variable without the knowledge of the other. In some cases it also helps avoid copying strings unnecessarily. Instead of creating several copies of the same string –

```
String s2 = new String(s1);   // s2 refers to a new copy of s1
```



as in the figure below – you can use

```
String s2 = s1;          // s2 refers to the same string as s1
```



On the other hand, if you build a new string for every little change, a program that frequently changes long strings, represented by `String` objects, may become slow.

Fortunately Java provides another class for representing character strings, called `StringBuffer`. `StringBuffer` objects are not immutable: they have the `setCharAt` method and other methods that change their contents. `Strings` may be easier to understand, and they are considered safer in student projects. However, with the `StringBuffer` class we can easily change one letter in a string without moving the other characters around. For example:

```
StringBuffer bandName = new StringBuffer("Foo Fighters");
char c = bandName.charAt(0);
bandName.setCharAt(0, Character.toLowerCase(c));
     // bandName still refers to the same object, but its first
     // character is not 'f';
```

If some text applications run rather slowly on your fast computer, it may be because the programmer was too lazy to use (or simply never learned about) the `StringBuffer` class. `StringBuffer` constructors and methods are summarized later. Make sure you've read it before writing commercial applications!