Chapter

15

Printing Objects

If you want to print information about an object, you send it the message `printString` and get back a string containing some information about the object. For example, `3 printString` returns '3'. `OrderedCollection new printString` returns 'OrderedCollection ()', which shows that you have an empty collection. Why is this useful? The `printString` message is used to display information for debugging. In particular, it is sent by inspectors; whenever you open an inspector on an object, the inspector sends the `printString` message to the object and displays the result.

The default `printString` is implemented by Object. It first creates a stream, then does `self printOn: stream`. Object also implements a default `printOn:` method that writes to the stream a string containing the class name preceded by 'a' or 'an'. If you create a new object, it will by default inherit `printOn:` from Object. Try the following. Define a new class MyClass then evaluate `MyClass new printString`.

If you want to change what `printString` returns, it's a simple matter of implementing `printOn:` (in the *printing* protocol). Take a look at the `printOn:` method for Array, Association, and ValueHolder.

Here's an example of a very general `printOn:`. It makes heavy use of meta-programming, which involves writing code that manipulates the information about such things as classes and instance variables. Not for the faint at heart, but it can be a lot of fun to look around classes such as Behavior and ClassDescription. For more on meta-programming, see Chapter 29, Meta-Programming.

```
printOn: aStream
   super printOn: aStream.
   self class allInstVarNames
      do:
         [:each | | index |
         index := self class instVarIndexFor: each.
         aStream
            crtab;
            nextPutAll: each;
            nextPut: $:;
            space;
            print: (self instVarAt: index)]
```

I use this scheme in a slightly modified way. I write this method as `Object>>printAllOn:`, replacing the `super printOn: aStream` line with the code from `Object>>printOn:`. Then in my new classes, I write the `printOn:` method to simply invoke `printAllOn:`. Sometimes this doesn't give me the formatting I want so I'll write a specific `printOn:`, but often this will suffice.

```
MyClass>>printOn: aStream
   super printAllOn: aStream
```

Display strings

There is another message, `displayString`, which returns a value suitable for displaying. Where `printString` provides a representation of the object that is useful for debugging, `displayString` provides a representation that can be presented to the user. The default `displayString`, implemented by Object, just does `^self printString` but is overridden, for example, by CharacterArray (the superclass of String) so that strings do not have surrounding quotes. For example,

```
Transcript cr; show: 'Hello' printString.      'Hello'
Transcript cr; show: 'Hello' displayString.    Hello
```

The `displayString` message is used when displaying objects in a List box. A List box has a SequenceView as its view, or widget. Its model is a SelectionInList containing the collection of objects. The widget displays what the objects return when sent `displayString`, although you can change this. To have a message different from `displayString` sent to the objects, send `displayStringSelector: aSymbol` to the List box widget (the SequenceView), passing as the parameter the message selector you want sent.

Printing formatted numbers, dates, and times

Sending the `printString` message to a number returns the obvious representation of the number as a string. Sent to dates and times it returns a good representation, but in a pre-defined format. For example,

```
234 printString.                          '234'
Date today printString.                   '28 September 1995'
Time now printString.                     '9:59:04 am'
Timestamp now printString.                '09/28/1995 09:59:29.000'
```

There are times when you want more. For example, when printing a number you might want commas to separate the thousands, parentheses to show a negative number, or to zero fill to a specified length. When specifying a date or a time you might have international formatting concerns. The *Date* class has an additional method called `printFormat:` that allows some, but not enough, flexibility. In the example below, we ask to print the month number, the day number and the two-digit year number, with the slash character as separator.

```
(Date newDay: 34 year: 1996) printFormat: #(2 1 3 $/ 1 2)  '2/3/96'
```

The class *PrintConverter* lets you do some reasonably sophisticated formatting. You create a PrintConverter of the right type, specifying the format string you want, then you can ask the instance to format your numbers or dates when needed. Note that the PrintConverter returns an instance of Text so you'll need to send this the

`string` message to get a string. Let's look at some examples of formatting numbers, dates, and times. The string returned will be shown on the following lines.

```
pc := PrintConverter for: #number withFormatString: '00000'.
(pc formatStringFor: 234) string.
'00234'

pc := PrintConverter for: #date withFormatString: 'dddd, mmmm d,
yyyy'.
(pc formatStringFor: Date today) string.
'Thursday, September 28, 1995'

pc := PrintConverter for: #timestamp withFormatString: 'mmm d,
hh:mm:ss.ffff'.
(pc formatStringFor: Timestamp now) string.
'Sep 28, 09:55:48.0000'
```

The PrintConverter actually uses the classes *NumberPrintPolicy* and *TimestampPrintPolicy* to do the formatting. To understand all the formatting options, look at the class comments for these classes. You can also use these classes directly if you wish, and here are some examples of this. Again, the policy returns an instance of Text, which we convert to a string.

```
(NumberPrintPolicy print: 1234 using: '#,###') string.
      '1,234'
(NumberPrintPolicy print: –1234 using: '#,###;(#,###)') string.
      '(1,234)'
(TimestampPrintPolicy print: Date today using: 'yymmdd') string.
      '950928'
(TimestampPrintPolicy print: Time now using: 'mm:ss.ff') string.
      '01:12.00'
```

There are more ways of using these classes than I've shown here. For example, you can write the formatted data to a stream, and you can create instances of the policy classes. The PrintConverter class also gives you an easier but less powerful interface for numbers. You can specify digit positions using #, so, for example, the following gives a string with two leading spaces.

```
PrintConverter print: 234 formattedBy: '#####.###'.
      '  234.000'
```

VisualWorks 2.5

VisualWorks 2.5 provides some extra formatting capabilities. In particular, dates and times now respond to the new messages `longPrintString` and `shortPrintString`. For example,

```
Date today printString.            'December 2, 1995'
Date today longPrintString.        'December 2, 1995'
Date today shortPrintString.       '12/2/95'
```

You can also create a string from a format string and arguments using a message from the `expandMacros` family in the class *CharacterArray* . For information on the parameter substitution, look at the class comments for *StringParameterSubstitution*. Here's an example, with the output following the example message. Note that the `expandMacrosWithArguments:` message returns an instance of *Text*, which we convert to a string.

```
('Hello <2s>.<n>There are <1p> <3?apples:oranges> in the basket'
   expandMacrosWithArguments: #(4 'Alec' true)) asString.

'Hello Alec.
There are 4 apples in the basket'
```

printf-scanf

In the Smalltalk archives there is a fileIn called `printf-scanf` which gives you the capability of doing formatted printing as if you were using C's printf function. Its location in the MANCHESTER archive is usenet/st80-r4.X/printf-scanf. For more information on retrieving code from the Smalltalk archives, see Chapter 35, Public Domain Code and Information.