

ShareScript Quick Reference (4th Edition)

This is intended to be a comprehensible, short reference to all key aspects of ShareScript – the scripting language of ShareScope based on JavaScript.

It is not comprehensive. It does not attempt to cross all the Ts or dot all the Is.

Note that most books on JavaScript focus on web pages – for ShareScript you should ignore all the HTML and web aspects and concentrate on the underlying language. *JavaScript: The Definitive Guide (5th Edition)*, from O'Reilly Press would be a good choice because it covers core JavaScript separately from JavaScript as a web language.

Programming is not trivial but it is possible – good luck!

Key terminology

This is the hardest bit. If you can sort of understand a lot of it you'll do fine. If it gets too much just move onto an example.

Variables

A name that refers to a number, some text or an object (ignore this for now). For example:

```
var a = 2;      // make the variable equal to 2 (this is a comment)
print(a);      // prints 2
a=a+3;         // make the variable 'a' equal to itself plus 3
print(a);      // prints 5
```

These variables always start with a letter and can be (almost) anything after that. A few words can't or shouldn't be used because they have a special meaning in the ShareScript e.g. 'var' or 'print'.

You should put 'var' before the variable name, the first time you use it: e.g. var a=2.

Function

A function is a series of instructions to the computer that return a value (either a number or text) to the function that called the function. A key reason behind functions is so that the code can be written once and then used over and over again.

A function can have a number of values set by the calling function set (again either number or text); these are called parameters.

Functions look like:

```
function main()
{
    print ( func1(10) );
}

function func1(val)
{
    return val * (val-1);
}
```

So you can see that the function name comes first followed by open and close brackets. If there are parameters they will appear between the open and close brackets and, if there is more than one parameter, they are separated by commas.

The rest of the function is enclosed in the left and right curly brackets. It consists of a series of instruction to the computer which we will list in the rest of this document. Here we want to focus on the text "func1 (10)" which tells the computer to start running the function "func1" using the number 10 as the input.

func1 simply takes the input value (10 in this case), multiplies by the values less one (9) and returns the result which in this case is 90.

The main function then prints "90".

Change the key line in func1 to:

```
return val * (val - 1) * (val - 2);
```

and the main function will print “720” which is $10 * 9 * 8$.

Functions can be built in to ShareScript (like ‘print’) or your own.

Errors

No one likes making a mistake but when writing programs it is inevitable. (One of my first programs had 93 errors – and it was only 3 lines long!). So don’t panic, just work out what is causing the error and fix it.

Errors are caused by the computer not knowing what to do or not correctly understanding what you want it to do. It is always the computer’s fault in the end. Computers are awfully stupid machines that are very quick and accurate at a very small range of things. You are the boss and, I personally guarantee, far more clever.

When an error is detected in your script a text window will appear called the console with the line that the error occurred. Future release will have better error reporting.

Here are some errors:

```
retrn val * (val-1); // should be ‘return’
return val * abc; // computer doesn’t know what ‘abc’ is
Return val * abc; // should be ‘return’
```

Case sensitive

This can drive beginners mad. ShareScript requires the right case (either upper or lower). So Colour.Red will be understood but colour.red and Color.Red are not. We may allow American spelling in a future release.

Arrays

Sometimes you will refer to a single share as a ‘share Object’. Sometimes you will want to refer to a number of shares e.g. FTSE 100. This is easily done through an array. You can think of an array as a list.

Zero

Let’s face it programmers are weird. The first element in an array is element 0 (zero). It does make sense (to them). Learn to enjoy the wrinkles.

Spaces, tabs and line breaks

These are referred to collectively as whitespace. They are often necessary to separate words but otherwise are ignored. They are however useful to make your code readable.

Semi colons - ;

Semi colons are used at the end of every programming phrase. You can put more than one phrase per line but for readability we recommend against.

Inverted commas – “

Put any text that ShareScript should not attempt to understand inside inverted commas. So `getShare(“LSE:MKS”) or load(“myLibrary.ss”)`.

Brackets – (and {

Brackets are very important in all computer languages. They show where something starts and where it stops. They must always be matched.

Comments

It is good practice to add comments as you write any serious piece of code.

These are marked as comments by ‘//’ (which makes the rest of that line into a comment) or by ‘/*’ to start and ‘*/’ to end.

Objects (only the more general ShareScript objects)

I’m not going to say what an object is. It’s a useful concept – the key thing is to know how to use them. See more objects (including **Date**) and their functions (also known as methods) in the last section.

Share Object

A specific share. Often you will get a Share by using `var shObj = getShare(“LSE:VOD”)`.

The other way is through a list or array. So to step through all shares in the FTSE 100:

```

var shList = getList(List.FTSE100);
for (var i=0; i<shList.length; i++) // end when we've run out of shares
{
    print( shList[i].getClose() ); // will print the closing price for the FTSE 100
}

```

You can find out about a share object by calling one of its methods. We used the `getClose()` method in the example above, which gets the latest closing price. The methods are listed in full below.

PriceData Object

Has all the price/volume data for one day. It is returned by functions such as `getPrice()`. So:

```

var prData1 = shObj.getPrice();
print ( prData1.open ); // will print out the latest open for the share 'shObj'.

```

The options are: **.open**, **.high**, **.low**, **.close**, **.volume**, **.adjustment**, **.isOHLCV**, **.date**, **.dateNum** and **.timeNum**.

.adjustment is the adjustment factor to adjust for any split or consolidations in the share. **.date** is a **Date** object – described in the last section. **.datenum** and **.timeNum** are numbers corresponding to the date and time parts of the **Date**, to allow quick and easy comparison.

Functions to get Share Objects

getShare(s)

Get the shareObj for the exchange+EPIC. So: `shObj = getShare("NNM:MSFT")` for Microsoft.

getShare(shareScopeID)

getShare(shareScopeID, shareNum)

An alternative way of getting a shareObj by specifying a company's ShareScope ID (and optionally a share number). Without a shareNum, the function will return the primary share for the company.

getPortfolioNames(groups)

Get an array of strings with the user-portfolio names. You can then pass one of the portfolio names to `getPortfolio()` to get the array of Share objects. Groups is an optional Boolean value to indicate whether you want group portfolios included.

getList(listID)

getPortfolio(s)

Get an array of all shares in one of ShareScope built in lists or a portfolio.

e.g. `ftse250 = getList(List.FTSE250);` // list of all members of the FTSE 250

e.g. `myISA = getPortfolio("MyISA");`

The **Lists** available:

.All, **.Shares**, **.InvestmentTrusts**, **.Indices**, **.FTSE100**, **.FTSE250**, **.FTSESmallCap**, **.FTSEFledgling**, **.LSENonIndex**, **.AIM**, **.Warrant**, **.Preference**, **.Convertibles**, **.Income**, **.Capital**, **.Other**, **.Imports**, **.UnitTrusts**, **.TechMark**, **.FT30**, **.TechMark100**, **.Gilts**, **.Commodities**, **.FX**, **.FTActuaries**, **.Bonds**, **.ETFS**, **.LSE**, **.NASDAQ**, **.NYSE**, **.AMEX**, **.Europe**, **.US**, **.UK**, **.LSEShares**, **.FTSE350SectorIndices**, **.FTSEAllShare**, **.CoveredWarrants**, **.LSEFullyListed**, **.IndexFutures**, **.PlusMarkets**

Share Object methods – Historical OHLCV data

shareObj.getClose()

shareObj.getClose(daysAgo)

shareObj.getCloseOnDate(date)

Get the closing price for the current share either today, a number of days ago or on a date.

shareObj.getOpen()

shareObj.getOpen(daysAgo)

shareObj.getOpenOnDate(date)

Get the opening price for the current share either today, a number of days ago or on a date.

shareObj.**getHigh**()
shareObj.getHigh(daysAgo)
shareObj.getHighOnDate(date)

Get the day's high for the current share either today, a number of days ago or on a date.

shareObj.**getLow**()
shareObj.getLow(daysAgo)
shareObj.getLowOnDate(date)

Get the day's low for the current share either today, a number of days ago or on a date.

shareObj.**getVolume**()
shareObj.getVolume(daysAgo)
shareObj.getVolumeOnDate(date)

Get the volume for the current share either today, a number of days ago or on a date.

shareObj.**getCloseArray**()
shareObj.getCloseArray(numDays)

Get an array of all closing prices for the current share or just the most recent days

shareObj.**getCloseArrayDates**()
shareObj.getCloseArrayDates(start)
shareObj.getCloseArrayDates(start, end)

Get an array of all closing prices from a date or from a date to another date

shareObj.**getPrice**()
shareObj.getPrice(daysAgo)
shareObj.getPriceOnDate(date)

Get PriceData for the current share today, a number of days ago or on a date.

shareObj.**getPriceArray**()
shareObj.getPriceArray(num)
shareObj.**getPriceArrayDates**()
shareObj.getPriceArrayDates(start)
shareObj.getPriceArrayDates(start, end)

Get an array of PriceData for the current share. See **getCloseArray** and **getCloseArrayDates**.

shareObj.**getWeeklyBarArray**()
shareObj.getWeeklyBarArray(num)
shareObj.**getMonthlyBarArray**()
shareObj.getMonthlyBarArray(num)

Get an array of OHLCV historical bars, based on calendar weeks or months. Can return bars for the entire instrument history, or just the latest num bars.

These functions return an array of **PriceData** objects.

Share Object methods – Share names

shareObj.**getEPIC**()

Get the EPIC code for the current share. e.g. "VOD". (to get the exchange use **getMarket()**)

shareObj.**getName**()

Get the name of the current share. e.g. "Vodafone PLC"

shareObj.**getShareName**()

Get the share name e.g. 'Ord 1p' or 'US\$0.50p'

shareObj.**getISIN**()
shareObj.**getSEDOL**()

Get the share's ISIN e.g. 'GB00B16GWD56' or SEDOL. Will return an empty string if the instrument does not have an ISIN or SEDOL.

Share Object methods – Other

shareObj.**getCurrency**()
shareObj.**getCurrencyR**()

Get the currency of the prices or the results (they are sometimes different). e.g. "GBP"

shareObj.**getSectorIndex**()

Get the sector for a share. e.g. getName(getSectorIndex(getShare("PSON"))) gets the sector for Pearsons which is 'Media'.

shareObj.**getMarket**()
shareObj.**getIndustry**()
shareObj.**getSector**()
shareObj.**getSubSector**()
shareObj.**getSuperSector**()
shareObj.**getListing**()
shareObj.**getTradingSystem**()

Get the exchange (e.g. "LSE"), industry, sector, sub or super-sector (e.g. "Financial"), LSE Listing (e.g. "Full" or "AIM") and Trading System (e.g. "SETSqx") for this share.

shareObj.**getResult**(year, result)

shareObj.**getResultArray**(year, result)

year is 0 (zero) for the current year, 1 for 1st year forecast, and minus for previous results.

Valid **Result**. types for both **getResult** and **getResultArray**:

.ResultType, **.Result**, **.Date**, **.Profit**, **.EPS**, **.Dividend**, **.Turnover**, **.ExDivDate**,
.DivPayDate, **.IsIFRS**

Valid **Result**. types for **getResult** only (when only annual figures are available):

.NormPreTax, **.NormEPS**, **.FRS3PreTax**, **.FRS3EPS**, **.FRS3PostTax**, **.FRS3PostTaxPS**,
.TurnoverPS, **.BookValue**, **.BookValuePS**, **.TangibleBookValue**, **.TangibleBookValuePS**,
.Cash, **.Tax**, **.CashPS**, **.CashFlow**, **.CashFlowPS**, **.Capex**, **.CapexPS**, **.RD**, **.RDPS**,
.Depreciation, **.ROCE**, **.ROE**, **.QuickRatio**, **.CurrentRatio**, **.OperatingMargin**,
.InterestCover, **.InterestPaid**, **.NetBorrowing**, **.NetCurrentAssets**, **.NetGearing**,
.NetGearingEx, **.CashPercent**, **.CashPercentEx**, **.GrossGearing**, **.GrossGearingEx**,
.GrossGearing5, **.GrossGearing5Ex**, **.GrossGearing1**, **.GrossGearing1Ex**

So for example: var profit = shObj.getResult(0, Result.Profit);

ResultType can be **ResultType**.Announced, **.Final**, **.Forecast**, **.Q1**, **.Interim**, **.Q3**, or **.Special**

getResultArray() returns an array. This is probably best regarded as a line of data from the results for a year; these result might include interim, final, announced, Q1, Q3 and special.

```
print(shObj.getResultArray(0, Result.Type));  
print(shObj.getResultArray(0, Result.Profit));
```

might result in the output:

```
Q1,Interim,Q3,Final  
499,988,1533,2040
```

shareObj.**getIndices**()

Get a list of all the indices which include this share. e.g. "FT30, FTSE 100, FTSE 350"

shareObj.**getCap**()

Get the market capitalization for the share (i.e. the price times the number of shares)

shareObj.**getNumShares**()

Get the number of shares for this share.

shareObj.**getType**()

Get the share type e.g. 'Ord', 'ETF' or 'Zero Pref'

shareObj.**isSuspended**()

Returns true if the share is suspended.

shareObj.**getShareScopeID**() / shareObj.**getShareNum**()

Return numbers representing the company (ShareScope ID) and share number. Together, these numbers uniquely identify a share.

shareObj.**getAssociatedShares()**

Get an array of all associated shares. i.e. preference or ADR shares

shareObj.**getMarketOpenTime()**

shareObj.**getMarketCloseTime()**

Gets the share's normal market open and close times. The value returned is a timeNum (seconds since midnight).

shareObj.**getMarketOffsetGMT()**

shareObj.**getMarketOffsetGMT(date)**

Get the difference in time between a share's current time-zone and GMT (in seconds) either today or on a specific date.

shareObj.**getNotes()**

Gets the values of the ShareScope notes columns as an array (length 10) of strings.

Share Object methods – Intraday data

shareObj.**getMid()**

shareObj.**getBid()**

shareObj.**getOffer()**

Get the latest intraday price for a share. Undefined is returned if not available.

shareObj.**getMidHigh()** / shareObj.**getMidLow()**

shareObj.**getTradeHigh()** / shareObj.**getTradeLow()**

Get the intraday trade or mid price high or low for a share. Undefined is returned if not available.

shareObj.**getOpen()** / shareObj.**getClose()**

shareObj.**getOpen(n)** / shareObj.**getClose(n)**

Get the latest intraday open or close price (these return undefined if price not available). You can also pass a parameter (n) to index backwards through the intraday history. 0 = latest day, 1 = the previous day, etc.

shareObj.**getDate()** / shareObj.**getDateNum()**

shareObj.**getDate(n)** / shareObj.**getDateNum(n)**

Get the intraday history date, as either a JavaScript date object, or a dateNum. You can also pass a parameter (n) to index backwards through the intraday history. 0 = latest day, 1 = the previous day, etc.

shareObj.**getBidOfferArray()**

shareObj.**getBidOfferArray(n)**

shareObj.**getBidOfferArrayOnDate(date)**

Get all the intraday prices for either the latest day, n days ago, or on a specific date. Will return undefined if no data is available on the requested date.

These functions return an array of **BidOfferData** objects. Valid BidOfferData fields are: **.bid**, **.offer**, **.mid**, **.date**, **.dateNum**, **.timeNum**.

shareObj.**getTradeArray()**

shareObj.**getTradeArray(n)**

shareObj.**getTradeArrayOnDate(date)**

Get all the intraday trades for either the latest day, n days ago, or on a specific date. Will return undefined if no data is available on the requested date.

These functions return an array of **TradeData** objects. Valid TradeData fields are: **.price**, **.volume**, **.type**, **.date**, **.dateNum**, **.timeNum**, **isPlusMarkets**.

The **type** field can be compared against constants defined on the **TradeType** class e.g. **TradeType.AT** for an automatic trade. Some of the more useful types are:

.O – ordinary, **.AT** – automatic, **.UT** – uncrossing trade.

shareObj.**getBarArray**(n, barLen)

shareObj.**getBarArrayOnDate**(date, barLen)

Get an array of OHLCV bars from the intraday data – n days ago, or on a specific date. **barLen** is the desired bar period length (in seconds). Will return undefined if no data is available on the requested date.

These functions return an array of **PriceData** objects.

Please see the *ShareScript Language Reference* for details about how the bars are calculated.

Functions – General

load(filename)

Run a ShareScript file. e.g. load("libraries/myLib.ssl"); Path relative to the ShareScript folder.

print(string)

Print the text to the text console. It automatically adds a new line. So print("Hello"); or
print("Y1 profits " + shObj.getResult(0, Result.Profit));

init(status)

This is called when ShareScript first runs either a Column, Indicator or ChartStudy script. The status parameter can be one of the following values:

Adding – When the user has just added the Column/Indicator/Study

Editing – When the user edits an existing Column/Indicator/Study

Loading – When Sharescope has just started up or is creating a ChartStudy object

Return false if you wish to prevent the Column/Indicator/Study from being added. You do not have to use the status parameter, or return a value if you don't need to.

Functions – Column

function getVal(shareObj)

When the values of a column are calculated, this function is called for each share. You never call this function – it is called by ShareScope. You create the body of this function to make a custom ShareScript column.

It returns either a number or some text which will be displayed in the column for that share.

setTitle(string)

Sets the heading for the column. You should call this from your column's init() function.

setValueForShare(shareObj, value) / **getValueForShare**(shareObj)

Sets a temporary user-defined value for on a per-instrument basis. This can be recalled later, but is not saved when the column is destroyed.

value can be any JavaScript type i.e. Boolean, number, string, array or object.

Functions – Indicator

function getGraph(shareObj, data)

When a graph is being drawn this is called to get any graph(s) that you want to be included. You never call this function – it is called by ShareScope. You create the body of this function to make a custom ShareScript indicator.

'data' is a PriceData array for the share.

If you return more than one series put them in square brackets.

To understand more see examples in the ShareScript Reference & Guide.

setTitle(string)

Sets the title of the indicator. You should call this from your indicator's init() function.

setRange(Range...)

setRange(Range..., min, max) (only if rangeMode is **Range.MinMax**)

Set the range used by the indicator. Range mode can be:

.Dynamic	Dynamically calculates the min and max according to the data e.g. MACD
.CentreZero	Makes zero in the centre (vertically) otherwise dynamic
.MinMax	You set the min and max by hand e.g. for RSI it would be -100 and 100
.Parent	Same as the main graph e.g. for a stop-loss or moving average
.ParentMerge	As .Parent but it is extended if your data would otherwise be out of range.

setSeriesChartType(series, **ChartType**....)

Sets the chart type. Series is a number corresponding to the series you are referring to. **chartType** can be one of : **.Line**, **.Histogram**, **.Filled**, **.Block** and **.Clouds**.

.Filled fills the colour to the bottom of the window. **.Block** fills in the colour to the zero line with square edges. **.Clouds** is from Ichimoku analysis – it fills in a line chart to the zero line.

setHorizontalLine(value)

Draws a horizontal line at the value e.g. to show key indicator values. The colour is the horizontal grid colour.

setSeriesColour(series, **Colour**....)

setSeriesColour(series, **Colour**...., **Colour**....)

Set the colour used by the indicator for a specified series. Two colours are used for example when the ColourMode is **.UpDown** or **.PosNeg**

Full list of **Colour**:

.Black, **.White**, **.Red**, **.Green**, **.Yellow**, **.Blue**, **.Magenta**, **.Cyan**, **.Grey**, **.Gray**, **.DarkRed**, **.DarkGreen**, **.DarkYellow**, **.DarkBlue**, **.DarkMagenta**, **.DarkCyan**, **.DarkGrey**, **.DarkGray**, **.LightRed**, **.LightGreen**, **.LightYellow**, **.LightBlue**, **.LightMagenta**, **.LightCyan**, **.LightGrey**, **.LightGray**

setSeriesColourMode(series, **ColourMode**....)

Set the colour mode used by the indicator for a specified series.

Possible values are **ColourMode.Single**, **.UpDown**, **.PosNeg**.

setSeriesLineStyle(series, **Pen**....)

setSeriesLineStyle(series, **Pen**...., width)

Set the line used by the indicator for a specified series.

Pen can be **Pen.Solid**, **.Dash**, **.Dot**, **.DashDot**, **.DashDotDot**.

The width can be between 0 (which is the narrowest) and 3. If not specified it is the narrowest. For a wide line the Pen must be **Pen.Solid**.

getBackColour()

Get the current background colour. You may want to change the colours you use for your indicators according to the background colour.

Functions – ChartStudy

ChartStudy functions called by ShareScope

function onNewChart()

Called whenever a new chart is about to be drawn. The existing set of chart bars are made available through the built-in **bars** array variable.

function onBarClose(preExisting)

Called after **onNewChart**(). It will be called once, in order, for each complete bar added to the chart. The current bar is made available through the **bar** and **barIndex** built-in variables.

The parameter (preExisting) will be true if the current bar is a pre-existing bar that was available in **onNewChart**(), and false if it is a new bar – i.e. just added to the chart.

function onNewBarUpdate(preExisting)

Called to tell you about changes to the newest (incomplete) bar at the right hand side of the chart. It is called when a new bar is created or when the OHLCV values of that bar change. The current bar is available through the **bar** and **barIndex** built-in variables.

It is followed by a call to **onBarClose**() when the bar completes.

function onMouseClick(frame, dateObj, value, altValue)

Called when the user clicks on the chart and the study has focus. The parameters describe the point clicked (see the ShareScript Language Reference for more detail).

If the user clicked on a bar, then the built-in variables **bar** and **barIndex** will refer to the clicked bar. These are undefined otherwise.

Return true if you process the click – ShareScope will then ignore it.

function onZoom()

Called when the user changes the range of visible bars.

ChartStudy functions you can call

setTitle(string)

Sets the title of the study, displayed in the study's panel.

setInfoText()

setInfoText(string)

Sets or removes the study's info text which is displayed in the study's panel below the title. Can be multi-line, use '\n' to break.

createButton(caption, callbackFunc)

Adds a button to the study's panel, returning a button handle.

deleteButton(buttonHandle)

deleteButtons()

Remove one or all the existing buttons from the study's panel.

setButtonText(buttonHandle, newCaption)

Change an existing button's caption.

getBackColour()

Get the current background colour. You may want to change the colours you use for your studies according to the background colour.

getBarLength()

Returns a string giving the chart's bar length (or period).

The first part of the string is the number of periods, the next part indicates the length of a period. e.g. "2w" is 2 weeks. The possible period lengths are s (seconds), d (days), w (weeks) and m (months).

getCurrentShare()

Returns a share object giving the currently charted instrument.

getMinVisibleBarIndex()

getMaxVisibleBarIndex()

Returns the bar indices of the left- and right-most visible bars on the chart.

setFrame(Frame....)

Set the current frame for drawing. Possible values are **Frame.Chart**, **.Volume**.

setLayer(Layer....)

Set the current layer for drawing. Possible values are **Layer.Bottom**, **.Top**.

setAltRange(min, max)

useAltRange(enable)

Set up an alternative y-axis range for the current frame. Once set up, you can draw using alternative y-axis values (across all frames) by passing true to useAltRange().

setBrushColour(Colour....)

setFontColour(Colour....)

setPenColour(Colour....)

Set the colour to be used for subsequent drawing. Pens are used for lines, and the outlines of paths, shapes, text boxes and symbols. The brush is used to fill paths, shapes, text boxes and symbols.

setPenStyle(Pen....)

setPenStyle(Pen...., width)

setPenStyle(Pen...., width, colour)

Set the pen style for drawing. Pen can be **Pen.Solid**, **.Dash**, **.Dot**, **.DashDot**, **.DashDotDot**.

The width can be between 0 (which is the narrowest) and 7. If not specified it is the narrowest. For a wide line the Pen must be Pen.Solid.

If not specified, the pen colour remains unchanged.

setFontStyle(face)

setFontStyle(face, pointSize)

setFontStyle(face, pointSize, colour)

Set the font style for drawing text. The font face is a string e.g. "Arial". If not specified, the font size and colour will be unchanged.

setFillMode(FillMode....)

Set the current fill mode for drawing. Possible values are **FillMode.Solid**, **.Transparent**.

moveTo(x, y)

Moves the drawing cursor to the specified (x,y) co-ordinate.

The x-coordinate (a date) can be specified as either (i) A JavaScript date object (slow), (ii) a bar index or (iii) an array of [dateNum, timeNum].

lineTo(x, y)

Draws a line from the cursor to the specified (x,y) co-ordinate.

Returns an integer handle to the drawn line (unless you are drawing a path).

beginPath()

endPath()

drawPath()

drawAndFillPath()

fillPath()

You can bracket a set of lineTo() commands together with begin/endPath() to form a path. The path can then be drawn and/or filled depending on the function called. This is much faster than drawing a number of individual lines.

The drawing commands return an integer handle to the drawn path.

clearDisplay()

undrawItem(handle)

Remove all objects, or a single drawn object by specifying its handle.

drawText(x, y, string)

drawText(x, y, string, **BoxAlign**....)

drawText(x, y, string, **BoxAlign**...., **TextAlign**...., isFilled, isBoxed)

Draw a text box at the specified (x,y) point. String can be multiple lines of text using '\n' to break. E.g. "Line1\nLine2".

The BoxAlign parameter allows you to position the box relative to the point. You should normally combine two values from the constants listed below using the | operator. E.g. BoxAlign.Left | BoxAlign.Above. See the ShareScript Language Reference for a diagram.

Possible values are **BoxAlign.Left**, **.Right**, **.Centre** (horizontal positioning) and **BoxAlign.Above**, **.Below**, **.VCentre** (vertical positioning).

When you draw multiple lines of text, the TextAlign constants determine the layout of the text within the box. Possible values are **TextAlign.Left**, **.Right**, **.Centre**.

IsFilled and isBoxed determine whether the box is drawn, and whether it is filled. By default these are both false.

drawTextEx(x, y, xOffset, yOffset, string, **BoxAlign**...., **TextAlign**...., isFilled, isBoxed)

As above, but with additional positioning control, using xOffset and yOffset (measured in pixels). Negative xOffset values move the box to the left, negative yOffset values move the box upwards.

drawSymbol(x, y, **Symbol**....)

drawSymbol(x, y, **Symbol**...., char, **BoxAlign**....)

drawSymbol(x, y, **Symbol**...., char, **BoxAlign**...., size, isFilled)

Draw a symbol at the specified (x,y) point. Possible symbols are **Symbol.Circle**, **.Square**, **.TriangleUp**, **.TriangleDown**, **.Cross**, **.FlagUp**, **.FlagDown**.

The symbol can contain a single character string. The **BoxAlign** parameter allows you to position the symbol relative to the point. Possible values as for **drawText()** above.

The symbol size is specified in pixels (defaults to 16). **isFilled** determines whether the symbol is filled (default is false).

drawSymbolEx(x, y, xOffset, yOffset, **Symbol**...., char, **BoxAlign**...., size, isFilled)

As above, but with additional positioning control, using **xOffset** and **yOffset** (measured in pixels). Negative **xOffset** values move the symbol to the left, negative **yOffset** values move the symbol upwards.

drawShape(x1, y1, x2, y2, **Shape**...., ratio)

drawAndFillShape(x1, y1, x2, y2, **Shape**...., ratio)

fillShape(x1, y1, x2, y2, **Shape**...., ratio)

Draws a shape from (x1,y1) to (x2,y2). The points specify the midpoint of each end of a rectangle defining the bounds of the shape. The ratio defaults to 0.5 and is the ratio of width to length. See the ShareScript Language Reference for a diagram.

Possible shapes are **Shape.Ellipse**, **.Rectangle**, **.Diamond**, **.Arrow**.

Moving Average (MA) Objects

MA(period) uses MA.Simple as type

MA(period, type)

Creates a moving average object (MAObj) that can be used to make your own MAs.

MA. types can be:

.Simple, **.Exponential**, **.Weighted**, **.Triangular**, **.VariableVHF**, **.VariableCMO**, **.Vidya**

MAObj.getNext(val)

Returns a new MA value after adding 'val' to the values used to calculate the MA.

MAObj.getValue()

Return the current MA value.

Example of Moving Averages (in an indicator)

```
function getGraph(shObj, data)
{
    var ma = new MA(10, MA.Exponential); // create a 10 day exponential MA object

    var myData = new Array();
    var myData2 = new Array();

    for (var i = 0; i < data.length; i++)
    {
        x = ma.getNext(data[i].close);
        myData[i] = x * 1.1;
        myData2[i] = x * 0.9;
    }
    return [myData,myData2]; // this is two lines +/- 10% from the 10 day EMA
}
```

Analytics Objects

Analytic objects provide access to most of ShareScope's built-in analytics. There are 2 sets of objects – those where the analytic returns a **single value**, and those where the analytic returns **multiple values** (e.g. the MACD analytic has a main value and a signal value).

Note that some analytics calculations need only a number as input (e.g. the close), and others require OHLCV (PriceData) records. This is indicated below.

Analytics – single value

These are used like MA objects. Create the desired analytic using the appropriate constructor, then use getNext(val) or getValue() to get the current analytic value.

ATR(period)

Creates an Average True Range analytic.

getNext() expects a single PriceData object (or an array of PriceData objects)

CMO(period)

Creates a Chande Momentum Oscillator analytic.

getNext() expects a number (or an array of numbers)

CCI(period)

Creates a Commodity Channel Index analytic.

getNext() expects a single PriceData object (or an array of PriceData objects)

CTI(period)

Creates a Chande Trend Index analytic.

getNext() expects a number (or an array of numbers)

Momentum(period)

Creates a Momentum analytic.

getNext() expects a number (or an array of numbers)

OnBalVol()

Creates an On Balance Volume analytic.

getNext() expects a single PriceData object (or an array of PriceData objects)

Oscillator(period)

Creates a Oscillator analytic.

getNext() expects a number (or an array of numbers)

PriceOsc(short, long)

uses MA.Simple as type

PriceOsc(short, long, MAType)

Creates a Price Oscillator analytic, with the specified short and long periods, and an optional moving average type.

getNext() expects a number (or an array of numbers)

RSI(period)

uses RSI.Wilder as type

RSI(period, RSIType)

Creates an RSI analytic, with the specified period and type.

RSI. types can be:

.Simple, .Exponential, .Wilder

getNext() expects a number (or an array of numbers)

UltimateOsc(period, period, period)

Creates a Ultimate Oscillator analytic.

getNext() expects a single PriceData object (or an array of PriceData objects)

Variance(period)

Creates a sample variance analytic.

getNext() expects a number (or an array of numbers)

Volatility(period)

Creates a Volatility analytic.

getNext() expects a number (or an array of numbers)

VHF(period)

Creates a Vertical Horizontal Filter analytic.

getNext() expects a number (or an array of numbers)

Williams(period)

Creates a Williams %R analytic.

getNext() expects a single PriceData object (or an array of PriceData objects)

WilliamsAD(period)

Creates a Williams Acc/Dist analytic.

getNext() expects a single PriceData object (or an array of PriceData objects)

Example of single value analytic use (in an indicator)

```
function getGraph(shObj, data)
{
    var rsi = new RSI(20, RSI.Simple); // create a 20 day simple RSI object
    var myData = new Array();
    for (var i = 0; i < data.length; i++)
    {
        myData[i] = rsi.getNext(data[i].close);
    }
    return myData;
}
```

Analytics – multi-value

These are used slightly differently from MA objects and single value analytics. Create the desired analytic using the appropriate constructor, and use `next(val)` to feed new value(s) into the analytic. You can then ask the analytic object for one or more of its current values.

AdaptiveStochOsc(min, max, slow, signal) uses MA.Simple as type
AdaptiveStochOsc(min, max, slow, signal, MAType)

Creates an Adaptive Stochastic Oscillator analytic, with the specified periods and an optional moving average type.

next() expects a single PriceData object (or an array of PriceData objects)

getMain() returns the analytic main value

getSignal() returns the signal line value

ADX(period)

Creates a Directional Movement (ADX) analytic, with the specified period.

next() expects a single PriceData object (or an array of PriceData objects)

getPDI() returns +ve directional indicator value

getNDI() returns -ve directional indicator value

getADX() returns the average directional index

getADXR() returns the average directional index rating

Aroon(period)

Creates an Aroon analytic, with the specified period.

next() expects a single PriceData object (or an array of PriceData objects)

getUp() returns the Aroon up value

getDown() returns the Aroon down value

MACD(short, long, signal)

Creates an MACD object with the specified periods.

next() expects a number (or an array of numbers)

getMain() returns the analytic main value

getSignal() returns the signal line value

MinMax(length)

Creates a MinMax object with the specified buffer length. This is a useful building block for custom indicators: new values are added to the buffer, and the oldest automatically removed when the buffer is full.

next() expects a number (or an array of numbers)

getMax() returns the maximum value in the buffer

getMin() returns the minimum value in the buffer

StochOsc(period, slow, signal) uses MA.Simple as type

StochOsc(period, slow, signal, MAType)

Creates a Stochastic Oscillator analytic, with the specified periods and an optional moving average type.

next() expects a single PriceData object (or an array of PriceData objects)

getMain() returns the analytic main value

getSignal() returns the signal line value

Trend(period)

Creates a Rolling Trend object with the specified period.

next() expects a number (or an array of numbers)

getSlope() returns the trend line gradient

getValue() returns the trend's latest value

getStdDev() returns the standard deviation of the trend

Example of multi-value analytic use (in an indicator)

```
function getGraph(shObj, data)
{
    var macd = new MACD(13,26,9); // create a standard MACD
    var main = new Array();
    var signal = new Array();
```

```

    for (var i = 0; i < data.length; i++)
    {
        macd.next(data[i].close);
        main[i] = macd.getMain();
        signal[i] = macd.getSignal();
    }
    return [main, signal];
}

```

Dialog boxes

You can create and display dialog boxes for interacting with the user using a Dialog object. You can show dialog boxes from programs run from the console, and from the `init()` functions of Columns and Indicator scripts.

Dialog()

Dialog(title, width, height)

Creates a dialog object (DialogObj), with optional title (string), width and height.

DialogObj.show()

Display the dialog and wait for the user to finish. This function will return one of the following values, depending on how the user exited:

Dialog.Ok (if Okayed), **Dialog.Cancel** (if cancelled or closed), **Dialog.User** (if a user defined button).

DialogObj.getValue(name)

Returns the value of the named control on the dialog. The return type depends on the control type. See the ShareScript Reference for details.

DialogObj.addOkButton()

DialogObj.addOkButton(x, y, w, h, caption)

DialogObj.addCancelButton()

DialogObj.addOkButton(x, y, w, h, caption)

Add Ok and Cancel buttons to the dialog. Use `-1` for any of x, y, w, h for automatic size / position.

DialogObj.addHelpButton(file)

DialogObj.addHelpButton(file, x, y, w, h, caption)

Add Help button to the dialog, which will display an HTML file when clicked. The file path should be relative to the ShareScript directory, and include an html extension.

DialogObj.addGroupBox(x, y, w, h)

DialogObj.addGroupBox(x, y, w, h, caption)

Add a group box with optional caption. You must pass valid x, y, w, h values – there is no automatic positioning for group boxes. However, Group boxes do modify the automatic positioning of subsequent controls.

DialogObj.addText(x, y, w, h, text)

Add text to the dialog. You must pass valid x, y, w, h values – there is no automatic positioning for text.

DialogObj.addButton(x, y, w, h, caption, id)

Add a custom button to the dialog. Use `-1` for any of x, y, w, h for automatic size / position. The id is returned by DialogObj.show() if the user exits using the button.

DialogObj.addTickBox(name, x, y, w, h)

DialogObj.addTickBox(name, x, y, w, h, text, initVal)

Add a tick/check box to the dialog. Text is displayed to the right of the control. The initial value can be true or false. Use `-1` for any of x, y, w, h for automatic size / position.

DialogObj.addIntEdit(name, x, y, w, h)

DialogObj.addIntEdit(name, x, y, w, h, leftLabel, rightLabel, initVal, min, max)

DialogObj.addNumEdit(name, x, y, w, h)

DialogObj.addNumEdit(name, x, y, w, h, leftLabel, rightLabel, initVal, min, max)

DialogObj.**addTextEdit**(name, x, y, w, h)

DialogObj.addTextEdit(name, x, y, w, h, leftLabel, rightLabel, initVal)

Add an integer, number or text edit box initialized to initVal, with min and max bounds and with left and right text labels. Use -1 for x, y, w, h to automatic size/position.

DialogObj.**addDropList**(name, x, y, w, h, items)

DialogObj.addDropList(name, x, y, w, h, items, leftLabel, rightLabel, initVal)

Add a drop-down list to the dialog. The list is populated from the items array. Use -1 for any of x, y, w, h for automatic size / position.

DialogObj.**addColPicker**(name, x, y, w, h)

DialogObj.addColPicker(name, x, y, w, h, leftLabel, rightLabel, initVal)

Add a colour picker to the dialog, initialized to initVal. Use -1 for any of x, y, w, h for automatic size / position.

DialogObj.**addColLinePicker**(name, x, y, w, h)

DialogObj.addColLinePicker(name, x, y, w, h, leftLabel, rightLabel, initCol, initStyle, initWidth)

Add a colour/line picker to the dialog, initialized to initCol, initStyle, initWidth. Use -1 for any of x, y, w, h for automatic size / position.

Example of Dialog box usage

```
var dlg = new Dialog("example", 200, 50);
dlg.addOkButton();
dlg.addCancelButton();
dlg.addIntEdit("period", -1, -1, -1, -1, "Period", "days", 7, 2, 500);
if (dlg.show() == Dialog.Ok)
{
    period = dlg.getValue("period");
}
```

File Objects

File()

create a File object

File(filename)

create a File object, and open (using File.ReadMode)

File(filename, mode)

create a File object, and open using the mode

Creates a File object (FileObj). The second two forms of the constructor create a File object and also open a specified file in a single operation.

File modes can be:

.ReadMode, .WriteMode, .AppendMode

Important notes:

You can read files from anywhere in the `ShareScript` directory. However, you can only write to the `ShareScript/Output` directory.

If the filename does not begin with a slash (e.g. "test.csv") it will be read/written relative to the `ShareScript/Output` directory.

If the filename does begin with a slash (e.g. "/Columns/test.csv") it will be read/written relative to the `ShareScript` directory.

You can use either forward (/) or back (\) slash characters to separate elements of the path. However, backslashes need to be escaped in strings ("\\").

FileObj.**open**(filename)

FileObj.**open**(filename, mode)

Open a file. If no mode is specified the file is opened using File.ReadMode.

FileObj.**close**()

Close the current file. The FileObj can be reused.

FileObj.**readLine**()

Return a complete line of text from the file as a string (with CR/LF removed).

FileObj.**writeLine**(string)

Write the string to the file, automatically adding a CR/LF.

Selected extra general purpose objects & functions

These functions are generally referred to as methods as they refer to a specific object.

Console Window

The console window allows you to write ShareScript for immediate execution. Use the up arrow to review / re-run previous lines.

The console window is also where errors are reported.

Note that you can still use ShareScope in the normal way while the console is showing though the console will always be the top screen.

clear()

Clears the **console window**.

beep()

Plays a short alert sound (the script is paused while it plays).

For **Arrays**: **.reverse** and **.sort**.

```
var ff=getList(List.FTSE250);
function mySort(A,B)
{
    if (A.getName().charAt(2) > B.getName().charAt(2))
        return 1;
    return -1;
}

ff.sort(mySort); // sort the FTSE 2500 by their third letter
```

For **Date**: **.getDate** (of month, 1-31), **.getDay** (0 is Sunday - 6), **.getMonth** (0-11), **.getFullYear** (4 digit e.g 2007), **.toDateString** (converts date into text)

So for example:

```
var d=new Date();
print (d.toDateString())
```

JavaScript date objects can be relatively slow. You can convert a date into a compact number by `dateNum(dateObj)` which can be easily compared to another `dateNum`.

Note that JavaScript date objects also store the time. You can convert the time part of a JavaScript date into a compact number by `timeNum(date)` which can be easily compared to another `timeNum`.

For **Math**: **.E** (constant e), **.PI** (constant pi), **.ceil()**, **.floor()**, **.max()**, **.min()**, **.round()**, **.log** (natural log bas e), **.exp()**, **.pow()**, **.abs()**, **.sqrt()**, **.cos()**, **.sin()**, **.tan()**, **.random** (0 to 1.0)

So for example: `Math.exp (Math.PI)` is e to power of pi. `Math.sqrt(9)` is 3.

For **Strings**: **.length**, **.charAt()**, **.concat()**, **.indexOf()**, **.lastIndexOf()**, **.replace()**, **.search()**, **.slice()**, **.split()**, **.substr()**, **.toLowerCase()**, **.toUpperCase()**

So for example:

```
var s= "abcaxy";
print(s.length + s.charAt(2) + s.indexOf("ax")); // prints "6c3"
```

undefined is used to indicate that no value is available.

So if, for example, a request for the forecast EPS is made and this value is not available, **undefined** is returned. You should test for **undefined** to make sure you are processing valid data.

Other aspects of coding ShareScript

All the program control is identical to JavaScript which is similar to C. This is super brief explanation/reminder. Ignore this section entirely if it is at all stressful.

if ... else if ... else ...

for loops, break and goto

The for loop has three parts: the initial setup, the test to end the loop and the action at the end of every loop. For example:

```
var shList = getList(List.FTSE100);
for (i=0 ; shList[i] ; i++) // end when we've run out of shares i.e. when shList[i] is not true
{
// will print the closing price for the FTSE 100 if there was a change of greater than 2%
var pToday = shList[i].getClose();
var pYest = shList[i].getClose(1);
if (((pToday - pYest) / pYest) > .02)
    print( shList[i].getName() + " : " + shList[i].getClose() );
else
    print(shList[i].getName() + " Not up 2%")
}
```

Comparisons, Logic etc

Comparison are >, >=, <, <=, ==, !=

Bit-wise operators: &, |, ^ (xor), ~ (compliment)

Logical operators: &&, || and ! (compliment)

++, --, += etc are supported along with >>>, <<< and the conditional operator ? :

Others

Switch/case, do/while and while loops.

```
switch (n)
{
    case 0:
        ... // your code here
        break;
    case 1:
        ...
        break;
    default:
        ...
}

do
{
    ...
    i++; // increment i
} while (shList[i]) // exits when this ceases to be true

while (shList[i]) // test at start i.e. the code in the loop may never run
{
    ...
    i++;
}
```

throw, try, catch & finally are available for error handling.

Variables and arrays are initialized to **undefined**. Arrays are only filled with **undefined** to their size.

Testing

Remember that data that is available for one company may not be available for another. For robustness, please check you ShareScript code on all ShareScope's data (i.e. the all list from top to bottom with the space key).

For more details please refer to either the ShareScript Reference & Guide or "JavaScript, The Definitive Guide" by David Flanagan from O'Reilly Press 5th Edition.