

Visibility Reference



Manual

[Contents](#) [Install](#) [Overview](#) [Supplied Objects](#)
[Hints and Tips](#) [Printing and Previewing](#) [Turtle Graphics](#)

Welcome to the Visibility Reference manual.

Visibility gives the impression of being a complex package but once you understand the way it works and create the correct sub-classes, then your pages should go together very rapidly. This manual tries to offer both the in-depth discussions of the manner of its working along with some easy to follow instructions on how to get going.

As always, there are two ways that you can start with the framework.

1. Read all the manuals and understand the structures and code of every facility and then start to build your major application from scratch or
2. Have a look at some of the demonstration methods provided in VisibilityDemo. There are quite a few of these and each one will provide you with a little insight as to how to go about the printing process. Once you have understood how one of these works, get stuck in and play with the program. Use the previews instead of the print options and you will save a forest of paper. Now you have some familiarity, try a real world page and go from there.

We recommend that you tackle this manual in the following order:

1. [Install](#)
2. [Overview](#)
3. [Demonstrations](#)
4. [Hints and tips](#)
5. [Supplied Objects](#)

Mind you, the very best way to go forward may be to jump straight to the Turtle Graphics Manual as that is by far and away the easiest way to get printed output from VisualAge.

If you do have trouble, don't forget that you get 30 days free e-mail help from support@totallyobjects.com and from then on it is only £25 for a whole year (and that includes all upgrade and bug fix releases as well!).

Visibility Reference



Manual



- [Contents](#)
- [Install](#)
- [Overview](#)
- [Supplied Objects](#)
- [Hints and Tips](#)
- [Printing](#)
- [Previewing](#)
- [Turtle Graphics](#)

Installation

This product has been packaged as a configuration map. To import it into your library select 'Browse Configuration Maps' from the 'Tools' menu of the 'System Transcript'. In the 'Configuration Maps Browser' select 'Import...' from the 'Names' menu and select the file **tobvisc6-0_1-0-XX.dat** You should then select the configuration map contained within this file:

- 'VisPreview' - Contains all the base classes for using the product.

To load the configuration map into your image select it in the 'Configuration Maps Browser' select 'Load' from the 'Editions' menu.

The configuration map contains the following applications:

- 'Visibility' - The main application.
- 'TobRotatedDeviceIndependentImageApp' - Support for rotated objects
- 'VisPrintPreviewPartsEditApp' - support for previewing
- 'VisPrintPreviewPartsRunApp' - support for previewing
- 'VisPrintPreviewPartsTestApp' - support for previewing
- 'VisPrintPreviewPartsWidgetsApp' - support for previewing
- 'ReportBuilder' - Visibility Reporter

Totally Objects - DirectDual Limited, 51 Waveney Road, Ipswich, Suffolk, IP1 5DF, England.
Tel: +44 1473 740101 (24 Hour Answer Service) Fax: +44 1473 743567 E-Mail: sales@totallyobjects.com

Visibility Reference



Manual

[Contents](#) [Install](#) [Overview](#) [Supplied Objects](#)
[Hints and Tips](#) [Printing](#) [Previewing](#) [Turtle Graphics](#)

Overview

Visibility comprises of IBM Smalltalk code to enable developers to print to any printer attached to the system with ease and flair. In addition, it provides the option of presenting a preview of the printed output.

Compatibility

Visibility is written, 100%, in IBM Smalltalk. This means that it is compatible both with IBM Smalltalk and IBM VisualAge Smalltalk. However, to use the preview features, the image requires the VisualAge extensions as well as IBM Smalltalk. As no code other than that supplied by IBM is used, Visibility should be compatible with all Smalltalk/VisualAge supported operating systems. As all printer attributes are derived from the standard Smalltalk facilities and all output is scaled, the printed result should be compatible with most printers whose drivers conform to the operating system requirements.

Note:

Due to the impossibility of testing Visibility under all operating systems and all possible printers, it cannot be guaranteed that it will produce acceptable output on every available combination. However, if any incompatibilities are discovered and reported to Totally Objects, we will do our best to resolve them. If you are not happy following that then we will offer a full refund, provided that we are informed within 30 days of providing the software to you.

Capabilities

Visibility provides the following capabilities:

- to build a series of pages from an array of strings and output them to the chosen printer with full pagination, headers, footers and page numbers;
- to build a series of pages made up from standard and user extendible building blocks
- to build a page comprising of specifically xy located Visibility print objects
- support for such items as drawing boxes around text
- text justification of left, right and centred
- word wrapping within defined text areas
- choice of portrait, landscape or fax output
- user defined process names - these appear in the print spooler window
- automatic page numbering
- full on-screen preview with built in scaling

In addition there is a new [Turtle Graphics mode](#) which provides for a simpler means of defining pages.

The rest is really up to your imagination.

3.1.2 Trying it out

Have a look at the Demo Page - it will introduce you to all of the demonstration code that is available.

Visibility Reference



Manual



- [Contents](#)
- [Install](#)
- [Overview](#)
- [Supplied Objects](#)
- [Hints and Tips](#)
- [Printing](#)
- [Previewing](#)
- [Turtle Graphics](#)

- [Visibility3dBoxObject](#)
- [VisibilityBoxObjectObject](#)
- [VisibilityCircleObjectsObject](#)
- [VisibilityDashedLineObject](#)
- [VisibilityDecimalTextObject](#)
- [VisibilityEllipseObject](#)
- [VisibilityGraphicObject](#)
- [VisibilityImageObject](#)
- [VisibilityLineAttributesObject](#)
- [VisibilityLineObject](#)
- [VisibilityShadowBoxObject](#)
- [VisibilityTextObject](#)

[Home](#)

Visibility3dBoxObject

This object is sub-classed from VisibilityBoxObject and provides all the facilities of that along with the ability to draw the box in 3d. The depth of the 3d effect is defined by a 'wing offset'. There is a cascade of methods that provide increasing control over the look of the printed output. Defaults are provided for variables omitted from the method call.

```
VisibilityBoxObject subclass: #Visibility3dBoxObject
instanceVariableNames: 'wingOffset '
classVariableNames: ''
poolDictionaries: ''
```

Visibility3dBoxObject public class methods

```
Visibility3dBoxObject class>>#newFor: width where: where whereEnd: whereEnd
filled: filled wOffset: wOffset
```

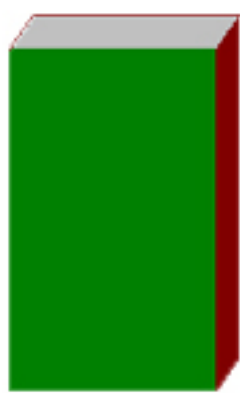
```
Visibility3dBoxObject class>>#newFor: width where: where whereEnd: whereEnd
filled: filled color: color wOffset: wOffset
```

```
Visibility3dBoxObject class>>#newFor: width where: where whereEnd: whereEnd
filled: filled color: color narrative: aString
```

```
Visibility3dBoxObject class>>#newFor: width where: where whereEnd: whereEnd
filled: filled color: color wOffset: wOffset narrative: aString
```

Parameters

width	defines the width of the border line to be drawn around the box. 0 will result in no border.
where	defines the origin of the rectangle
whereEnd	defines the location of the extent of the rectangle
filled	is a boolean where true fills the rectangle with the colour whilst false leaves the rectangle just with the border
color	defines the color of the to be used. This should be an integer of 0 - 15 and is used to select a color from the current color palette
wingOffset	the wing offset - defines the depth of the 3d effect
narrative	This is the narrative that is held with the object throughout its life. This is always attached to any debugging output and also may be seen in the debugger.



Example

Code

```
Visibility3dBoxObject
newFor: self
standardLineWidth
where: ((self
adjustX: 600) @
(self adjustY: 700))
whereEnd: ((self
adjustX: 1200) @
(self adjustY: 1700))
filled: true
color: 2
wOffset: (self
adjustX: 50)
narrative: '3d box'
```

Explanation

Uses the defined line width in Visibility see below for an explanation of adjustX: and adjustY:

says that the box should be filled with color sets the color to color 2 from the palette sets the depth of the 3d effect describes the object during debugging

self adjustX: is inherited from VisibilityBasicPrintObject. It assumes you are providing 600 x 600 dpi points and recalculates them into points at the current printer resolution. Hence on a 300 dpi printer self adjustX: 600 will actually pass 300 to the object because 600 at 600dpi = 1".

Note: the output of code from the Visibility application was created by use of [Totally Objects 'Readability'](#)

Visibility Reference



Manual



[Contents](#) [Install](#) [Overview](#) [Supplied Objects](#)

[Hints and Tips](#) [Printing](#) [Previewing](#) [Turtle Graphics](#)

Hints, tips and explanations

Which methods should you use

We have not made the usual distinction between public and private methods - in that normally the developer is suggesting that you do not use private methods as they might change. Many of our private methods are just this. However, we make some methods private as they are at a low level and should not be used to construct your pages. More than that, changing them might break something..

The misuse of a graphics context can be very difficult to track down and the whole point of this utility is that we make it easy (well easier) for you to print.

To make it very clear, those methods that are available for your use are all within the VisibilityAPI category. Anything else is held within the plain vanilla Visibility category.

Printer point issues

We use a range of names for printer points in this document. This is because a point is not always a point!

Basic dpi

All printers can manage a set number of dots across the page and down. This is regarded as the resolution and noted in dots per inch (dpi). Thus a 600 dpi printer can manage 600 dots by 600 dots in each square inch = 36,000 dots. A 300 dpi printer can only manage 300 by 300 = 9,000. Notice the acceleration in dots as we get higher. In addition, some printers - most usually ink jets - have unequal resolution - 600 by 360 or something.

We wanted to make the printing process simple in terms of getting the same page out of all printers, no matter what their resolution. Constructing a page for 300 by 300 results in a quarter sized image in the top left hand corner on a 600 by 600 printer.

Scaling factor

Visibility is constructed for a basic 600 dpi printer. Everything else is scaled from that. Thus, if you want to start printing a piece of text at a point 1000@1000, on a 600 dpi, it will send 1000@1000, on a 300 dpi printer it will send 500@500. This results in a text line at the same place on the page (well actually, there is sometimes some roundings going on as we have to work in integers). In other words, when the printing process starts, we inquire of the resolution and divide that into 600@600. Thus a 600 dpi printer returns a scaling of 1@1 whilst a 300 dpi printer results in a scaling of 2@2.

Raw points

When we refer to raw points, we are talking about the points that the printer itself understands; i.e. the unscaled points. Thus the 1000@1000 mentioned earlier will go to a 600 dpi printer at 1000@1000 (its raw points) whilst the 300 dpi printer will get 500@500 - its raw points. Some elements of the printing process are better handled in raw points as there is no reason to make them scaled only to scale them back. Line heights are a good example. You should never use a line height directly yourself. You should always let the utility handle them and thus they are kept in raw points as that is how we get them from the font sizing methods of Smalltalk.

Printer points

These are the ones that you should use, both conceptually and actually. If you always think in 600 dpi mode, you won't go wrong. Thus, the example above would result in you using 1000@1000 for both the 600 dpi and 300 dpi printers. You don't even want to think about conceptualising in 600@360 points!

Page size

Don't forget that the printable area of various printers differs. A printer may not even have the same left margin as it has right margin. Fax machines often don't even have a left margin at all! The top and bottom margins are almost certain to differ. We always return the size of the largest writeable area on the page. This does mean that the aspect ratio of boxes, etc. might differ when printed on different printers - text will be ok. We have decided that it is better to get everything on the page rather than have some lost. We would be interested in the users reactions to this and we may decide to add a facility to a future release that allows you the choice to keep the aspect ratio or the page contents consistent.

Fax machines

We use Windows NT 4.0 and have recently installed Microsoft's own fax software. On printing the test page to through that, we get all of the text left justified, irrespective of where we actually place the text, and we get no graphics. You should send a test page through your target fax to check the output before you rely on it. We are looking at this problem.

Constructing a new print object

The standard 'print an object routine' takes VisibilityBasicPrintObject (and its subclasses) objects and converts them to actual printer output.

Example - printing a box

A box can have two forms - filled or open and has a range of parameters that must be set. The creation routine for a box is a class method in VisibilityLineObject:

```
newFor:width where:where whereEnd:whereEnd
```

width: Integer. the width of the line - 0 (zero) will add no border. This is defined as the width of the line that you want to see - remember all operations look as though they are being done on a 600 dpi printer. Thus a 4 will result in a 4 point line at 600dpi but only a 2 point line at 300 dpi (but it looks like the 4 that was asked for due to its lack of resolution).

Where: Point. The top leftmost corner of the box.

WhereEnd: Point. The bottom rightmost corner of the box.

All of these are instance variables of VisibilityLineObject or inherited from VisibilityBasicObject. The actual object is created as follows:

```
newFor:width where:where whereEnd:whereEnd
```

```
|o|
```

```
o:=self new.  
o width:width;  
  where:where;  
  whereEnd:whereEnd.  
^o
```

Making your own

1. Subclass an appropriate existing object.
2. Decide on how the object must be described - an arc needs two angles, for instance, in IBM Smalltalk.
3. Add any new instance variables to the instance methods of the new class - angle1 and angle2 for example.
4. Build a newXXX method and add it as a class method. This method must accept all of the parameters required and answer a single instance of the object, similar to the above example.
5. Add an instance method to the class called drawOn:aPrinterShell using:aGraphicsContext that draws the new object. The form used is described in the IBM Smalltalk Programmers Reference but an open box looks like this.

```
drawBoxAs:aLine on:shell using:gc  
self changeWidth:self width. "always change the width as the  
requirement will have changed"  
shell window drawRectangle:gc "printerShell is the passed printer shell. gc is  
the passed printer graphics context"  
x:self where x  
y: self where y  
width:( self whereEnd x - self where x)  
height:( self whereEnd y - self where y)
```

"the shell and gc are passed from VisibilityPrint. This class has the instance variables printerShell and gc which provide these objects.

6. Construct your new page using the new object class method constructor - and bingo!

Fonts

Fonts are tricky things to handle. If you have read the IBM Smalltalk Programmer's Reference, you will see that the whole area of fonts is complex. We have tried to make it as simple as possible. Please remember, however, that Visibility is a programmer's tool and not an end user tool. We do not provide an end user printer selection window, neither do we provide a font selection window. You must allocate fonts yourself. However, we have made the use of fonts as easy as we can. VisibilityPrint contains a class dictionary called FontArray (? Sometimes our naming is not as good as it could be!) This provides simple access to the range of fonts required for the page.

At the start of the print process, VisibilityPrint builds the dictionary from a list held within initFontArray. Each line of this method builds an individual copy of a font at a specific size and weight. Thus:

```
self defineFont:'*Times New Roman-medium-r*' called:'Standard' size:10
```

builds a 'TimesNewRoman' font that is medium weight (not bold or italic) at 10 points and saves it into FontArray at 'Standard'.

Note: You must always have one font called 'Standard' for the printArray facility to work.

Another line in that method is:

```
self defineFont:'*Times New Roman-bold-r*' called:'BigBold' size:16.
```

This builds a bold version of the font at 16 point size.

defineFont: called: size does all of the actual creation by asking the operating system for a copy of a font that matches the name string. The construction of that string; i.e. why it says: '*Times New Roman-bold-r*' is described in the Programmer's Reference. Fundamentally, this returns the first font that matches the specification.

You can add to the list in the initFontArray method and provide any fonts that are available on the current computer. Please note that your customers must also have the same font installed. If they do not, then the operating system will return the nearest match, which might actually be no match at all!

You can call the fonts by any name you wish (but keep 'Standard' clear as mentioned).

You do have to build medium, bold, italic and bold italic fonts separately and name them differently - this is an operating system requirement - not ours.

Changing fonts

If you do not define a font, the printer will use its default font - normally some sort of Courier -ugh! There are two ways to manage the font situation.

Global change

If you are constructing your page in a coherent manner and are confident that a global font change is what you want, you can call the VisibilityPagePrinter instance method

```
resetFont:aName addToCommon:which
```

which will answer a font, which you can ignore. This will set the system wide font to the required one. It will appear in the instance variable font, which is inherited from VisibilityPagePrinter. You can add this to a text line at will.

The addToCommon part accepts a boolean and if true will also add a font printer object to the commonItems list. This will ensure that the printer receives the font and sets itself to it. You can then send a nil to each text line and the printer will use this global y font for every line.

Warning

Setting a font just once in a text line will reset the printer to that font and thus cause that new font to be the default.

Individual assignment

It is better, if you are building a complex page, to define the font with each line. Then you can never get confused as to which font the printer is going to use. Call

```
resetFont:aName addToCommon:which
```

with which set to false. This will load the chosen font into the class instance variable font. When you construct a line now, you can send this instance variable to the constructor and be sure that this font will be used with this text line.

This is demonstrated within [VisibilityDemo](#).

Visibility Reference



Manual



- [Contents](#)
- [Install](#)
- [Overview](#)
- [Supplied Objects](#)
- [Hints and Tips](#)
- [Printing](#)
- [Previewing](#)
- [Turtle Graphics](#)

Printing

This is what it is all about. There are two main parts of the printing process:

1. Developing the page(s)
2. Printing (or maybe [previewing](#)) them

Developing the pages

To develop the pages, you need to use the basic tools that Visibility provides. these tools are all built in a class called VisibilityPagePrinter. This is the super class of any class that you would build to create pages for your use. As you will see, VisibilityDemo is a sub-class of this. It provides all of the lower level routines that you can use to create the page. These include:

- obtaining resolution adjusted points, line heights and point locations
- setting headers, footers and page numbers
- setting fonts
- obtaining printer information
- processing the print objects

Each of the public methods are described in details later. However, the main process of creating a page needs to be described. [Click here](#) to see the source of the VisibilityDemo method that was used to develop this manual - we kept hacking it to provide the output for each object type as we went along. You can see from that the basics of the code however.

Things to note

1. The two pages came out in different orientations. This was defined by the VisibilityHoldingCollections that were used to hold each page. these are sub-classed from OrderedCollection. If you were to use a simple OrderedCollection for this, the whole document would come out in a single orientation - that set at the start of the method. However, that setting is ignored if any of the contents of printList are VisibilityHoldingCollections because they can supply individual orientation instructions. For more details, [go to Landscape/Portrait switch](#)
2. The very last line of the code is:
`#printer printDoc: printList`
if you wanted to have a window opened with the output displayed within, then replace printDoc: with previewDoc:. Please note that this is a simple window and will only display the first page of any list. If you want to preview this way.
3. If you wanted to have the output displayed in a way that you can see every page and also scale the view of the chosen displayed page down or up, then you can use

```
VisibilityPagePrinter new previewPages: printList on: #default title: 'Visibility Demo' startPage: 1 startScale: 100.
```

This will open up a much more developed window that enables you to page through the whole document, scale any page between 10% and 1000% and then print either the displayed page or the whole document. The startPage number indicates which page it should open up on and startScale tells it at which scaling to display the page.

[Click here to see the output of that operation](#)

4. The full facilities of VisibilityPagePrinter are available [here](#).

4.3 VisibilityPrint

This class does the actual interpretation of the print objects into points on the page. Most of the class is private and solely concerned with the general housekeeping of running the output and managing the graphics context. However, some elements of the support are used during the process. This is achieved by using the 'printer' instance variable assigned when VisibilityPagePrinter initialises. This can be seen in use in the last line of the [demo code](#):

```
printer printDoc: printList "printDoc loads the printer with the print array and kicks off the print process."
```

There are only three other methods of this class that you need to use.

1. previewDoc: outputs the printList to a preview window
2. debug: aBoolean if set to true will cause Visibility to output a stream of debugging information whereby the final details of every object will be put to the debug stream.
3. debugPath: aPath if set will be a fully qualified path to a directory. The debug stream will be sent to a text file in this directory and be available after the print run. If you do not set this, the debug stream will be sent to the transcript.

VisibilityPrint also manages the [FontDictionary](#). This is covered in another section of this guide.

4.4 VisibilityPrinterInfo

This class provides a wrapper for the basic page information about the printer. This information is obtained from the operating system and is thus controlled by the way that you have configured the printer using the control panel (and maybe the VA set up).

When you are sub-classed from VisibilityPagePrinter, you will have access to an instance variable called printerInfo which provides an instance of this class so all of this information is directly accessible to the page creation process. [Click here to see the list of methods you can use.](#)

[Back to Visibility Manual Home Page](#)

Visibility Reference



Manual



- [Contents](#)
- [Install](#)
- [Overview](#)
- [Supplied Objects](#)
- [Hints and Tips](#)
- [Printing](#)
- [Previewing](#)
- [Turtle Graphics](#)

Preview

Visibility has been built with the ability to preview the output on screen as well as print to a printer. There are three layers to the preview ability.

1. Use the provided method within VisibilityPrint. Normally one prints a page by calling printDoc:.
- If you use previewDoc: instead then a window will open instead of a print operation being carried out.
2. At the Smalltalk Widget level, there is VisPrintPreviewWidget. This can be used in any non-VisualAge windows.
3. At the VisualAge level there is VisPrintPreviewView.

VisPrintPreviewWidget

TBA

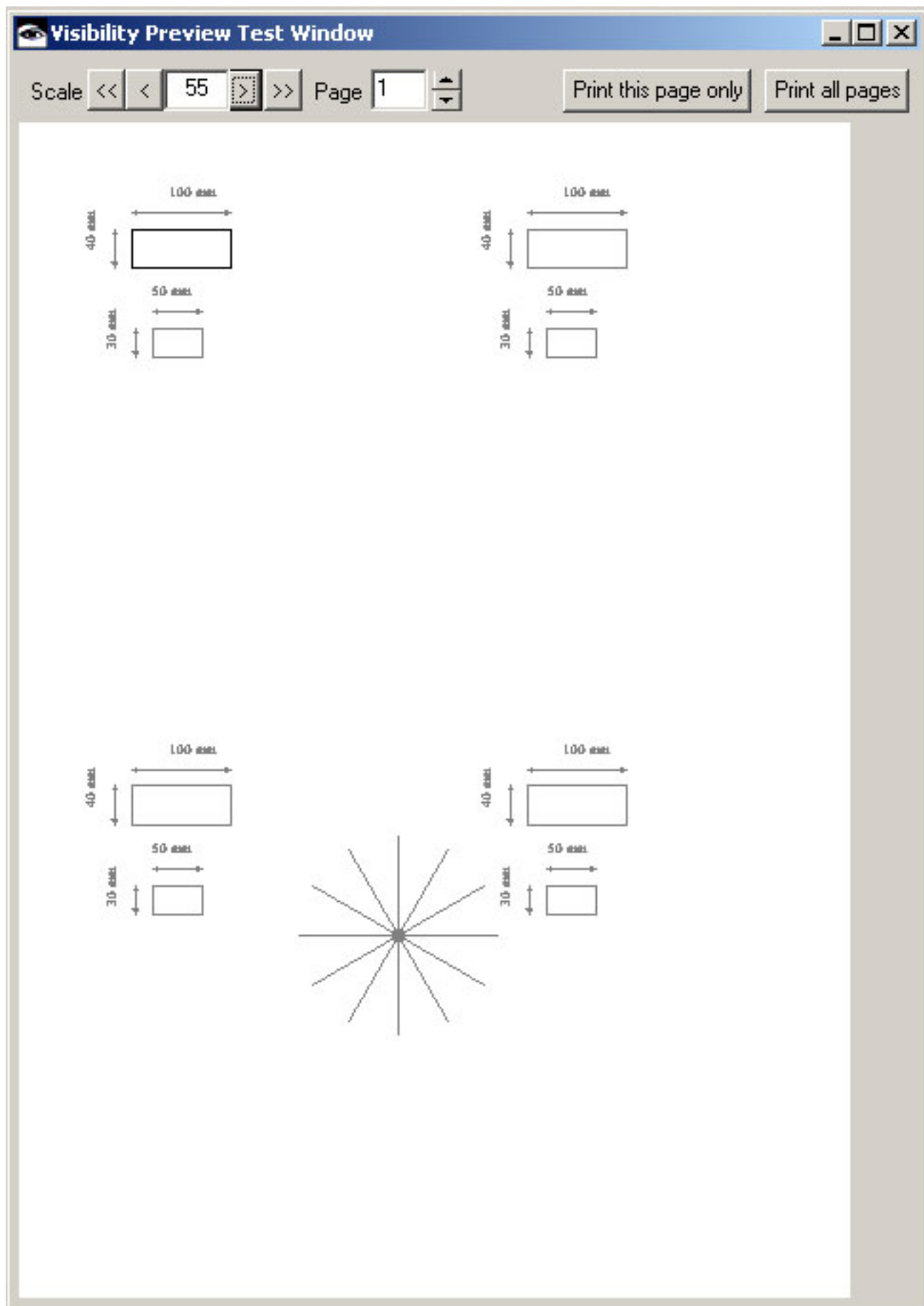
VisPrintPreviewView

This is a fully wrapped VisualAge part that can be used within the VisualAge Composition Editor. An example of this is VisPrintPreviewTestWindow. This window has been developed to show how you can use the preview part to its best ability within your own windows.

Important note:

This window is provided as an example of code and not as a fully developed end user function. Although Totally Objects would like to be informed of any problems with it, they make no warranties on its function or use. You are advised to build your own windows incorporating the VisPrintPreviewView.

Functions



- Pages:** The window can be pre loaded with Visibility print arrays or it can generate the pages on opening and from a Refresh button.
- One Page:** The window can be pre loaded with a required page number or it can default to page 1 on opening.
- Scale:** The window can be set to a pre determined scale on opening or it can default to 100%.
- Changing Scale:** The << and >> buttons change the scale down or up by a 'major' scale change factor - default is 5%. The < and > buttons change the scale down or up by a 'minor' scale change factor - default is 1%. These defaults can be changed.
- Changing Page:** Use of the Page spin button changes the page currently on display.
- Printing:** The two buttons provide for printing the whole document or just the displayed page.
- Paper background:** The physical page size will be delineated as a white rectangle.
- Orientation:** orientation will default to the printer orientation or will use individual page orientation if [VisibilityHoldingCollections](#) are used

You are referred to the code of the window for full examples of how to utilise any of these features.

Examples of starting the Preview

VisibilityPrint previewDoc:

All printing is done by sub classing VisibilityPagePrinter. VisibilityPagePrinter automatically supplies an instance of VisibilityPrint. The last line of any printing method is a call to VisibilityPrint to instigate the printing process. This is seen as the following line:

```
printer printDoc: printList.
```

If instead of calling VisibilityPrint printDoc: you call previewDoc: printList then a window will be opened containing a preview of the first page of the document (or the only page of a one page document). This is a quick and easy means of testing your printer output without wasting a forest of paper.

With a pre loaded print list.

This is a much more sophisticated means of preview.

```
| previewWindow previewList |
```

```
previewList:= VisibilityDemo makePreviewListOn: ##default. "Make the print list using the default printer"
VisibilityPagePrinter new previewPages: previewList on: ##default title: 'Test' startPage: 2 startScale: 65
"This will open the window with its contents set to previewList, the window title as 'Test', with page 2 showing set to 65%"
```

Note that there are a variety of other methods available that expose or hide a variety of the available settings.

[Back to Visibility Manual Home Page](#)

Visibility Reference



Manual



[Contents](#) [Install](#) [Overview](#) [Supplied Objects](#)
[Hints and Tips](#) [Printing](#) [Previewing](#) [Turtle Graphics](#)

Landscape/Portrait Switch with a print job

Normally, Windows programs are only able to print a document using either portrait or landscape but not both. (MS Word does manage it but we assume that the Microsoft programmers know more than others about such things). However, because of the abilities that come from being able to rotate text, Visibility can now rotate a whole page before printing. This means that, although the print job is set to portrait, some pages can be printed in landscape. To achieve this, we have subclassed `OrderedCollection` with our own `VisibilityHoldingCollection`.

When creating the collection that will hold you print objects, use a `VisibilityHoldingCollection` instead of an `OrderedCollection`. Using the provided constructor `#newForOrientation: anOrientation`, you can tell it whether the collection should be printed in `##portrait` or `##landscape`. Your final document will contain `n` collections within its overall `OrderedCollection`. If these collections are of the type `VisibilityHoldingCollection`, then the orientation held there will be used.

For backwards code compatibility, if the collection is a standard `OrderedCollection` then the preset orientation will be used.

This change ensures that all pages of a report can now be printed within a single print job, adding convenience and removal of the annoyance of finding that others have interspersed their print runs with yours.

[Back to Visibility Manual Home Page](#)

[Click here to see the output from this code](#)

demoForWebManual

| page1 page2 page1List page2List |

"init: processName: is used to initialise the print process.

orientation sets the page orientation.

processName is the name that appears in the Print spooler to denote the job that is being printed."

self

init: ##portrait

processName: 'Visibility test print'

on: ##default.

printList := OrderedCollection new."printList is the instance variable that holds the final print array"

page1List := OrderedCollection new."contains the individual objects for page 1"

page1 := VisibilityHoldingCollection new."page1 holds the first page"

page1 orientation: ##landscape." set page 1 to being in landscape mode"

page1List add: (VisibilityTextObject

newFor: 'Landscape page with Text Rotated between 20 and 360 degrees'

where: (self

atLine: 2

x: self pageCenter)

justification: ##c

font: font

color: 0

narrative: 'centered title').

20

to: 360

by: 20

do: [:i |

page1List add: (VisibilityTextObject newFor: 'Text Rotated to: ', i printString , ' degrees'

where: 1200 @ 1200

justification: ##l

font: font

color: 0

narrative: 'rotated text'

angle: i)

]." this will place a 360 degree print out of text on page 1"

page1 addAll: page1List."put the objects onto the final page"

"printList add: page1." "put page 1 onto the final output list."

page2 := VisibilityHoldingCollection new."page2 holds the second page"

page2 orientation: ##portrait."set page 2 to be a portrait page"

page2List := OrderedCollection new."contains the individual objects for page 2"

page2List add: (VisibilityTextObject

newFor: 'A couple of lines drawn on the page'

where: (self

atLine: 2

x: self pageCenter)

justification: ##c

font: font

color: 0

narrative: 'centered title').

page2List add: (VisibilityLineObject

newFor: self standardLineWidth

where: (self

atLine: 15

x: (self adjustX: 100))

whereEnd: (self

atLine: 15

x: (self adjustX: 2400))

narrative: 'base line').

page2List add: (VisibilityLineObject

newFor: self standardLineWidth

where: (self

atLine: 15

x: (self adjustX: 100))

whereEnd: (self

atLine: 20

x: (self adjustX: 2400))

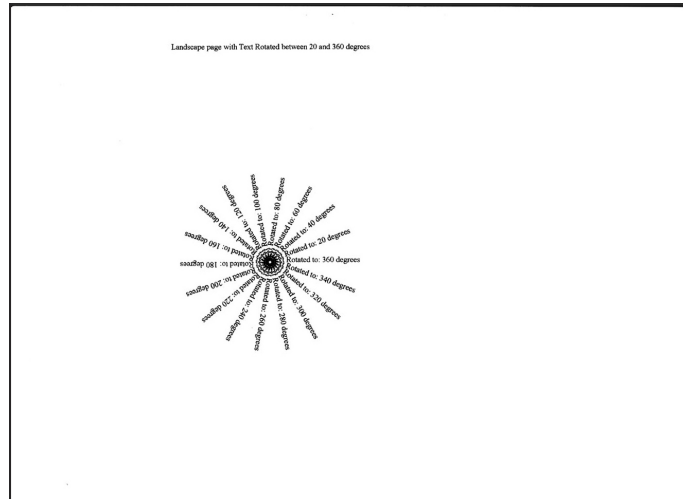
narrative: 'angled line').

page2 addAll: page2List."put the objects onto the final page"

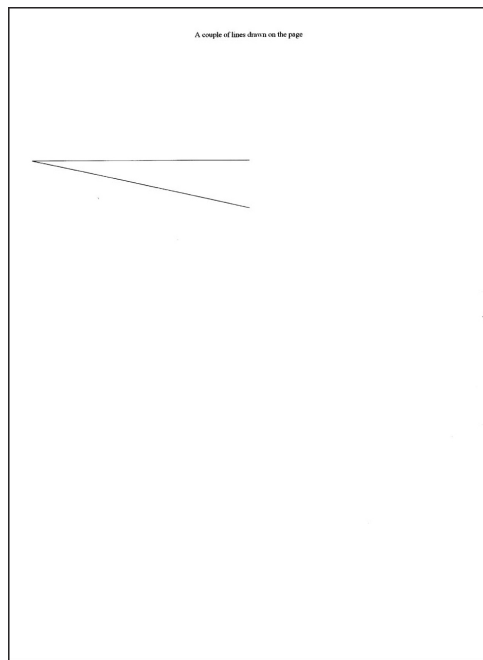
printList add: page2."put page 2 onto the final output list."

printer printDoc: printList "printDoc loads the printer with the print array and kicks off the print process."

Page 1 - output in landscape orientation



Page 2 - output in portrait orientation



You should download these pictures and see them in full size

VisibilityPrinterInfo

VisibilityPrinterInfo>>#actualRightBorder

"Return the number of points from the left hand side of the page to the right border."

VisibilityPrinterInfo>>#bottomBorder

"Return the number of points from the top of the page to the bottom border."

VisibilityPrinterInfo>>#bottomCorner

"Return a Point defining the number of printer points from the top left hand corner to the bottom right hand corner."

VisibilityPrinterInfo>>#centrePage

"Return the number of points from the left hand side of the page to the page center."

VisibilityPrinterInfo>>#fontHeight: aFontName

"Return the number of points representing the sum of the ascent and descent of a font around its baseline, i.e. its maximum height for any combination of characters ."

VisibilityPrinterInfo>>#height

"The number of points in the page height ."

VisibilityPrinterInfo>>#leftBorder

"The number of points to the first printable point across the page from the left."

VisibilityPrinterInfo>>#numLines: lineHeight

"The number of lines on the page using the current font."

VisibilityPrinterInfo>>#paperHeight

" The height of the paper in 1/10mm - A4 = 2970 for instance"

VisibilityPrinterInfo>>#paperWidth

" The width of the paper in 1/10mm - A4 = 2100 for instance"

VisibilityPrinterInfo>>#showPageNum: anObject

"A boolean. set this to true to have page numbers added to each page automatically."

VisibilityPrinterInfo>>#topBorder

"Return the value of topBorder."

VisibilityPrinterInfo>>#topCorner

VisibilityPrinterInfo>>#width

"Return the value of width ."

Visibility3dBoxObject

This object is sub-classed from VisibilityBoxObject and provides all the facilities of that along with the ability to draw the box in 3d. The depth of the 3d effect is defined by a 'wing offset'. There is a cascade of methods that provide increasing control over the look of the printed output. Defaults are provided for variables omitted from the method call.

VisibilityBoxObject subclass: #Visibility3dBoxObject
instanceVariableNames: 'wingOffset '
classVariableNames: "
poolDictionaries: "

Visibility3dBoxObject public class methods

Visibility3dBoxObject class>>#newFor: width where: where whereEnd: whereEnd filled: filled wOffset: wOffset

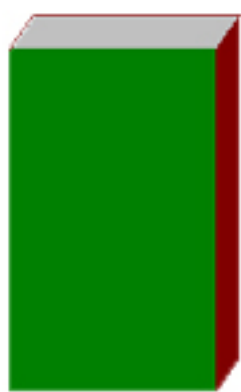
Visibility3dBoxObject class>>#newFor: width where: where whereEnd: whereEnd filled: filled color: color wOffset: wOffset

Visibility3dBoxObject class>>#newFor: width where: where whereEnd: whereEnd filled: filled color: color narrative: aString

Visibility3dBoxObject class>>#newFor: width where: where whereEnd: whereEnd filled: filled color: color wOffset: wOffset narrative: aString

Parameters

width	defines the width of the border line to be drawn around the box. 0 will result in no border.
where	defines the origin of the rectangle
whereEnd	defines the location of the extent of the rectangle
filled	is a boolean where true fills the rectangle with the colour whilst false leaves the rectangle just with the border
color	defines the color of the to be used. This should be an integer of 0 - 15 and is used to select a color from the current color palette
wingOffset	the wing offset - defines the depth of the 3d effect
narrative	This is the narrative that is held with the object throughout its life. This is always attached to any debugging output and also may be seen in the debugger.



Example

Code

```
Visibility3dBoxObject  
newFor: self  
standardLineWidth  
where: ((self  
adjustX: 600) @  
(self adjustY: 700))  
whereEnd: ((self  
adjustX: 1200) @  
(self adjustY: 1700))  
filled: true  
color: 2  
wOffset: (self  
adjustX: 50)  
narrative: '3d box'
```

Explanation

Uses the defined line width in Visibility see below for an explanation of adjustX: and adjustY:

says that the box should be filled with color sets the color to color 2 from the palette sets the depth of the 3d effect describes the object during debugging

self adjustX: is inherited from VisibilityBasicPrintObject. It assumes you are providing 600 x 600 dpi points and recalculates them into points at the current printer resolution. Hence on a 300 dpi printer self adjustX: 600 will actually pass 300 to the object because 600 at 600dpi = 1".

Note: the output of code from the Visibility application was created by use of [Totally Objects 'Readability'](#)

VisibilityBoxObject

Used to create boxes. Boxes can be either filled or open. If they are filled, they can have a border or not. there is a cascade of methods that provide increasing control over the look of the printed output. Defaults are provided for variables omitted from the method call.

Important: The instance variable 'absolute' is regarded as a private instance variable and should not be used in your code.

```
VisibilityLineObject subclass: #VisibilityBoxObject
instanceVariableNames: 'absolute '
classVariableNames: "
poolDictionaries: "
```

VisibilityBoxObject public class methods

```
VisibilityBoxObject class>>#newFor: width where: where whereEnd: whereEnd filled: filled
```

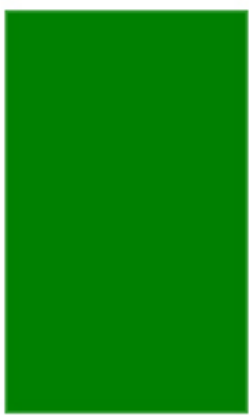
```
VisibilityBoxObject class>>#newFor: width where: where whereEnd: whereEnd filled: filled color: color
```

```
VisibilityBoxObject class>>#newFor: width where: where whereEnd: whereEnd filled: filled narrative:
aNarrative
```

```
VisibilityBoxObject class>>#newFor: width where: where whereEnd: whereEnd filled: filled color: color
narrative: aNarrative
```

Parameters

width	defines the width of the border line to be drawn around the box. 0 will result in no border.
where	defines the origin of the rectangle
whereEnd	defines the location of the extent of the rectangle
filled	is a boolean where true fills the rectangle with the colour whilst false leaves the rectangle just with the border
color	defines the color of the to be used. This should be an integer of 0 - 15 and is used to select a color from the current color palette
narrative	This is the narrative that is held with the object throughout its life. This is always attached to any debugging output and also may be seen in the debugger.



Example

Code

```
VisibilityBoxObject
newFor: self
standardLineWidth
where: ((self
adjustX: 600) @
(self adjustY:
700))
whereEnd: ((self
adjustX: 1200) @
(self adjustY:
1700))
filled: true

color: 2

narrative:
'Standard box'
```

Explanation

Uses the defined line width in Visibility see below for an explanation of adjustX: and adjustY:

says that the box should be filled with color sets the color to color 2 from the palette describes the object during debugging.

self adjustX: is inherited from VisibilityBasicPrintObject. It assumes you are providing 600 x 600 dpi points and recalculates them into points at the current printer resolution. Hence on a 300 dpi printer self adjustX: 600 will actually pass 300 to the object because 600 at 600dpi = 1".

Note: the output of code from the Visibility application was created by use of [Totally Objects](#) ['Readability'](#)

VisibilityCircleObject

Draw a circle of a defined diameter with or without a border.

VisibilityBasicPrintObject subclass: #VisibilityCircleObject
instanceVariableNames: 'diameter width '
classVariableNames: ''
poolDictionaries: ''

VisibilityCircleObject public class methods

VisibilityCircleObject class>>#newFor: width where: where diameter: aDiameter filled: filled

VisibilityCircleObject class>>#newFor: width where: where diameter: aDiameter filled: filled narrative: aString

VisibilityCircleObject class>>#newFor: width where: where diameter: aDiameter filled: filled color: aColor narrative: aString

Parameters

width	defines the width of the border line to be drawn around the box. 0 will result in no border.
where	defines the origin of the rectangle which is the bounding box of the circle
diameter	defines the diameter of the circle
filled	is a boolean where true fills the rectangle with the colour whilst false leaves the rectangle just with the border
color	defines the color of the to be used. This should be an integer of 0 - 15 and is used to select a color from the current color palette
narrative	This is the narrative that is held with the object throughout its life. This is always attached to any debugging output and also may be seen in the debugger.



Example

Code

```
VisibilityCircleObject  
newFor: self  
standardLineWidth  
where: ((self  
adjustX: 600) @  
(self adjustY: 700))  
diameter: (self  
adjustX: 1200)  
filled: true  
color: 2  
narrative: 'Standard  
circle'
```

Explanation

Uses the defined line width in Visibility see below for an explanation of adjustX: and adjustY:

says that the box should be filled with color sets the color to color 2 from the palette describes the object during debugging.

self adjustX: is inherited from VisibilityBasicPrintObject. It assumes you are providing 600 x 600 dpi points and recalculates them into points at the current printer resolution. Hence on a 300 dpi printer self adjustX: 600 will actually pass 300 to the object because $600 \text{ at } 600\text{dpi} = 1$ ".

Note: the output of code from the Visibility application was created by use of [Totally Objects 'Readability'](#)

VisibilityDashedLineObject

Sub-classed from VisibilityLineObject, this object provides a simple way to make a dashed line.

VisibilityLineObject subclass: #VisibilityDashedLineObject
instanceVariableNames: "
classVariableNames: "
poolDictionaries: "

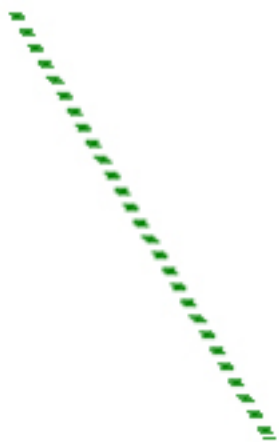
VisibilityDashedLineObject public class methods

VisibilityDashedLineObject class>>#newDashedLineFor: width where: where whereEnd: whereEnd color: color narrative: aNarrative

VisibilityDashedLineObject class>>#newDashedLineFor: width where: where whereEnd: whereEnd color: color dashStyle: dashStyle narrative: aNarrative

Parameters

width	defines the width of the border line to be drawn around the box. 0 will result in no border.
where	defines the origin of the rectangle
whereEnd	defines the location of the extent of the rectangle
filled	is a boolean where true fills the rectangle with the colour whilst false leaves the rectangle just with the border
color	defines the color of the to be used. This should be an integer of 0 - 15 and is used to select a color from the current color palette
narrative	This is the narrative that is held with the object throughout its life. This is always attached to any debugging output and also may be seen in the debugger.



Example

Code

```
VisibilityDashedLineObject  
newDashedLineFor: 15  
  
where: ((self adjustX: 600)  
@ (self adjustY: 700))  
whereEnd: ((self adjustX:  
1200) @ (self adjustY:  
1700))  
color: 2  
  
dashStyle: '4 4'  
  
narrative: 'Standard  
dashed line'
```

Explanation

Sets the line width to 15 absolute points see below for an explanation of adjustX: and adjustY:

sets the color to color 2 from the palette sets the dash style to 4 points on and 4 points off. A variety of styles are possible (see below) describes the object during debugging.

self adjustX: is inherited from VisibilityBasicPrintObject. It assumes you are providing 600 x 600 dpi points and recalculates them into points at the current printer resolution. Hence on a 300 dpi printer self adjustX: 600 will actually pass 300 to the object because 600 at 600dpi = 1".

The dashStyle attribute is defined in the IBM Smalltalk Programmer's Reference Manual as follows

dashStyle specifies the lengths of segments in the dash list	Array of integers specifying the length of each dash in number of pixels	# (4 1 1 2)	→	
		# (2 2)	→	

The array takes a form as the following examples (not exhaustive): #(4 4), #(24 8), #(12 8 4 8) or #(12 4 4 4 4).

Important note: Visibility accepts strings as well as arrays for the definition of the dashes. The above list then looks like: '4 4', '24 8', '12 8 4 8' or '12 4 4 4'

Note: the output of code from the Visibility application was created by use of [Totally Objects](#) ['Readability'](#)

VisibilityDecimalTextObject

Sub-classed from VisibilityTextObject, this provides a simple means of providing formatted number output aligned on the decimal point. In all other aspects it operates as VisibilityTextObject

```
VisibilityTextObject subclass: #VisibilityDecimalTextObject
instanceVariableNames: 'leftX rightX '
classVariableNames: ''
poolDictionaries: ''
```

VisibilityDecimalTextObject public class methods

```
VisibilityDecimalTextObject class>>#newFor: aLine where: where font: font color: color
```

12345.50

12345.50

The first line in the above is decimal aligned whilst the second line has been given the same location but is left aligned.

Please see VisibilityTextObject for the general details of this object.

Note: the output of code from the Visibility application was created by use of [Totally Objects](#) ['Readability'](#)

VisibilityEllipseObject

Provides the ability to define an ellipse object. There are four main parameters that define the ellipse. These are the height and width plus two angles which represent the amount of the ellipse to be drawn. The arc starts at the first angle and extends for the second angle, ending at the sum of the two. For a fully complete ellipse use 1 and 360 (see examples below). For further details see the IBM Smalltalk Programmer's reference under 'Drawing Arcs and Circles'.

VisibilityBasicPrintObject subclass: #VisibilityEllipseObject
instanceVariableNames: 'rMajor angle2 angle1 rMinor height width '
classVariableNames: ''
poolDictionaries: ''

VisibilityEllipseObject public class methods

VisibilityEllipseObject class>>#newFor: width rMajor: r1 rMinor: r2 where: where filled: filled

VisibilityEllipseObject class>>#newFor: width rMajor: r1 rMinor: r2 where: where filled: filled color: color

VisibilityEllipseObject class>>#newFor: width rMajor: r1 rMinor: r2 where: where filled: filled color: color narrative: aString

VisibilityEllipseObject class>>#newFor: width extent: aPoint where: where filled: filled color: color narrative: aString

VisibilityEllipseObject class>>#newFor: width extent: aPoint where: where angles: anAnglePoint filled: filled color: color narrative: aString

Parameters

width	defines the width of the border line to be drawn around the box. 0 will result in no border.
rMajor	The starting angle of the arc of the ellipse
rMinor	The angle representing the extent of the arc of the ellipse
where	defines the origin of the rectangle bounding box of the ellipse
extent	defines the location of the extent of the ellipse
angles	a point containing rMajor @ rMinor - used instead of rMinor and rMajor
filled	is a boolean where true fills the ellipse with the colour whilst false leaves the ellipse just with the border
color	defines the color of the to be used. This should be an integer of 0 - 15 and is used to select a color from the current color palette
narrative	This is the narrative that is held with the object throughout its life. This is always attached to any debugging output and also may be seen in the debugger.

Examples



rMajor = 1, rMinor = 360



rMajor = 60, rMinor = 180



rMajor = 90, rMinor = 180

Code

```
VisibilityEllipseObject  
newFor: self  
standardLineWidth  
  
extent: 1000 @ 500  
where: ((self adjustX:  
100) @ (self adjustY:  
1700))  
angles: 1 @ 360  
filled: true  
color: 2  
narrative: 'Show a  
green ellipse'
```

Explanation

Uses the defined line width in Visibility see below for an explanation of adjustX: and adjustY:

says that the box should be filled with color sets the color to color 2 from the palette describes the object during debugging.

self adjustX: is inherited from VisibilityBasicPrintObject. It assumes you are providing 600 x 600 dpi points and recalculates them into points at the current printer resolution. Hence on a 300 dpi printer self adjustX: 600 will actually pass 300 to the object because 600 at 600dpi = 1".

Note: the output of code from the Visibility application was created by use of [Totally Objects](#) 'Readability'

VisibilityGraphicObject

This object gives the ability to place an image on the page. The format of the image can be in any of the currently supported VisualAge formats; i.e. JPEG, PCX, BMP, TIFF, GIF. Please note that Visibility will test the file for its format. It does NOT rely on the suffix of the file name.

Every image is given a bounding box. This defines the area within which the image will be presented. this bounding box can also be drawn with the color and width defined. The image can be placed into the bounding box in a variety of ways:

- At its normal proportions. This will take the largest extent of the bounding box and use this is the size determiner. Additionally, it can be placed in the bounding box either left, right or center justified. Note that if the width of the picture is wider than the height, you will not see any difference in these three options as the width will fill the available rectangle width. If the height is greater than the width, then you will see a difference. Examples are given below.
- Stretched to fit the bounding box. This could result in distortion if the bounding box proportions do not match the image proportions

The image can be obtained from a given file path. If the file requested does not exist then an empty box with a cross in it will appear on the page.

The image can also be added from one held in current memory. If this mode is used, the image must have been pre-loaded using [VisibilityImageObject](#). If an image is to be used frequently it is better to cache the it into the running program as loading from file is a time expensive process.

```
VisibilityBasicPrintObject subclass: #VisibilityGraphicObject
instanceVariableNames: 'filePath boundingBox borderWidth image '
classVariableNames: ''
poolDictionaries: ''
```

VisibilityGraphicObject public class methods

Methods obtaining their image from a file.

```
VisibilityGraphicObject class>>#newFor: filePath boundingBox: box mode: mode
```

```
VisibilityGraphicObject class>>#newFor: filePath boundingBox: box borderWidth: width borderColor: color mode: mode
```

```
VisibilityGraphicObject class>>#newFor: filePath boundingBox: box borderWidth: width borderColor: color mode: mode narrative: aString
```

Methods using an image already cached.

```
VisibilityGraphicObject class>>#newForImage: imageRec boundingBox: box borderWidth: width borderColor: color mode: mode
```

```
VisibilityGraphicObject class>>#newForImage: imageRec boundingBox: box borderWidth: width borderColor: color mode: mode narrative: aString
```

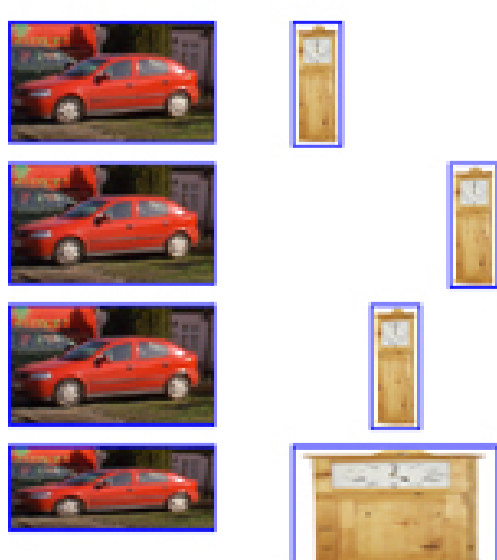
Parameters

filePath	a fully qualified path that includes the file name and suffix; e.g. 'C:\mypictures\apicture.jpg'
boundingBox	a Rectangle that defines the area on the page to be used for the image.
borderWidth	defines width in printer points of the border - 0 = no border
borderColor	defines the color to be used for the border. This should be an integer of 0 - 15 and is used to select a color from the current color palette
mode	defines how the image is placed within the bounding box. The options are: <ul style="list-style-type: none"> • #left - placed at its normal proportions to fit within the bounding box, aligned to the left hand side of the box • #center - placed at its normal proportions to fit within the bounding box, aligned to the center of the box • #right - placed at its normal proportions to fit within the bounding box, aligned to the right hand side of the box • #stretch - placed at its proportions adjusted to fit within the bounding box
narrative	This is the narrative that is held with the object throughout its life. This is always attached to any debugging output and also may be seen in the debugger.



Example

Code	Explanation
VisibilityGraphicObject newFor: 'car.jpg' boundingBox: (Rectangle origin: 100 @ 100 extent: 500@ 200)	Path of image file defines the bounding box
borderWidth: 5	defines the width of the border
borderColor: 12	sets the color to color 2 from the palette
mode: ##left	sets the justification mode
narrative: 'Left justified picture'	describes the object during debugging



Example showing the various justification options with pictures where the long side differs. The top three pictures show justification happening with images where the width is smaller than the height. The bottom row shows the different effects of ##stretch

Note: the output of code from the Visibility application was created by use of [Totally Objects](#) ['Readability'](#)

VisibilityImageObject

This class enables you to load images directly in from a file and hold them cached by assigning them to variables. The format of the image can be in any of the currently supported VisualAge formats; i.e. JPEG, PCX, BMP, TIFF, GIF. Please note that Visibility will test the file for its format. It does NOT rely on the suffix of the file name. Caching saves time and memory when an image is to be used more than once. In addition there are constructors that enable you to take any Device Independent Image (DIB) and allocate it to a VisibilityImageObject for use within the Visibility print system.

You can also rotate an image at object creation time. The rotation is limited to 90 degree steps. These steps also have corresponding symbols that may be used instead of integers.

Setting	Integer	Symbol
0 degrees	0 or 360	#normal
90 degrees	90 or -270	#up
180 degrees	180 or -180	#upsideDown
270 degrees	270 or -90	#down

Object subclass: #VisibilityImageObject
instanceVariableNames: 'image filePath '
classVariableNames: ''
poolDictionaries: ''

VisibilityImageObject public class methods

VisibilityImageObject class>>#newImage: **anImagePath**

VisibilityImageObject class>>#newImage: **anImagePath** rotatedBy: **degrees**

VisibilityImageObject class>>#newImageFromDIB: **aDIB**

VisibilityImageObject class>>#newImageFromDIB: **aDIB** rotatedBy: **degrees**

Examples

There are no visual examples of the activity of this class. However, the following code could be used to import a JPEG, cache it and then allocate it to a print object.

```
| image printList|
```

"The following code scrap will build two VisibilityGraphicsObjects that show the same image on the same line but 500 points apart. Only one image is held in memory."

```
printList := OrderedCollection new.  
image := VisibilityImageObject newImage: 'car.jpg'.  
printList add: (VisibilityGraphicObject newForImage: image  
    boundingBox: (Rectangle origin: 100 @ 100 extent: 500@ 200)  
    borderWidth: 5  
    borderColor: 12  
    mode: ##left  
    narrative: 'Left justified picture').  
printList add: (VisibilityGraphicObject newForImage: image  
    boundingBox: (Rectangle origin: 600 @ 100 extent: 500@ 200)  
    borderWidth: 5  
    borderColor: 12  
    mode: ##left  
    narrative: 'Left justified picture').
```

Note: the output of code from the Visibility application was created by use of [Totally Objects](#) ['Readability'](#)

VisibilityLineAttributes

This class provides the ability to define a line in detail. There are six provided class constructors but you may also define the object yourself (see below). There are some attributes - capStyle, joinStyle and dashStyle which are platform dependent in that not all possible styles are available on all platforms. We have provided extracts from the IBM Smalltalk Programmer's Reference that deal with the definitions and exceptions:

- [Attributes and their descriptions](#)
- [Platform differences](#)

You can also define simple lines (no dashes or arrowheads) using [VisibilityLineObject](#) and you can define dashed lines only using [VisibilityDashedLineObject](#).

The constructors provided below cover most eventualities. There are three for directly defining lines with arrows and dashes, three for lines with arrows only, and four more general ones that provide access to a variety of the attributes. You can, of course, write your own additional constructors as required.

```
Object subclass: #VisibilityLineAttributes
instanceVariableNames: 'lineWidth arrowHeads joinStyle narrative dashStyle whereEnd capStyle
whereLineStyle arrowStyle color '
classVariableNames: ''
poolDictionaries: 'CgConstants '
```

VisibilityLineAttributes public class methods

```
VisibilityLineAttributes class>>#newFor: lineWidth capStyle: capStyle joinStyle: joinStyle dashStyle:
dashStyle linestyle: linestyle location: aLocation color: aColor arrowHead: showArrow arrowStyle:
arrowStyle narrative: aNarrative
```

```
VisibilityLineAttributes class>>#newFor: lineWidth capStyle: capStyle joinStyle: joinStyle dashStyle:
dashStyle linestyle: linestyle location: aLocation color: aColor arrowHead: showArrow narrative:
aNarrative
```

```
VisibilityLineAttributes class>>#newFor: lineWidth capStyle: capStyle joinStyle: joinStyle dashStyle:
dashStyle location: aLocation color: aColor arrowHead: showArrow narrative: aNarrative
```

```
VisibilityLineAttributes class>>#newFor: lineWidth capStyle: capStyle joinStyle: joinStyle dashStyle:
dashStyle location: aLocation color: aColor narrative: aNarrative
```

Simple Arrow constructors

```
VisibilityLineAttributes class>>#newWithBothArrowFor: lineWidth arrowStyle: arrowStyle location:
aLocation color: aColor narrative: aNarrative
```

```
VisibilityLineAttributes class>>#newWithEndArrowFor: lineWidth arrowStyle: arrowStyle location:
aLocation color: aColor narrative: aNarrative
```

```
VisibilityLineAttributes class>>#newWithStartArrowFor: lineWidth arrowStyle: arrowStyle location:
aLocation color: aColor narrative: aNarrative
```

More complex Arrow constructors

```
VisibilityLineAttributes class>>#newWithBothArrowFor: lineWidth arrowStyle: arrowStyle capStyle:
capStyle joinStyle: joinStyle dashStyle: dashStyle location: aLocation color: aColor narrative:
aNarrative
```

```
VisibilityLineAttributes class>>#newWithEndArrowFor: lineWidth arrowStyle: arrowStyle capStyle:
capStyle joinStyle: joinStyle dashStyle: dashStyle location: aLocation color: aColor narrative:
aNarrative
```

```
VisibilityLineAttributes class>>#newWithStartArrowFor: lineWidth arrowStyle: arrowStyle capStyle:
capStyle joinStyle: joinStyle dashStyle: dashStyle location: aLocation color: aColor narrative:
aNarrative
```

Parameters

lineWidth defines the width of the line to be drawn. 0 will result in no line.

location a Rectangle where the origin defines the start point of the line and the corner defines the end point

capStyle see [Attributes and their descriptions](#)

joinStyle see [Attributes and their descriptions](#)

dashStyle see [Attributes and their descriptions](#)

linestyle see [Attributes and their descriptions](#)

color defines the color to be used. This should be an integer of 0 - 15 and is used to select a color from the current color palette

arrowHead defines the inclusion of arrow heads. The choice is from:

- #none - no arrow heads
- #both - an arrow head at both ends
- #start - an arrow head at the start of the line
- #end - an arrow head at the end of the line

arrowStyle defines the style of any arrow heads. The choice is from:

- #fancy
- #normal



Nil is OK if arrowHead is #none




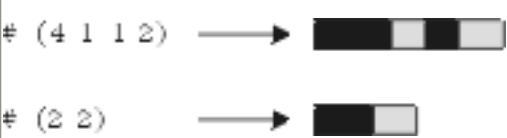
narrative This is the narrative that is held with the object throughout its life. This is always attached to any debugging output and also may be seen in the debugger.

Note: the output of code from the Visibility application was created by use of [Totally Objects 'Readability'](#)

Graphics contexts attributes

Table 1 contains the an excerpt from the list of attributes exposed by Visibility that can be initialized or modified. In most cases, the attribute is either an integer value or a constant from the *CgConstants* pool dictionary. For a detailed explanation, consult the method comment for the *createGC:values:* method of *CgDrawable*.

Table 1. Graphics context attributes

Value mask	Value expected	Default value	
GCLineStyle specifies how a line should be drawn. Dash pattern is specified using a dash list (see "dashes" below)	LineSolid (foreground color) LineOnOffDash (foreground color) LineDoubleDash (fore- and background color)	 LineSolid LineOnOffDash LineDoubleDash	LineSolid
GCCapStyle specifies how a line should be ended	CapNotLast CapButt CapRound CapProjecting		CapButt
CGJoinStyle specifies how two lines should be joined	JoinMiter JoinRound JoinBevel		JoinMiter
dashes specifies the lengths of segments in the dash list	Array of integers specifying the length of each dash in number of pixels		#(4 4)

Tip: Some platforms impose constraints on the implementation of the graphics context attributes. These limitations are described in ["Common graphics platform differences"](#).

VisibilityLineObject

This object provides the ability to add a line to the printed page. The main constructors provide for fairly simple line designs with the line width and color under direct control. In fact, Visibility lines can also be drawn with a variety of dashes and can have arrow heads added to either or both ends. There is direct support for a dashed line through a sub-class of VisibilityLineObject - [VisibilityDashedLineObject](#).

There is a specialised constructor for both dashes and arrow heads using a special definition class called [VisibilityLineAttributes](#). This is used in this class through the newForAttributes: aLineAttributes constructor described below. To use this facility, instantiate an instance of VisibilityLineAttributes using one of its class constructors and then pass that to newForAttributes:.

```
VisibilityBasicPrintObject subclass: #VisibilityLineObject
instanceVariableNames: 'whereEnd width lineAttributes '
classVariableNames: 'ArrowWeight '
poolDictionaries: 'CgConstants '
```

VisibilityLineObject public class methods

Constructors

```
VisibilityLineObject class>>#newFor: width where: where whereEnd: whereEnd
```

```
VisibilityLineObject class>>#newFor: width where: where whereEnd: whereEnd color: color
```

```
VisibilityLineObject class>>#newFor: width where: where whereEnd: whereEnd narrative: aString
```

```
VisibilityLineObject class>>#newFor: width where: where whereEnd: whereEnd color: aColor narrative:
aString
```

```
VisibilityLineObject class>>#newForAttributes: aLineAttributes
```

Class Instance Variable Accessors

```
VisibilityLineObject class>>#arrowWeight: aWeight
```

VisibilityShadowBoxObject

Sub-classed from VisibilityBoxObject, this object provides for a box with a shadow behind it. The shadow depth is defined by the shadowWidth instance variable. The color of the shadow is defaulted.

VisibilityBoxObject subclass: #VisibilityShadowBoxObject
instanceVariableNames: 'shadowWidth '
classVariableNames: "
poolDictionaries: "

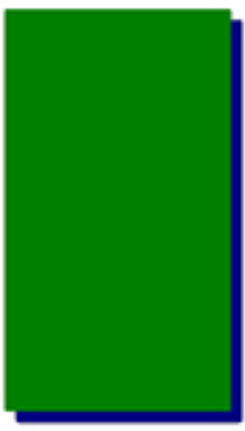
VisibilityShadowBoxObject public class methods

VisibilityShadowBoxObject class>>#newFor: width where: where whereEnd: whereEnd filled: filled color: color sWidth: sWidth

VisibilityShadowBoxObject class>>#newFor: width where: where whereEnd: whereEnd filled: filled color: color sWidth: sWidth narrative: aString

Parameters

- width defines the width of the border line to be drawn around the box. 0 will result in no border.
- where defines the origin of the rectangle
- whereEnd defines the location of the extent of the rectangle
- filled is a boolean where true fills the rectangle with the colour whilst false leaves the rectangle just with the border
- color defines the color of the to be used. This should be an integer of 0 - 15 and is used to select a color from the current color palette
- shadowWidth defines the depth of the shadow in points
- narrative This is the narrative that is held with the object throughout its life. This is always attached to any debugging output and also may be seen in the debugger.



Example

Code

```
VisibilityShadowBoxObject  
newFor: self  
standardLineWidth  
  
where: ((self adjustX: 100)  
@ (self adjustY: 700))  
whereEnd: ((self adjustX:  
500) @ (self adjustY:  
1400))  
  
filled: true  
  
color: 2  
  
sWidth: 10  
  
narrative: 'Plain green  
shadow box'
```

Explanation

Uses the defined line width in Visibility see below for an explanation of adjustX: and adjustY:

says that the box should be filled with color sets the color to color 2 from the palette sets the shadow width at 10 points

describes the object during debugging.

self adjustX: is inherited from VisibilityBasicPrintObject. It assumes you are providing 600 x 600 dpi points and recalculates them into points at the current printer resolution. Hence on a 300 dpi pirnter self adjustX: 600 will actually pass 300 to the object because 600 at 600dpi = 1".

Note: the output of code from the Visibility application was created by use of [Totally Objects](#) ['Readability'](#)

VisibilityTextObject

Places text onto the page. Text can be shown using any font defined in the VisibilityPrint fontArray.

Text can be placed justified in three main ways :

- ##l = left
- ##r = right
- ##c = center

If you wish to use justification based upon the decimal point in a number then you should use VisibilityDecimalTextObject

Text can also be rotated at any angle around its location point. You can give the object any number between 0 and 360. 0 degrees is the normal alignment. In addition there are atoms which provide for specific angles. These are:

- ##normal = 0
- ##up = 90
- ##down = 270
- ##upsideDown = 180

Text is always placed by the VAST printing system by reference to the left hand bottom-most point of the string. Consequently, text always rotates on the same point. Visibility calculates its position by reference to the justification requested. If you ask for ##c (center justification) then Visibility will calculate the leftmost point required to place the center of the string where you requested it to be. With rotation, this means that if the text is not left justified, its leftmost point will be calculated with reference to the justification and that point will then be used to rotate.

The example below of using VisibilityTextObject demonstrates the rotation of text.

Text can also be printed within an area and 'word wrapped'. This saves you having to manage line endings on the page. For more details of this see [Managing Word Wrapping!](#).

```
VisibilityBasicPrintObject subclass: #VisibilityTextObject
instanceVariableNames: 'font content justification scale fontName angle previewFont printerFont
textWidth '
classVariableNames: 'CurrentFont '
poolDictionaries: "
```

VisibilityTextObject public class methods

```
VisibilityTextObject class>>#newFor: aLine where: where justification: aJustification font: font
```

```
VisibilityTextObject class>>#newFor: aLine where: where justification: aJustification font: font angle:
anAngle
```

```
VisibilityTextObject class>>#newFor: aLine where: where justification: aJustification font: font color:
color
```

```
VisibilityTextObject class>>#newFor: aLine where: where justification: aJustification font: font narrative:
aString
```

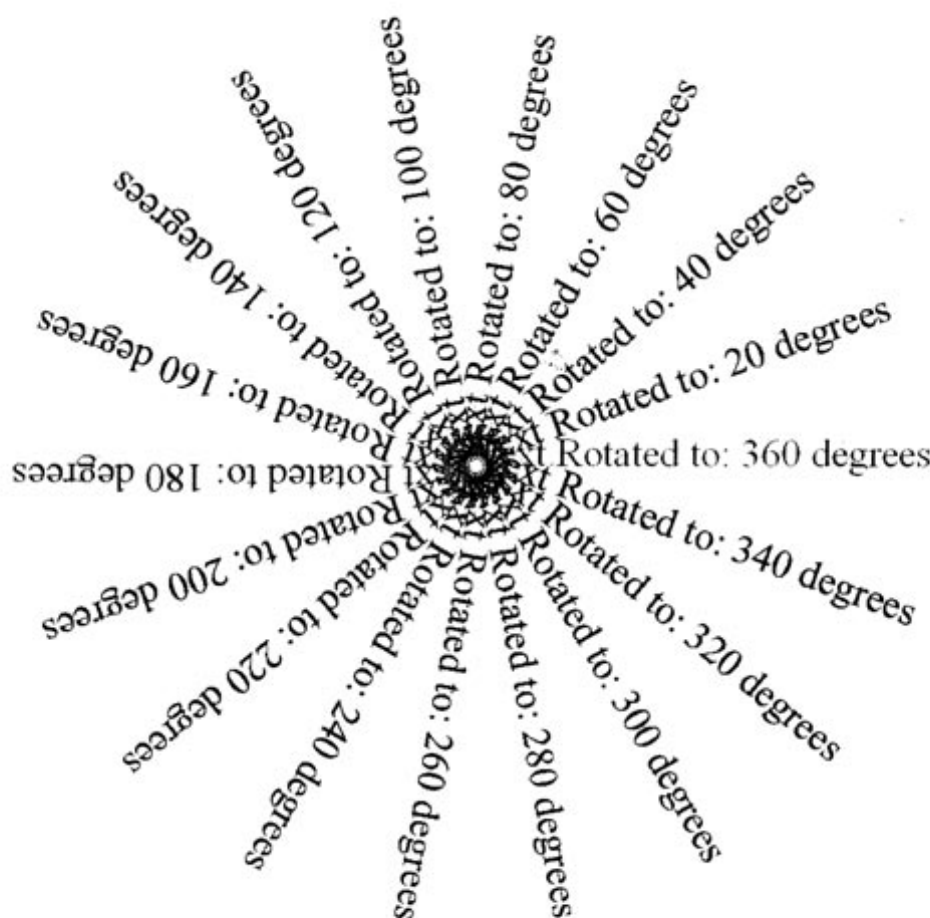
```
VisibilityTextObject class>>#newFor: aLine where: where justification: aJustification font: font color:
color angle: anAngle
```

```
VisibilityTextObject class>>#newFor: aLine where: where justification: aJustification font: font color:
color narrative: aString
```

```
VisibilityTextObject class>>#newFor: aLine where: where justification: aJustification font: font color:
color narrative: aString angle: anAngle
```

Parameters

- aLine** the text to be displayed
- where** defines the location of the text. The text alignment to this point depends upon the justification can be ##l, ##c or ##r. The text will be justified to align with the location. As an example, if you have 'Hello world' right justified, then the location given will be at the rightmost point of the final \$d.
- justification** the font to be used. This can be Nil if you are using VisibilityFont or the inline font capability of VisibilityPagePrinter to define a font. If you want an explicit font to be used, it can be obtained from VisibilityPrint using VisibilityFont fontFor: yourFontName. As an example to get the standard font, you would assign VisibilityPrint fontFor: 'Standard' to this variable.
- font** defines the color of the to be used. This should be an integer of 0 - 15 and is used to select a color from the current color palette
- color** the angle that the text is displayed at.
- angle** This is the narrative that is held with the object throughout its life. This is always attached to any debugging output and also may be seen in the debugger.
- narrative**



Example

Explanation

Code

```
20 to: 360 by: 20
do: [ : i | tempList
add:
(VisibilityTextObject
newFor:'Text
Rotated to: ', i
printString,'
degrees'
where: (self
adjustX: 1200) @
(self adjustY: 1200)
justification: ##l
font: font
color: 0
narrative: 'rotated
text'
angle: i)].
```

Content of the string to be displayed

see below for an explanation of adjustX: and adjustY:

left justified

font is a built in instance variable that is automatically loaded with the 'Standard' font and create time. (See VisibilityPagePrinter for more details)

sets the color to color 2 from the palette

describes the object during debugging.

sets the angle of the text. In this example, i is set to 20, 40, 60, etc. up to 360.

self adjustX: is inherited from VisibilityBasicPrintObject. It assumes you are providing 600 x 600 dpi points and recalculates them into points at the current printer resolution. Hence on a 300 dpi printer self adjustX: 600 will actually pass 300 to the object because 600 at 600dpi = 1".

Note: the output of code from the Visibility application was created by use of [Totally Objects](#) [Readability](#)

Word Wrapping

VisibilityTextObjects are designed to place a single line of text at a specific point on the page. Often, there is a larger amount of text than will fit on a single line. To place this text you can either spilt it into individual lines of text or place the whole text into a specified area on the page. This is managed by a process called 'word wrap'.

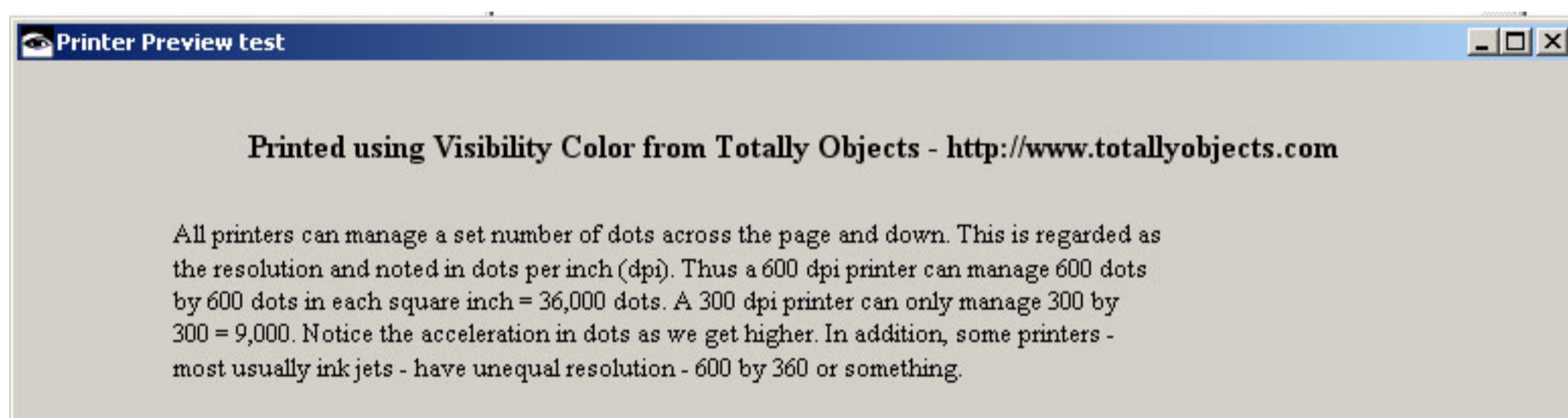
Visibility provides this facility by having a process built in that can create an array of VisibilityTextObjects that will split the text into correctly sized pieces to fit a defined width of the page. Please note that the word wrap process does not limit the number of lines delivered. It will return an array of objects for the whole text, even if this should result in text off the page.

Example of Word Wrap

The following is an excerpt from our Hints and Tips page:

"All printers can manage a set number of dots across the page and down. This is regarded as the resolution and noted in dots per inch (dpi). Thus a 600 dpi printer can manage 600 dots by 600 dots in each square inch = 36,000 dots. A 300 dpi printer can only manage 300 by 300 = 9,000. Notice the acceleration in dots as we get higher. In addition, some printers - most usually ink jets - have unequal resolution - 600 by 360 or something."

The following is a screen print of this being word wrapped on a preview screen in Visibility.



Producing wrapped text

Word wrapping is a function of the page, not of an individual text object. When creating output, you should always create you page as a sub-class of VisibilityPagePrinter. This class provides the basic framework for producing wrapped text.

VisibilityPage printer has the following instance method:

buildWrapAreaUsing: text left: left width: width startLine: startLine into: aCollection

This method takes the string 'text' that contains the required content and returns the collection provided 'aCollection' containing sufficient VisibilityTextObjects to display the whole text within the width given. The first object will be built to start at 'startLine' down the page. The above screen output was created as follows:

```
| width startLine outputArray left centre |
```

```
outputArray := OrderedCollection new.
centre := printerInfo centrePage. "Get the centre of the page"
left := centre - 1000. "Set the left hand edge of the wrap area to be 1000 points to the left of centre"
width := 1500. "Set the wrap area width to be 1500 points"
startLine := 4. "Set the start line at 4"
text := 'All printers can manage a set number of dots across the page and down. This is regarded as the resolution and noted in dots per inch (dpi). Thus a 600 dpi printer can manage 600 dots by 600 dots in each square inch = 36,000 dots. A 300 dpi printer can only manage 300 by 300 = 9,000. Notice the acceleration in dots as we get higher. In addition, some printers - most usually ink jets - have unequal resolution - 600 by 360 or something.'
```

```
outputArray := self
    buildWrapAreaUsing: text
    left: left
    width: 1500
    startLine: startLine
    into: outputArray .
```

Calculating the number of lines required

As mentioned, the wrap procedure guarantees width but produces sufficient number of lines to encompass the whole text. If you need to know how much space the wrapping will take, you can use the following method - inherited from VisibilityPagePrinter:

wrapAreaDepthUsing: aString left: left width: width

This method answers an integer which represents the number of lines required to display the whole text. This gives you the opportunity to split the text over pages, etc. if required.

Common graphics platform differences

Parts of the Common Graphics subsystem can behave differently depending on constraints of the platform (hardware, operating system, and window system). For example, Windows provides only four dashed-line styles and does not support user-defined dash styles. Where possible, Common Graphics features are mapped to the closest available features on the platform.

The table below is an extract from the full list in the IBM Smalltalk Programmers reference and identifies the platform constraints of the Common Graphics subsystem. Blank cells indicate that the corresponding item is fully supported for the indicated platform. Only those attributes that are exposed in Visibility are described here.

Table 1. Constraints on graphics context attributes (CgGC)

Item	DOS/Windows	OS/2 PM	X/MOTIF
capStyle	CapRound only		
dashes	Closest match done to 4 predefined dash lists: #(4 4), #(24 8), #(12 8 4 8), #(12 4 4 4 4 4)	Closest match done to 7 predefined dash lists: #(1 1), #(4 4), #(8 8), #(12 8 4 8), #(12 8), #(4 4 4 20), #(12 4 4 4 4 4)	
joinStyle	JoinRound only		
lineStyle	Dashes not supported by thick lines. Color not supported by LineOnOffDash.	Dashes not supported by thick lines. LineDoubleDash not supported.	
lineWidth			