Chapter

12

Strings

Strings are just another type of collection, one that consists of a sequence of characters. However, we tend to use strings enough to warrant a chapter of their own. *String* is subclassed off *CharacterArray*, which is subclassed off *ArrayedCollection*, which is subclassed off *SequenceableCollection*, which is subclassed off *Collection*, which means that strings inherit a lot of behavior. We will cover a lot of the most useful behavior, but there is still a great deal more. To learn about the other behaviors, you'll have to browse the methods that String implements and inherits.

Creating a string

The easiest way to create a string is to enclose a sequence of characters in single quotes. To create a string with a single character, send `with:` to the class String. You can specify the size of a string and initialize it to be filled with a specified character by sending `new:withAll:`. For example,

```
string := 'here is a string'.                  'here is a string'
string := String with: $X.                     'X'
string := String new: 5 withAll: Character space.    '     '
```

Once you have a string, you can tell how big it is by sending the `size` message. For example,

```
'here is a string' size.                       16
```

Adding to a string

When you add something to a string, you are creating a new string which contains both the old string and the added characters. The most common way of adding to a string is to concatenate another string by sending the `,` (comma) message.

```
'now ', 'is ', 'the'                           'now is the'
```

Unfortunately, the , message doesn't allow you to concatenate a single character. To append a character to a string you can send `copyWith:` or you can create a string from the character and concatenate this; the former technique is more than twice as fast as the latter. For example,

```
'Why' copyWith: $?.                                    'Why?'
'Why' , (String with: $?).                             'Why?'
```

I dislike the semantics of both approaches and prefer to write and use non-standard methods that are more intuitive. In the examples below, the performance is about the same as for the techniques shown above. The examples are shown first, and the method definitions afterwards.

```
'Why' + $?.                              'Why?'
'Why' , $? asString.                                   'Why?'

SequenceableCollection>> + anObject
   ^self copyWith: anObject

Character>>asString
   ^String with: self
```

Depending on how many strings you are concatenating, it may be more efficient to create a Stream and write the strings to the stream. See Chapter 13, Streams, for more information on how to use Streams with Strings.

Comparing strings

To compare two strings for equality, you can use the = message. This checks that the two strings are exactly the same — ie, it is case sensitive. To compare two strings for equality when you *don't* care about differences in case, use `sameAs:`.

```
'abcde' = 'ABCDE'.                                     false
'abcde' sameAs: 'ABCDE'.                               true
```

Two compare two strings for the various less and greater than combinations, you can use the standard <, <=, >, and >= messages. These four messages all ignore case. (These four messages use the private message `compare:`. Another private message, `trueCompare:`, uses a primitive and is about eight times faster than `compare:`. Unfortunately, they return different results so you can't just substitute one for the other.)

```
'abcde' < 'ABCDF'.                                     true
'ABCDE' < 'abcdf'.                                     true
```

The `sameCharacters:` message will tell you how many characters are the same before the strings start differing. For example,.

```
'abcdef' sameCharacters: 'abcxyz'.                     3
```

Changing case and converting to a number

To change the case of a String, you can send it the messages `asLowercase` and `asUppercase`. (You can also send these messages to instances of Character.)

```
'HELLO' asLowercase.                    'hello'
'there' asUppercase.                    'THERE'
```

You can convert numbers in string format (for example, '54' or '3.14') to number objects by sending the `asNumber` message. It will figure out what type of number the string represents and create the correct type of number. A string that does not represent a valid number will be converted to 0. So, for example,

```
'54' asNumber.                          54
'3.14' asNumber.                        3.14
'abcd' asNumber.                        0
'a12c' asNumber.                        0
```

Pattern matching

To see if a string matches a pattern, send the message `match:` to the pattern, passing the string as a parameter. (I find this counter-intuitive: I expect to send the message to the string passing the pattern as the parameter. However, a good mnemonic is that to do Pattern Matching, use `pattern match: someString`.) Wildcard characters are `*`, which means any sequence of characters, and `#`, which means any single character. `match:` does case-insensitive matching. For example:

```
'Hello *' match: 'Hello Alec'.          true
'Hello #ave' match: 'HELLO Dave'.       true
'Hello #ave' match: 'Hello Alec'.       false
```

You can specify whether to be case-sensitive or case-insensitive by sending `match:ignoreCase:` instead, passing a boolean parameter (`match:` simply invokes `match:ignoreCase:` passing *true* as the parameter). For example,

```
'HELLO' match: 'hello' ignoreCase: true.    true
'HELLO' match: 'hello' ignoreCase: false.   false
```

Looking for characters or words in a string

If you have a string, you can tell whether it is empty by sending the `isEmpty` message. If you want to know if a character appears somewhere in a string, send `includes:`. An alternative message, `contains:`, allows you to be a bit more sophisticated and execute a block of code for each character until it finds a character for which the block evaluates to *true*. (Note that the `testing` protocol of *Character* contains many useful messages.) You can find the number of times a specified character appears in the string by sending `occurrencesOf:`.

```
'Now is the time for all' isEmpty.          false
'Now is the time for all' includes: $f      true
'Now is the time for all' contains: [
    :each | each asUppercase == $F].        true
'Now is the time for all' occurrencesOf: $t 2
```

To find *where* things are in a string, there are various options. To find the position of the first occurrence of a character, send `indexOf:`. For the position of the last occurrence, send `lastIndexOf:`. Both of these methods return 0 if the character is not found. Both also have an alternative message in which you can specify a block of code to be executed if the character is not found. Alternatively, send `findFirst:` or `findLast:` to

find the first or last character that satisfies a condition specified in a block of code. Both methods return 0 if no character satisfies the condition. If you are at a known position in the string and want to find the next or previous occurrence of a character, send `nextIndexOf:from:to:` or `prevIndexOf:from:to:`.

```
'Now is the time for all' indexOf: $t          8
'Now is the time for all' lastIndexOf: $t      12
'Now is the time for all' findFirst: [
    :char | char isLowercase].                 2
'Now is the time for all' findLast: [
    :char | char isSeparator].                 20
```

To find the position of a substring, you can send `findString:startingAt:`. Again, this returns 0 if the substring is not found, and a similar message will execute a block of code if the substring is not found. If you want to do a case-insensitive search or want to use wild-cards, you can send `findString:startingAt:ignoreCase:useWildcards:` (this message returns an interval rather than an integer).

```
'Now is the time for all' findString: 'time'
    startingAt: 4.                             12
'Now is the time for all'
   findString: 'T#ME'
   startingAt: 1
   ignoreCase: true
   useWildcards: true                          (12 to: 15)
```

Replacing characters

There are two standard ways of changing the characters in a string: one creates a new string, while the other modifies the string in place.

Creating a new string

Since we are creating a new string, we have the flexibility to insert additional characters or to remove characters from the string, as well as changing characters. The standard method is `changeFrom:to:with:`, which takes a start position, a stop position, and a replacement string as parameters, and returns a new string. For example,

```
'now is the time' changeFrom: 12
    to: 15 with: 'tune'.                       'now is the tune'
```

If *stop* is greater than or equal to *start*, both stop and start must be within the string bounds otherwise an exception is raised. If the replacement string contains the same number of characters as specified by the range from start to stop, we are doing a straight replacement. If the replacement string size is less than the range, we are replacing and deleting characters. If the replacement string size is greater than the range, we replace and insert. If we are strictly inserting characters, we insert before the start position, and the stop position should be exactly one less than the start position — ie, stop := (start - 1). So, to insert before the string, start is 1 and stop is 0. To append to the end of the string, start is the string size + 1 and stop is the string size. Let's look at some examples to clarify this.

```
'now is the time' changeFrom: 12
```

```
    to: 15 with: 'tune'.                    'now is the tune'
'now is the time' changeFrom: 12
    to: 11 with: 'right '.                  'now is the right time'
'now is the time' changeFrom: 1
    to: 0 with: 'right '.                   'right now is the time'
'now is the time' changeFrom: 16
    to: 15 with: ' for'.                    'now is the time for'
'now is the time' changeFrom: 8
    to: 11 with: ''.                        'now is time'
'now is the time' changeFrom: 8
    to: 10 with: 'a'.                       'now is a time'
'now is the time' changeFrom: 8
    to: 10 with: 'a great'.                 'now is a great time'
```

If you want to go through an entire string replacing all occurrences of one substring with another substring, use `copyReplaceAll:with:`. This method does all the replacements regardless of whether the substrings are the same length, returning the converted string.

```
'now is not the time'
    copyReplaceAll: 'no'
    with: 'yes'.                            'yesw is yest the time'
```

Modifying the string in place

To change characters of a string in place, we use a method from the `replace` family. Obviously, if we are modifying the string in place, we can only specify a range of characters that exists in the string. Also, the number of characters we want to replace must be the same as the length of the replacement string.

To replace a single character, use `at:put:` (remember that this method returns the parameter to `put:` rather than the receiver). To replace every occurrence of a specified character with another character, use `replaceAll:with:`. To do the same thing within a specified range of positions, use `replaceAll:with:from:to:`. To replace a substring with another string of the same length, use `replaceFrom:to:with:`, specifying the start and end positions of the substring. For example,

```
string := 'now is the time'.
string at: 1 put: $N.
string.                                     'Now is the time'

'now is the time' replaceAll: $t
    with: $T.                               'now is The Time'
'now is the time' replaceAll: $t
    with: $T from: 1 to: 10.                'now is The time'
'now is the time' replaceFrom: 12
    to: 15 with: 'tune'.                    'now is the tune'
```

Getting a substring

There are two standard ways to get a substring from a string. If you want to start at the beginning and copy up to a particular character, use `copyUpTo:`. Note that the character you specify is *not* included in the new string. For example,

```
'now is the time' copyUpTo: $m.             'now is the ti'
```

If you know the starting and ending position, you can use `copyFrom:to:`. For example,

```
'now is the time' copyFrom: 5 to: 10.    'is the'
```

One useful message that does not come standard with VisualWorks is `asArrayOfSubstrings`, which breaks a string into its component substrings and returns an array of those substrings. Here is an implementation. You'll need to also add a method to SequenceableCollection: `findFirst:startingAt:`, an easy extension to `findFirst:`. By adding `asArrayOfSubstrings` to CharacterArray rather than to String, it will also work with a Text object, creating an array of Text objects, each holding one of the space separated substrings.

```
CharacterArray>>asArrayOfSubstrings
   | first last collection |
   collection := OrderedCollection new.
   last := 0.
   [first := self findFirst: [ :char | char isSeparator not]
startingAt: last + 1.
   first ~= 0]
      whileTrue:
         [last := (self findFirst: [ :char | char isSeparator]
startingAt: first) - 1.
         last < 0 ifTrue: [last := self size].
         collection add: (self copyFrom: first to: last)].
   ^collection asArray
```

Here's an example of using `asArrayOfSubstrings`. We've gone further and sorted the array, then recombined it so that we are creating a string of sorted words. This shows another example of `inject:into:`. The result is 'all come for good is men now the time to '. (We end up with an extra space at the end of the string, which we could strip off easily.)

```
'now is the time for all good men to come' asArrayOfSubstrings
asSortedCollection
   inject: String new
   into: [ :string :word | string , word, (String with: Character
space)].
```

Expanding a format string

VisualWorks 2.5 provides a new family of messages (`expandMacros`) that are implemented by *CharacterArray* and which allow you to expand strings by doing parameter substitution. The message is sent directly to the format string, which uses a new class, *StringParameterSubstitution*, to do the work. Note that you are returned an instance of *Text*. The basic formatting capabilities are:

`<n>`	Insert a CR in the string.
`<t>`	Insert a tab in the string.
`<1p>`	Replace with the first argument's printString.
`<2s>`	Replace with the second argument, which must be a CharacterArray.
`<3?now:then>`	Replace with 'now' if the third argument evalues to *true* or 'then' if it evaluates to *false*.
`<4#now:then>`	Replace with 'now' if the fourth argument == 1, and with 'then'

otherwise. The fourth argument must be a Number.

For more information on the parameter substitution, look at the class comments for *StringParameterSubstitution*. Here's an example of using these parameter substitution capabilities. If you try this with count set to 1 then 2, you get back instances of Text with the strings 'There is 1 apple in the box' and ' There are 2 apples in the box'.

```
count := 1.
'There <1#is:are> <1p> apple<1#:s> in the box' expandMacrosWith:
count.
```

Here is another example using carriage returns and tabs. Notice that you can combine these two. The result is an instance of Text containing the string shown.

```
'Variables are:<nt>Var1=<1p><nt>Var2=<2p>'
   expandMacrosWithArguments: #(11 22).

'Variables are:
   Var1=11
   Var2=22'
```

Additional useful methods

Trimming white space from a string

VisualWorks doesn't come with a method to trim leading and trailing white space from a string. I find this a very useful method and so added it to CharacterArray. For a String, the method returns a new string with white space removed from both ends. For a Text object, it returns a Text object with white space removed from the string.

```
CharacterArray>>trimWhiteSpace
   "Remove leading and trailing white space"
   | first |
   first := self findFirst: [:ch | ch isSeparator not].
   ^first = 0
      ifTrue: [String new]
      ifFalse: [self copyFrom: first to: (self findLast: [:ch | ch
isSeparator not])]
```

Capitalizing a string

I find it useful to add a method that will capitalize a String or a Text object. We'll use this method in Chapter 29, Meta-Programming.

```
CharacterArray>>capitalized
   | newString |
   newString := self copy.
   newString size > 0 ifTrue: [newString at: 1 put: (newString at: 1)
asUppercase].
   ^newString
```