

How to display an object as a string: *printString* and *displayString*

Bobby Woolf

WHEN I TALK about how to use different sorts of objects, people often ask me what these objects look like. I draw a bunch of bubbles and arrows, underline things while I'm talking, and (hopefully) people nod knowingly. The bubbles are the objects I'm talking about, and the arrows are the pertinent relationships between them. But of course the diagram is not just circles and lines; everything has labels to identify them. The labels for the arrows are easy: The name of the method in the source that returns the target. But the labels for the bubbles are not so obvious. It's a label that somehow describes the object and tells you which one it is. We all know how to label objects in this way, but what is it that we're doing?

This is a Smalltalk programmer's first brush with a bigger issue: How do you display an object as a string? Turns out this is not a very simple issue. VisualWorks gives you four different ways to display an object as a string: *printString*, *displayString*, *TypeConverter*, and *PrintConverter*. Why does there need to be more than one way? Which option do you use when?

This article is in two parts. This month, I'll talk about *printString* and *displayString*. In September, I'll talk about *TypeConverter* and *PrintConverter*.

printString AND displayString

There are two messages you can send to an object to display it as a string:

- *printString*—Displays the object the way the developer wants to see it.
- *displayString*—Displays the object the way the user wants to see it.

printString is as old as Smalltalk itself. It was part of the original Smalltalk-80 standard and was probably in Smalltalk long before that. It is an essential part of how Inspector is implemented, an inspector being a development tool that can open a window to display any object. An inspector shows all of an object's slots (its named and indexed instance variables); when you select one, it shows that slot's value as a string by sending the slot's

value the message *printString*. The inspector also shows another slot, the pseudovariable *self*. When you select that slot, the inspector displays the object it's inspecting by sending it *printString*.

displayString was introduced in VisualWorks 1.0, more than 10 years after *printString*. *displayString* is an essential part of how *SequenceView* (VisualWorks' List widget) is implemented. The list widget displays its items by displaying a string for each item. The purpose of this display-string is very similar to that of the print-string, but the results are often different.

printString describes an object to a Smalltalk programmer. To a programmer, one of an object's most important properties is its class. Thus a print-string either names the object's class explicitly (a *VisualLauncher*, *OrderedCollection* (*#a #b*), etc.) or the class is implied (*#printString* is a *Symbol*, *1/2* is a *Fraction*, etc.). The user, on the other hand, couldn't care less what an object's class is. Because most users don't know OO, telling them that this is an object and what its class is would just confuse them. The user wants to know the name of the object. *displayString* describes the object to the user by printing the object's name (although what constitutes an object's "name" is open to interpretation).

STANDARD IMPLEMENTATION

The first thing to understand about *printString* is that it doesn't do much; its companion method, *printOn:*, does all of the work. This makes *printString* more efficient because it uses a stream for concatenation.¹ Here are the basic implementors in VisualWorks:

```
Object>>printString
| aStream |
aStream := WriteStream on: (String new: 16).
self printOn: aStream.
^aStream contents
```

```
Object>>printOn: aStream
| title |
title := self class name.
```

```
aStream nextPutAll: ((title at: 1) isVowel
  ifTrue: ['an '] ifFalse: ['a ']).
aStream print: self class
```

```
nextPutAll: ' ',self name]
```

displayString is not implemented as gracefully as printString. Rather than using a two-step process and a stream, displayString is a single method that returns a string. By default, that string is the object's print-string:

```
Object>>displayString
^self printString
```

Ideally, displayString should be implemented using displayOn:, but that message already has a different meaning in the VisualComponent hierarchy. However, those methods in the VisualComponent hierarchy would be better named "displayWith:," which more accurately describes what the method does. This would then free up displayOn: to be implemented to add an object's name onto a stream. Until displayString is implemented this way, subimplement displayString in your own classes.

displayString is a VisualWorks convention that the other Smalltalk dialects do not have. However, as you can see, its implementation is very simple, so you can easily add it to your VisualSmalltalk or IBM Smalltalk image if you'd like to.

YOUR IMPLEMENTORS

You should never implement printString in your own class (even though ParcPlace did in HelpPage and HelpSeeAlso). However, you will often want to enhance the string it produces; do so by subimplementing printOn:.

Your implementors of printString should always specify the object's class. Furthermore, it should tell the developer which instance of that class it is. To do this, printString (implemented in printOn:) should print out one or more of the object's identity variables. Identity variables are one of the types of instance variables I described in my previous article.² The values in an object's identity variables identify which instance it is and rarely change. They are the keys used to find that object in a dictionary or a database. By printing the identity variables, you're telling the developer which instance this is. If he wants to see its status and cache variables, he can use an inspector. If printOn: needs to print out a variable that's not a string, it should send that variable printString or displayString.

Cursor has a good example of printOn:. A Cursor has a name aspect to identify which cursor it is. Thus its printOn: method looks like this:

```
Cursor>>printOn: aStream
self name == nil
ifTrue: [...]
ifFalse: [aStream
  print: self class;
```

Basically, the cursor prints its class and its name (separated by a space). That tells the developer this is a Cursor and which one it is.

Your implementors of displayString should never specify what the object's class is, but they should specify which instance it is. displayString does this by printing one or more of the object's identity variables. Many objects don't have any identity variables. In these cases, there probably is no good way to display this object to the user. In such a case, just inherit Object>>displayString and avoid using it.

Remember that printString is how you want this object to appear in an inspector to a developer; displayString is how you want it to appear in a list widget to a user.

AN EXAMPLE

Let's say you're implementing the class Person. It has an aspect, name, which is an instance of PersonName. The classes will be subclassed from Object. This means that their print- and display-strings will be "a Person" and "a PersonName." This is of limited use in an inspector; worse, a selection-in-list for a collection of Persons will list "a Person" in every slot.

Here's how we could implement printString (via printOn:) and displayString to make them more useful:

```
Person>>printOn: aWriteStream
super printOn: aWriteStream.
aWriteStream
  nextPutAll: ' ';
  nextPutAll: self displayString
```

```
Person>>displayString
^self name displayString
```

```
PersonName>>printOn: aWriteStream
super printOn: aWriteStream.
aWriteStream
  nextPutAll: ' ';
  nextPutAll: self displayString
```

```
PersonName>>displayString
^self lastName, ', ', self firstName
```

The results for a person named "John Smith" are shown in Table 1.

Note that implementing printString to send the message displayString is somewhat unusual. However, I find it to be a simple and convenient example of reuse for many objects.

<i>Method</i>	<i>Default Output String</i>	<i>Custom Output String</i>
Person>>printString	a Person	a Person: Smith, John
Person>>displayString	a Person	Smith, John
PersonName>>printString	a PersonName	a PersonName: Smith, John
PersonName>>displayString	a PersonName	Smith, John

Table 1. The strings produced by printString and displayString.

DISPLAY AN OBJECT AS A STRING

This can have adverse consequences in ENVY since `Object>>printString` and `Object>>displayString` are defined in separate applications, `Kernel` and `WindowSystem`, respectively. Thus in ENVY, your applications that contain implementors of `printString` that use `displayString` may need to have `WindowSystem`—and thus `Kernel` as well—as prerequisites. Specifically, the implementors of `displayString` that `printString` uses must be in the prerequisites; luckily, `Object>>displayString` is usually not one of them. Setting up the prerequisites is usually not a problem for `Application Model` applications, but can be a problem for `Domain Model` applications, because they should not have `WindowSystem` as a prerequisite. If this is a problem for your code, the solution is to modify the OTI applications to move the necessary implementors of `displayString` from `WindowSystem` to `Kernel`. (Or you can ignore the problem because you probably won't use the image without a windowing system anyway!)

printString SHOULD NOT FAIL

Sometimes `printString` fails and issues an error notifier. This is *really* annoying. Often during development, you have an object that is not working correctly. As you inspect it to figure out why, you keep getting message-not-understood errors saying that `UndefinedObject` does not understand some message. This really limits the usefulness of the inspector!

One way to get around this problem is to have your implementors of `printOn:` check each variable before using it. Only print out a variable if it's not nil. However, checking for nil all of the time is tedious. Even if the variable is not nil, it may still be of the wrong type (which would explain why the object is not working correctly). But since the variable's value is the wrong type, it probably won't understand the messages `printOn:` sends to it, so `printOn:` will still fail.

Another tactic is to only send the variables messages that all objects understand. If you only send messages like `printString` to a variable, the message is guaranteed to work no matter what the variable's value is. However, if your implementor of `printOn:` contains a bug, it will fail and fixing the bug will be frustrating.

The universal way to prevent `printString` from failing is to have it trap errors and handle them. You can trap all errors by implementing `printString` like this:

```
Object>>printString
| aStream |
aStream := WriteStream on: (String new: 16).
Object errorSignal
handle:
[:ex |
aStream
reset;
nextPutAll: 'an invalid ';
print: self class.
ex return]
do: [self printOn: aStream].
```

`^aStream contents`

This way, if `printOn:` fails, the error handler will print out the name of the class and say that the instance is invalid. At this point, you can inspect the object to see why it is invalid. I think that is a lot better than getting an error notifier.

You may want to make this modification in your image. This will require modifying `ParcPlace's Object>>printString` method. You should usually avoid modifying vendor code, but in this case I think doing so is the best solution.

displayString AND asString

A common problem with using strings is that string concatenation (implemented in `VisualWorks` by `SequenceableCollection>>`) is not very polymorphic (nor should it be). If the concatenation argument is nil, a `Character`, an `Exception`, or some other nonstring-like object, `Smalltalk` will issue an error. To avoid this problem, developers routinely send an object `printString` before concatenating it. But `printString` does a lousy job of printing the object for concatenation: strings have quotes around them, symbols have pound-signs in front of them, most objects are called "an Object," etc.

To do a better job of printing an object out so that it can be concatenated onto a string, many developers use `asString`. They implement `Object>>asString` to define the standard protocol, then implement `asString` in all kinds of classes as they find objects that don't convert "correctly." I contend that this is a haphazard way to program and overloads `ParcPlace's` original `asString` protocol. `asString` is a message `VisualWorks` uses for converting a string-like object (such as a symbol, text, or filename) into a `String`. If an object is not at all string-like, it really has no clear implementation for `asString`.

Instead, I think that `displayString` is the solution developers are looking for. Both `asString` and `displayString` return strings. Neither message puts any junk in the string to specify the object's class. The main difference is that `asString` is an "as..." message. This implies that the receiver can be (and will be) converted to a `String` equivalent. `displayString` makes no such promises of equivalency; it simply says it will display the object as a string that describes the object.

Thus I recommend implementing `displayString` for any object you need to concatenate onto a string. Implementors you might need are:

- `UndefinedObject>>displayString` should return an empty string;
- `Character>>displayString` should return a one-character string;
- `CharacterArray>>displayString` should be reimplemented as "`^self asString displayString`."

I think this policy will be more consistent and easier to reuse than random implementors of `asString`.


CONCLUSIONS

Here are the main points in this article:

- `printString` displays an object the way a developer would

describe it. It specifies the object's class and specifies which instance the object is by displaying one or more of its identity variables.

- `displayString` displays an object the way a user would describe it. It does not specify the object's class because users never do. It specifies the object's name, that being one or more of its identity variables.
- In `VisualWorks`, don't subimplement `printString`; subimplement `printOn:` instead. Do subimplement `displayString`.
- Consider reimplementing `Object>>printString` with an error handler so that it cannot fail.
- Do not implement `Object>>asString` or most other implementors of `asString`. Use `displayString` instead.

In the next article, I'll talk about `TypeConverter` and `PrintConverter`. 

References

1. Woolf, B. "A Sample Pattern Language: Using Streams for Concatenation." *Smalltalk Report*, Feb. 1995.
2. Woolf, B. "A Strategy for Using Instance Variables." *Smalltalk Report*, June 1996.

Bobby Woolf is a Senior Member of Technical Staff at Knowledge Systems Corp. in Cary, NC. He mentors `Smalltalk` developers in the use of `VisualWorks`, `ENVY`, and `Design Patterns`. Comments are welcome at woolf@acm.org or at <http://www.ksscary.com>.