Strings

Is a string an object or a primitive data type? This question has a clear answer — a string is an object! — however, the way strings typically appear in Java programs can lead to confusion. There are shortcuts available for strings that are not available for other objects in Java. These shortcuts are highly used and they give strings the appearance of a primitive data type. But, strings are objects and they possess all the attributes and behaviours of objects in Java.

In this section, we will formally introduce strings. We'll learn how to declare and initialize `String` objects, and we'll learn how to manipulate and access strings through operators and methods.

Since strings are objects, we are, in a sense, formally introducing the concept of objects in Java, and we'll begin by presenting "strings as objects". (String shortcuts are discussed later.) In the following paragraphs, you can replace the word "`String`" by the name of any other class in Java and the story line still makes sense. So, when we talk about "instantiating a `String` object", we could just as easily be talking about "instantiating a `BufferedReader` object" or "instantiating a `PopupMenu` object". Keep this in mind and you will be well-prepared to extend your knowledge of strings to the wider and more general concept of objects.

Note in the previous paragraphs the use of the terms "string" and "`String`". In a general sense, we'll refer to a collection of characters grouped together as a "string". When referring to the same collection of characters as a formal entity in the Java language, we'll refer them as a "`String` object" or "an object of the `String` class".

Strings as Objects

A string is collection of characters grouped together. For example, "`hello`" is a string consisting of the characters `'h'`, `'e'`, `'l'`, `'l'`, and `'o'`. Note that a character constant is enclosed in single quotes, whereas a string constant is enclosed in double quotes. In Java, a string is an object. As with primitive data types, an object doesn't exist until it is declared. The following statement declares a `String` object variable named `greeting`:

```
String greeting;
```

The first hint that we are not dealing with a primitive data type, is that the word "`String`" begins with an uppercase character `'S'`. The names of Java's primitive data types all begin with lowercase letters (e.g., `int`, `double`). Indeed, class names in Java all begin with an uppercase letter. (Note that this is a convention, rather than a rule.)

The effect of the statement above is to set aside memory for an object variable named `greeting`. However the memory set aside will not hold the characters of the `greeting` string; it will hold a *reference* to the string. This important point deserves special emphasis:

An object variable holds a reference to an object!

Since the declaration does not initialize the `greeting` string, the `greeting` reference is initialized with **`null`** — because it refers to "nothing". Note that `null` is a reserved word in Java. This is illustrated in Figure 1a.

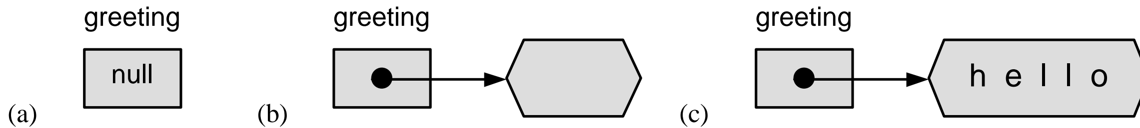

Figure 1. Creating a `String` object (a) declaration only (b) instantiating with default constructor (c) instantiating with constructor containing a string literal.

To set aside memory to hold the string, we must construct a `greeting` object. This job is performed by a type of method called a *constructor*. The process of constructing an object is called *instantiating an object*. This is performed as follows:

```
greeting = new String();
```

Declaring and instantiating a `String` object can be combined in a single statement:

```
String greeting = new String();
```

Note that the constructor method has the same name as the class. In fact, this is a rule of the Java language. The reserved word "`new`" signals to the compiler that new memory is to be allocated.

Since a constructor is a method, it is followed by a set of parentheses containing arguments passed to it. In the example above, the constructor contains no arguments. When a constructor is used without arguments, it is called a *default constructor*. In the case of the default `String` constructor, memory is set aside for the content of the string, but there are no characters in the string. This is an *empty string*, as shown in Figure 1b. The arrow emphasizes that `greeting` holds a reference, not a value. It points to the object to which `greeting` refers.

A more common way to construct a `String` object is to pass a literal — a string constant —as an argument to the constructor:

```
String greeting = new String("hello");
```

The statement above fulfills the complete task of instantiating a `String` object variable named `greeting` and initializing it with a reference to a `String` object containing "`hello`". This is illustrated in Figure 1c. Note in Figure 1, the use of a rectangle to denote a variable (in this case an object variable) and a hexagon to denote an object.

So, now that we have a `String` object variable and a `String` object, what can we do with them? Of course, the string can be printed. The statement

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

prints "`hello`" on the standard output.

Perhaps the most fundamental of all Java operators is assignment. One string can be assigned to another, just as one integer can be assigned to another integer. Consider the following Java statements:

```
// assignment of integers
int x;
int y;
x = 5;
y = x;

// assignment of strings
String s1;
String s2;
s1 = new String("Java is fun!");
s2 = s1;
```

If you followed the preceding discussion carefully, it is apparent that these two sets of statements are slightly different. Although two `String` object variables are declared, only one `String` object is instantiated. The assignment in the final line above makes a copy of a reference to a `String` object. It does not copy the object! This is illustrated in Figure 2.

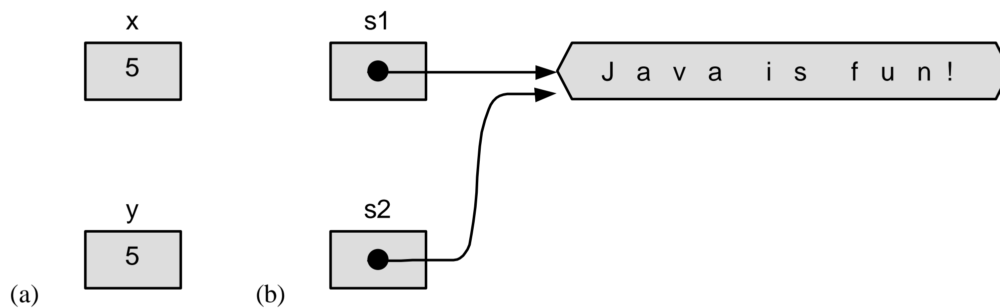

Figure 2. Assignment of (a) integers (b) strings

This example underscores a key difference between an object and a primitive data type. A primitive data type is *stored by value*, whereas an object is *stored by reference*. Stated another way, a primitive data type variable holds a value, whereas an object variable holds a reference to an object.

If you really want to make a copy of a `String` object, you can do so as follows:

```
String s1 = new String("Java is fun!");
String s2 = new String(s1);
```

Clearly, two `String` objects have been instantiated, because the constructor `String()` appears twice. In the second line the argument passed to the constructor is `s1`. The effect is to instantiate a new `String` object and initialize it with a copy of the data in `s1`. This is illustrated in Figure 3.

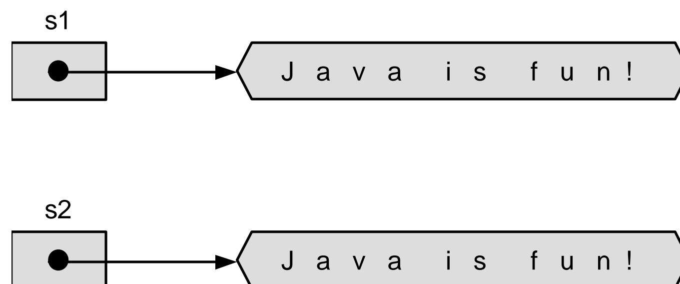

Figure 3. Copy `String` objects

 © Scott MacKenzie

Strings as Primitive Data Types

Next to numbers, strings are the most common type of data manipulated in computer programs. Because of this, `String` objects are given somewhat special treatment in the Java language. A set of shortcuts or special operations exist for `String` objects that do not exist for most other objects. These shortcuts are extremely handy, but they can also confuse students learning to program in Java. In this section we will examine how strings are commonly treated in Java. Please bear in mind that, for the most part, these discussions do not apply to other types of objects. They are unique to `String` objects.

The following shortcut provides an alternate, simplified way to instantiate a `String` object:

```
String greeting = "hello";
```

This looks very much like declaring and initializing a primitive data type, like an `int`, but don't be fooled. The statement above is just a shortcut for

```
String greeting = new String("hello");
```

This is simple enough. Let's move on to the next shortcut. Once a primitive data type, like an `int`, is assigned a value, it can be assigned a new value, and this new value replaces the old one. Similarly, once a `String` object has been instantiated and assigned a value, it can be assigned a new value, and the new value replaces the old value. Well, sort of. Let's explore this idea with a demo program. The program `DemoStringRedundancy.java` is shown in Figure 4.

```
 1  public class DemoStringRedundancy
 2  {
 3     public static void main(String[] args)
 4     {
 5        // change the value of an int variable
 6        int x = 5;
 7        x = 6;
 8        System.out.println(x);
 9
10        // change the value of a String object
11        String greeting = "hello";
12        greeting = "bonjour";
13        System.out.println(greeting);
14     }
15  }
```

Figure 4. `DemoStringRedundancy.java`

The output from this program is as expected:

```
6
bonjour
```

The statements in `DemoStringRredundancy` appear innocent enough, but something important is taking place. Replacing the value of an `int` variable with a new value does not require new memory. The new value overwrites the old value and that's the end of it. This is illustrated in Figure 5.

Figure 5. Memory allocation in `DemoStringRedundancy`
(a) after line 6 (b) after line 7

This is not the case with strings. When a `String` object is assigned a new value, the new value often differs from the old value. The string "`hello`" contains five characters, whereas the string "`bonjour`" contains seven characters. So, space for an extra two characters is needed. Where does this space come from? Let's answer this question by first presenting the following rule for `String` objects in Java:

A String object is immutable.

By "immutable", we mean that a `String` object, once instantiated, cannot change. We can perform all kinds on "read" operations on a `String` object, like determine its size or count the number of vowels, but we cannot change the content of a `String` object. The statements in `DemoStringRedundancy` are perfectly legal, and they certainly give the impression that the content of the `String` object `greeting` is changed from "`hello`" to "`bonjour`". But there is more taking place than meets the eye. All the relevant action for this discussion is in line 12. The effect of line 12 is this:

1. A new `String` object is instantiated with the `String` literal "`bonjour`".
2. The object variable `greeting` is assigned a reference to the new `String` object. (The old reference is overwritten.)
3. The old `String` object containing "`hello`" is redundant.[1]

The old `String` object is redundant because there is no longer an object variable referencing it. This is illustrated in Figure 6.

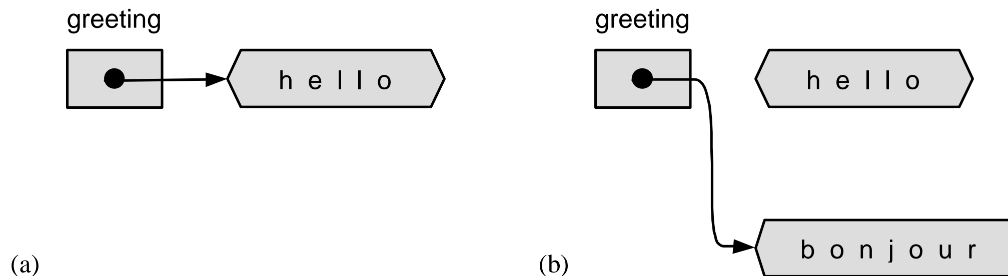

Figure 6. Memory allocation in `DemoStringRedundancy` (a) after line 11 (b) after line 12

[1] Because the original "hello" string appeared as a literal, it continues to occupy space as the program executes. Often, strings are created during the execution of a program. When such strings become redundant, the spaced occupied is eventually collected by a background process known as "garbage collection". This important service ensures that vast amounts of space are not vacated and lost as a program executes. Garbage collection occurs automatically by the method `gc()` in Java's `Runtime` class.

String Concatenation

Next to assignment, one of the most common string operations is joining two strings to form a new, larger string. This operation is called *concatenation*. Once again, `String` objects are given somewhat special treatment in Java. Strings are most-commonly concatenated using the + operator as a shorthand notation.[2] For example, the expression

```
"happy" + "birthday"
```

concatenates two strings to form a new string. The string "`birthday`" is appended to the end of the string "`happy`" to form "`happybirthday`". Forgetting to consider spaces is a common programming slip when concatenating strings. The following expression

```
"happy" + " " + "birthday"
```

gives the intended result, concatenating three string to form a new string containing "`happy birthday`".

Concatenation can involve string literals as well as `String` object variables and assignment. The statements

```
String s1 = "happy";
String s2 = " ";
String s3 = "birthday";
String name = "Billy";
String greeting = s1 + s2 + s3 + s2 + name;
```

declare and instantiate five `String` object variables and five `String` objects. The last line declares a `String` object variable named `greeting` and initializes it with a reference to a `String` object containing "`happy birthday Billy`".

If one of the arguments in a string concatenation expression is a string and the other is an `int` or a `double`, the numeric value is converted to a string as part of the concatenation. Thus, the following statements

```
int x = 53;
String answer = "The answer is " + x;
```

instantiate a `String` object containing "`The answer is 53`". It follows that a convenient way to convert a numeric variable to its string equivalent is by concatenating it with an empty string. For example, if `x` is an `int` variable containing 53, the expression

```
"" + x
```

yields a string containing "`53`".

The behaviour of the + operator is an example of *operator overloading*, as noted earlier. It is means "addition" when both operands are numeric variables, but it means "concatenation" when either operand is a string. In other words, the operation depends on the context. A brief example should suffice. The expression

---

[2] The string concatentation operator (+) is implemented through the `StringBuffer` class and its `append()` method. See the discussion of the `StringBuffer` class for more details.

```
53 + 7
```

uses the + operator for addition. Two integers are added to yield an integer result, 60. On the other hand, the following expression

```
"53" + 7
```

uses the + operator for concatenation, because one of the operands is a string. The other operand is an integer, but it is converted to a string during concatenation. The result of the expression is a string equal to "537".

String Methods

Other than assignment and concatenation, string operations usually employ methods of the `String` class. Including all variations, there are over 50 methods in the `String` class; however, we will only examine the most common. These are listed in Table 1.

Table 1. Common methods in the `String` class

Method	Purpose
`String(s)`	constructs a `String` object and initialize it with the string `s`; returns a reference to the object
`s.length()`	returns an `int` equal to the length of the string `s`
`s.substring(i, j)`	returns a `String` equal to a substring of `s` starting at index `i` and ending at index `j - 1`
`s.toUpperCase()`	returns a `String` equal to `s` with letters converted to uppercase
`s.toLowerCase()`	returns a `String` equal to `s` with letters converted to lowercase
`s.charAt(i)`	returns the `char` at index `i` in string `s`
`s1.indexOf(s2)`	returns an `int` equal to the index within `s1` of the first occurrence of substring `s2`; return `-1` if `s2` is not in `s1`
`s1.compareTo(s2)`	returns an `int` equal to the lexical difference between string `s1` and `s2`; if `s1` lexically precedes `s2`, the value returned is negative
`s1.equals(s2)`	returns a `boolean`; `true` if `s1` is the same as `s2`, `false` otherwise

Method Signatures

Methods are often summarized by their *signature*, which shows the first line of the method's definition. For example, the signature for the `compareTo()` method, as given in the documentation for the `String` class, is

```
public int compareTo(String anotherString)
```

The reserved word "`public`" is a *modifier*. It identifies the method's *visibility* in the larger context of a Java program. A `public` method can be accessed, or "called", from outside the class in which it is defined; it is visible anywhere the class is visible. (A method is declared **`private`** if it helps a public method achieve its goal but it is not accessible outside the class.)

The reserved word "`int`" identifies the data type of the value returned by the method. The `compareTo()` method returns an integer. A method can also return an object; so a class name may appear instead of a data type.

Next comes the name of the method. It is a Java convention that names of methods always begin with a lowercase letter, as noted earlier. If the method's name is composed of more than one word, the words are run together and subsequent words begin with an uppercase character. Hence, `compareTo()` begins with a lowercase letter; but uses uppercase "`T`" in "`To`". Constructor methods are an exception, since they always use the same name as the class, and therefore always begin with an uppercase character.

Following the method's name is a set of parentheses containing the arguments passed to the method separated by commas. Each argument is identified by two words. The first is its type; the second is its name. The name is arbitrary and need not match the name of an argument in the program that uses the method.

String Methods

The demo program `DemoStringMethods.java` exercises all the methods in Table 1. The listing is given in Figure 7.

```
 1  public class DemoStringMethods
 2  {
 3     public static void main(String[] args)
 4     {
 5        String s1 = new String("Hello, world");
 6        String s2 = "$75.35";
 7        String s3 = "cat";
 8        String s4 = "dog";
 9
10        System.out.println(s1.length());
11        System.out.println(s1.substring(7, 12));
12        System.out.println(s1.toUpperCase());
13        System.out.println(s1.toLowerCase());
14        System.out.println(s1.charAt(7));
15        System.out.println(s2.indexOf("."));
16        System.out.println(s2.substring(s2.indexOf(".") + 1, s2.length()));
17        System.out.println(s3.compareTo(s4));
18        System.out.println(s3.equals(s4));
19     }
20  }
```

Figure 7. `DemoStringMethods.java`

This program generates the following output:

```
12
world
HELLO, WORLD
hello, world
w
3
35
-1
false
```

We'll have many opportunities the meet these methods later in more interesting programs. For the moment, let's just examined the basic operation of each.

String()

In line 5, the `String()` constructor is used to instantiate a `String` object. A reference to the object is assigned to the object variable `s1`. In fact, `String` objects are rarely instantiated in this manner. It is much more common to use the shorthand notation shown in lines 6-8.

length()

In line 10, the `length()` method determines the length of `s1`. The syntax shown in line 10 illustrates an "instance method" operating on an "instance variable". Dot notation connects the instance variable (`s1`) to the instance method (`length()`). `s1` is called an instance variable because the object to which it refers is an "instance of" an object — a `String` object.

The term `"s1.length()"` is an expression. It returns an integer value equal to the length of the string. So, the term `"s1.length()"` can be used anywhere an integer can be used, such as inside the parentheses of the `println()` method. The length of `s1` is 12, as shown in the 1st line of output.

substring()

In line 11, a substring is extracted from `s1`. The substring begins at index 7 in the string and ends at index 11. In counting-out character positions within strings, the first character is at index "0". This is illustrated in Figure 8.

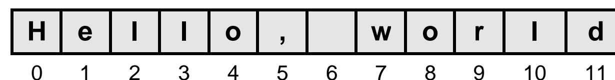

Figure 8. Characters positions within a string

Note that the second argument in the `substring()` method is the index "just passed" the last character in the substring. So, to extract the substring "`world`" from "`Hello, world`", the arguments passed to the `substring()` method are 7 and 12, respectively. The 2nd line of output shows the result: `world`.

Since the first position in a string is at index 0, the size of the string returned by the `length()` method is "one greater than" the last index in the string.

It is illegal to access a non-existent character in a string. This would occur, for example, by using the `substring()` method on an empty string:

```
String s1 = "";
String s2 = s1.substring(0, 1);
```

When these statements execute, a `StringIndexOutOfBounds` exception occurs and the program crashes with a run-time error (unless the exception is caught and handled explicitly in the program).

toUpperCase () and toLowerCase()

Line 12 demonstrates a simple conversion of a string to uppercase characters. Non-alphabetic characters are left as is; alphabetic characters are converted to uppercase. Note that the object referenced by the instance variable, `s1` in this case, remains unchanged. The method simply returns a reference to a new `String` object containing only uppercase characters. If, by chance,

the string does not contain any lowercase characters, then no conversion is necessary and a reference to the original object is returned.

Conversion to lowercase characters, shown in line 13, is similar to the process described above, except uppercase characters are converted to lowercase. The result of the case conversions is shown in the program's output in the 3rd line ("`HELLO, WORLD`") and 4th line ("`hello, world`").

charAt()

In line 14, the `charAt()` method is used to determine which character is at index 7 in the string referenced by `s1`. As evident in Figure 8, the answer is "`w`" and this is shown in the 5th line of output.

indexOf()

In line 15, the `indexOf()` method is demonstrated. The method is called on the instance variable `s2`, which references a `String` object containing `"$75.35"` (see line 6). The decimal point ("`.`") is at index 3, as shown in the 6th line of output.

Line 16 demonstrates the convoluted sorts of operations that can be performed with methods of the `String` class. The statement retrieves the decimal places from the `String` object referenced by instance variable `s2`. Have a close look at the syntax as see if you agree with the result shown in the 7th line of output.

compareTo()

In line 17 the `compareTo()` method is demonstrated. As noted in Table 1, the method performs lexical comparison between two strings and returns an integer equal to the lexical difference between the two strings. This sounds complex, but it's really quite simple. "Lexical order" is, for the most part, "dictionary order"; so "able" precedes "baker", which in turn precedes "charlie", and so on.

The lexical difference between two strings is simply the numerical difference between the Unicodes of the characters at the first position that differs in the two strings. For, example the expression `"able".compareTo("baker")` equals -1. The first characters differ and the numerical difference between them is one (1). Since `'a'` precedes `'b'` the value returned is minus one (-1). The expression `"aardvark".compareTo("able")` also equals -1, but this is the lexical difference between the second two characters (since the first two are the same). In the example program, the comparison is between "`cat`" and "`dog`" (see line 17). The lexical difference, -1, appears in the 8th line of output.

There are a couple of situations not accounted for in the discussion above. The numerical difference between two characters that differ only in case (e.g., `'a'` and `'A'`) is 32, with uppercase characters lexically preceding lowercase characters. So, for example, the expression `"zebra".compareTo("ZEPHYR")` equals 32. Comparisons that involve digits or punctuation symbols are a little trickier to evaluate. The Unicodes for these and other special symbols are given in Appendix A.

If there is no index position where the two strings differ, yet one string is shorter than the other, then the shorter string is considered to lexically precede the longer string. The value returned is the difference in the lengths of the two strings. So, for example, the expression

```
"lightbulb".compareTo("light")
```

equals 4 and

```
"light".compareTo("lightbulb")
```

equals -4.

equals()

The last string method demonstrated in Figure 7 is the `equals()` method. `equals()` is a similar to `compareTo()` except it only tests if for equality or inequality. The return type is a `boolean`: `true` if the two strings are identical, `false` otherwise. Since "cat" is not equal to "dog", the result is `false`, as seen in the 9th line output.

Note that the relational test for equality(==) is a valid string operator; however, its behaviour is distinctly different from the `equals()` method. For example, the expression

```
s1 == s2
```

returns `true` if `s1` and `s2` refer to the same object, where as the expression

```
s1.equals(s2)
```

returns `true` if the character sequence in `s1` is the same as in `s2`.