Type String

Type `String` is different from primitive types **`int`** and **`double`**. `String` is a *class* type. We explain what this means later, after we have explained what a class is.

Each value of type `String` is a sequence of characters. We write a value by enclosing the sequence of characters within double quotes `"`. You can see in the interactions pane of DrJava that these literals evaluate to themselves. The second example contains a capital letter and three occurrences of the space, or blank character, which is a character. The last example shows the *empty string*, the string containing no characters.

```
"first"    "The price is $25.00."    "5 <= 20"    ""
```

Use a new-line character `"\n"` to indicate a jump to a new line. For example, notice how evaluation of `"123\n4"` appears on two lines. The new-line character begins with the so-called escape character, the back-slash. Use the escape character to write the backslash itself, as well as the double quote character. For example, note how the String literal `"1\\2\"3"` evaluates to a String containing can contain the characters 1. \, 2, ", and 3. Look in the text, Chapter 6, to find other characters that are written using the escape character.

Catenation (String +)

The *one* operation of type `String`: catenation (or concatenation), is used to append one string to another. The result of the expression

```
"Store this" + " in s."
```

is the value

```
"Store this in s."
```

Here's a useful rule:

> If one operand of + is a `String`, the other operand is converted to a `String` and catenation is performed.

We look at some examples of the use of this rule. First, watch how the constant 1 is changed into a `String`. Second, watch how two numbers are first added, because of the parentheses, and the result is changed to a `String` so that catenation can be performed. If the parentheses are removed, the + operations are carried out left to right, so that both +'s are catenations:

```
"three" + 1       // yields  "three1"
"three" + (1+2)   // yields  "three3"
"three" + 1 + 2   // yields  "three12"
```

Here's a useful trick: use catenation with an empty string to turn the value of any expression into a `String`.

```
"" + (945 - 3/2.0)  yields  "943.5"
```

Useful functions

Type `String` provides a number of functions. We explain four of them.

length. For a `String s`, `s.length()` yields the number of characters in `s`. This rather funny notation, with the period in it, will make sense after we introduce classes.

```
"love".length()  // yields 4
```

You can obtain the length of any `String` expression, and not just of a literal. Here, the catenation is first performed to produce a `String` value, and then its length is obtained.

```
("non" + "violence").length()
```

The position or index of a character

Two functions are used to get at subparts of a `String`, To understand these, you have to know that the first character of a `String` is at position 0, the second is at position 1, ..., and so on. So, numbering of the characters begins at 0 and not at 1! The position is more frequently called the *index* of the character.

charAt. For a `String s`, `s.charAt(i)` gives the character at position `i` of `s`. Here are examples.

```
"Truth".charAt(0)    // yields 'T'
"Truth".charAt(4)    // yields 'h'
```

For `t.charAt(i)` to be legal, `i` must be a valid position number, or index. If not, execution aborts with an error message. For example, `"T".charAt(2)` causes a `StringIndexOutOfBoundsException`, giving the invalid index. So be careful. Whenever you write a function call `charAt(i)`, make sure `i` is in the appropriate range.

substring. Function substring extracts a substring of a string. For example,

```
"0123456".substring(1,4);
```

extracts the string consisting of the characters at positions 1, 2, and 3, but not 4. The first argument says where to start. The second argument gives the index of the character *after* the last one to be extracted.

If you leave off the second argument, the result contains every character from the first argument to the end of the string:"

```
"harmony".substring(2);   // yields "rmony"
```

A substring call with equal arguments yields the empty string `""`.

`"harmony".substring(2,2)` is `""`

You can even ask for an empty substring of the empty string:

`"".substring(0,0)` yields `""`.

equality. Please don't use `s1 == s2` to test whether strings `s1` and `s2` are the same, because it won't work the way you think —because String is a class type. Again, you'll have to wait until we discuss classes to understand this. Here's an example to show that `==` doesn't work:

```
"abc" == "ab" + "c"       // is false
```

The proper way to test equality of strings is to use function `equals`:

```
"abc".equals("ab" + "c")  // is true
```