



WealthScript Language Guide

Wealth-Lab Developer 4.0

© 2003-2006 WL Systems, Inc.

Wealth-Lab Developer 4.0 WealthScript Language Guide

by WL Systems, Inc.

Revised: Monday, December 11, 2006

Wealth-Lab Developer 4.0 WealthScript Language Guide

© 2003-2006 WL Systems, Inc.

No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

Third party trademarks and service marks are the property of their respective owners.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use or misuse of information contained in this document or from the use or misuse of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Printed: Monday, December 11, 2006

Special thanks to:

Wealth-Lab's great on-line community whose comments have helped make this manual more useful for veteran and new users alike.

EC Software, whose product HELP & MANUAL printed this document.

Table of Contents

Part I Introduction	5
Part II WealthScript Language Syntax	7
1 Overview	7
2 Comments	8
3 Statements and Delimiters	8
4 Variables and Data Types	9
Overview	9
Declaring Variables	10
Variable Naming Rules	10
Data Types	11
Record Types	12
Enumerated Types	13
5 Assignment Statements	14
6 Constants	16
7 Operations	18
Overview	18
Mathematical Operations	18
Boolean Operations	19
Logical Operations	21
Summary	21
And Operator	21
Or Operator	22
Xor Operator	24
Not Operator	25
String Operations	25
8 Conditional Statements	26
9 Case Statement	30
10 Looping Statements	32
Summary	32
For Loop	32
While Loop	33
Repeat Loop	34
Breaking Out of a Loop	34
11 Functions and Procedures	35
Overview	35
Declaring Procedures	36
Declaring Functions	37
Calling Functions and Procedures	39
Passing Parameters	40
Scope of Variables	42
Exiting a Procedure	43
Native and Re-usable Functions	44
12 Error Handling	44
13 Arrays	45
Part III Working with Price Series	48
1 Introduction to Price Series	48

2 What is a Price Series?	48
3 Handles to Price Series	49
Overview	49
Standard Price Series and Their Constants	50
Functions that Return a Price Series Handle	51
Functions that Accept a Price Series Handle	52
4 Creating Your Own Price Series	54
5 Accessing a Single Value of a Price Series	55
6 Using @ Syntax to Access Values from a Price Series	57
7 Series Math	58
Practice	58
Answers	59
8 Price Series FAQs	61
Part IV Painting the Chart	64
1 Overview	64
2 Chart Panes	65
3 Creating New Panes	65
4 Plotting an Indicator in a Pane	67
5 Plotting Multiple Symbols	68
6 Specifying Colors	69
7 Drawing Text in a Pane	70
8 Drawing Objects in a Pane	70
Part V Writing Your Trading System Rules	72
1 Overview	72
2 Scripting Trading Rules	72
Overview	72
The Main Loop	73
Triggering a Market Buy Order	74
Triggering a Limit or Stop Buy Order	75
Checking for Open Positions	75
Using Automated Stops	76
Selling Short	77
3 Implementing Trading System Rules	78
4 Managing Multiple Positions	79
Part VI Working with Technical Indicator Functions	83
1 Overview	83
2 Accessing Indicator Values	83
3 Accessing Indicator Price Series Handles	84
Part VII Accessing Data from Files	86
1 Overview	86
2 Creating and Opening Files	86
3 Reading and Writing	87
4 Closing Files	88

Part VIII Understanding Time Frames	89
1 Overview	89
2 Accessing a Higher Time Frame	89
3 Expanding the Series	91
4 Accessing Higher Time Frame Data by Bar	93
5 Scaling and Trading	94
Part IX Creating a Custom Indicator	95
1 Overview	95
2 Using the New Indicator Wizard	96
3 Deleting a Custom Indicator	99
4 The Guts of a Custom Indicator	99
5 Other Possibilities and FAQs	101
Part X CommissionScripts	103
1 Overview	103
2 CommissionScript Variables	103
3 Creating and Testing CommissionScripts	104
Part XI PerfScripts	106
1 Overview	106
2 PerfScript Functions	106
3 Creating PerfScripts	107
4 Using PerfScripts	108
Part XII SimuScripts	110
1 Overview	110
2 SimuScript Function Notes	110
3 How do SimuScripts Work?	112
4 Creating a SimuScript	112
5 Testing a SimuScript	114
6 SimuScript FAQs	115
Part XIII Objects	117
1 Overview	117
2 Object Type Declarations	118
3 Providing Access via Properties	119
4 Creating and Using Instances of a Type	121
5 Putting it all Together	122
6 Inheritance	123
7 Polymorphism	125
8 The TList Object	126
Overview	126
TList Functions	127

Add.....	127
AddData.....	128
AddObject.....	129
Count.....	130
Create.....	130
Data.....	131
Item.....	131
IndexOf.....	132
IndexOfData.....	132
IndexOfObject.....	133
Object.....	134
TList Procedures	134
ChangeItem.....	134
Clear.....	135
Delete.....	136
Free.....	136
SortNumeric.....	137
SortString.....	137

Index

139

1 Introduction

Welcome to the WealthScript Language Guide


The main purpose of the WealthScript Guide is to provide you with the basic (and some not-so-basic) concepts to express your trading strategies in WealthScript, which is the scripting language that you'll use within Wealth-Lab Developer 4.0. WealthScript is a complete programming language based on the standard computing language Pascal. You'll be amazed with what you can accomplish by coding trading systems with WealthScript!

Though many of the most essential WealthScript functions are used in this guide to demonstrate programming and trading system development concepts, it is not within the scope of the WealthScript Guide to highlight every single WealthScript function. All functions with syntax, descriptions, and examples, may be found in the [WealthScript Function Reference](#).


For COM Support in WealthScript, please refer to the Wealth-Lab Developer 4.0 User's Guide.

Following Along with the Examples

As you come across examples in the Reference we suggest actually typing the code or at least copying and pasting the examples to get a feel for how to create scripts. To do this, perform the following steps:

1. Click the *New* button  or select the "File/New ChartScript" menu item. This action will create a new ChartScript Window, and position you within the ChartScript Editor.
2. The Editor will contain some boilerplate code common to most new scripts. Delete this code.
3. Type in the code from the example, or copy and paste it into the Editor.
4. To execute the script, change to the Chart view in the ChartScript Window. Then, click any of the stock symbols in the DataSource Tree.

So that you can see dynamic data or data stored in variables, many examples output their results to the *Debug Messages* window. To see this window you can do one of the following actions:


- Strike the **F11** key, or,
- Select View/Debug Window, or,
- Click the *Debug Messages* button  in the toolbar.

Syntax Notes

Some topics include code syntax. When an optional statement is encountered, it shall be enclosed in brackets. For example, in the following code fragment the 'begin' and 'end;' statements are optional.

```
if booleanexpression then  
  [begin]  
    statement;  
  [end];
```

WealthScript Function Reference

For a complete list of functions available in Wealth-Lab Developer 4.0, please refer to the WealthScript Function Reference .

2 WealthScript Language Syntax

2.1 Overview

The following sections describe the basic syntax you must use when writing scripts in Wealth-Lab Developer 4.0. When you become comfortable using the basic syntax, more advanced programming techniques are available under the [Objects](#)^[117] topic and also the Wealth-Lab Developer 4.0 [Add-On API](#) on the Wealth-Lab web site.

[Comments](#)^[8]

Use comments to annotate your code.

[Statements and Delimiters](#)^[8]

A WealthScript program is composed of a series of statements that are delimited by semicolons.

[Variables and Data Types](#)^[9]

Variables are place holders in computer memory that store values that will likely vary (hence "variables") during the execution of your code.

[Assignment Statements](#)^[14]

Use assignment statements to place values into your variables.

[Constants](#)^[16]

Declare constants for values in your scripts that will never change. WealthScript [pre-defined constants](#)^[16] give you quick access to named price series and help make your code more readable.

[Operations](#)^[18]

Use operators to manipulate numeric and string expressions within your WealthScript code.

[Conditional Statements](#)^[26]

Use conditional statements to compare and test expressions with the purpose of controlling the flow (order) of execution in your WealthScript code.

[Case Statement](#)^[30]

Group a set of cases into blocks of code to improve your script's organization and readability.

[Looping Statements](#)^[32]

Use looping statements to repeat the execution of one or more statements numerous times

[Functions and Procedures](#)^[35]

Write your own functions and procedures when you use the same block of code over and over in different parts of a script. Go one extra step by saving them to the "Studies" folder and you'll be able to use them over and over

[Error Handling](#)^[44]

Write robust scripts by expecting and handling errors that occur in your code.

[Arrays](#)^[45]

Use arrays to index and then iterate through a list of elements of the same data type.

2.2 Comments

You can use **comments** to annotate your code. Comments don't affect the execution of the WealthScript, and can be a useful documentation tool. There are several forms of comments available.

Comment Blocks

Use the curly braces to create a comment block.

Example

```
{ This is a comment block
  this text will not be executed
  by the script }
```

Single Line Comments

Use the "//" characters to create single line comments.

Example

```
//This is truly the Holy Grail of Trading Systems!
{ Code Omitted }
```

2.3 Statements and Delimiters

A WealthScript program is composed of a series of **statements**. WealthScript executes the statements in order, from top to bottom. You can use [Conditional Statements](#)^[26] and [Looping Statements](#)^[32] to control this flow of execution.

Semicolons

Each WealthScript statement must end with the semicolon character (;). The semicolon lets WealthScript know that one statement is completed and another one is beginning. The following example indicates that carriage return/line feeds and other formatting characters are essentially ignored by the compiler.

Example

```
This is one statement;  
This is another statement;  
  
This is  
one statement;  
This is  
another  
statement  
;  
This is yet another;  
  
This is one statement;This is another;
```

Note: An exception exists to using line feeds - a [string type](#)^[11] cannot extend across more than one line.

Be sure to read the [WealthScript Style Guide](#) article on the Wealth-Lab.com site for guidance on formatting your code. A consistent block-formatting style will help reduce programming errors and make your code easier to read and maintain.

2.4 Variables and Data Types

2.4.1 Overview

Variables

A **variable** is a placeholder in computer memory that can store a particular value. Each variable has its own unique name, much like a PO Box in a Post Office. You can use the variable name to recall or modify the value contained in the variable.

[Declaring Variables](#)^[10]

You cannot refer to a variable in your code without declaring it first.

[Variable Naming Rules](#)^[10]

Name a variable anything you like, but follow the rules!

[Data Types](#)^[11]

Declare your variables based on the type of data they will hold.

[Record Types](#)^[12]

Record Types are useful structures for grouping varied, yet related data into a single variable type. They can be used, for example, to pass data between procedures in order to make long parameter lists saner.

[Enumerated Types](#)^[13]

Enumerated Types are special data types that you define. When defining an Enumerated Type you specify a list of possible values, each with its own unique label. Variables declared for the type can only assume one of these values.

See Also: [Scope of Variables](#)^[42] in the chapter [Functions and Procedures](#)^[35]

2.4.2 Declaring Variables

Use the **var** statement to declare a variable in your WealthScript code.

Syntax

```
var variablelist : variabletype;
```

Item	Description
<i>variablelist</i>	A single variable name, or a comma-separated list of variables that follow the variable-naming rules ^[10] .
<i>variabletype</i>	One of the valid data type names ^[11] .

Remarks

- You can declare multiple variables of the same data type with a single **var** statement by separating each new variable name with a comma as shown below.
- The variable declaration must occur before you use the variable in your code.
- Variable names are not case sensitive. Therefore, you may refer to a variable declared as *MyVariable* equally as *MYVARIABLE*, *myVariable*, *MyVaRIAbLe*, etc.

Example

```
var MyVariable: integer;  
var Var1, Var2: integer;  
var Var1: integer; var Var2: float;  
var Name, Rank, Serial_Number: string;  
var IsLong: boolean;
```

Tip: If you forget to declare a variable in your code, the compiler will give you an "Unknown name" error when you try to run your script. You can quickly fix this error by pressing F4 or by selecting "Chart/Fix ChartScript" from the main menu.

2.4.3 Variable Naming Rules

You can name your variables anything you like, provided that you follow these **rules**:

Rule 1: Variable names must begin with an alphabetic character.

Rule 2: Variable names can contain alphabetic, numeric, or underscore characters only.

Rule 3: You cannot create variables that have the same name as WealthScript reserved words or built-in function names.

Tip: When using many variables, sometimes it can be difficult to remember their data type without referring to their declaration. You can help yourself by using the same prefix for all variables of the same type. For example you could use "f" as a prefix for variables of type float (e.g. fSimpleMovingAvg, fStdDeviation, etc.).

These are suggested prefixes using a 1-letter or 3-letter "Hungarian-style" notation. Use them only if they seem helpful to you.

<code>flt, f</code>	float (examples: <code>fClose</code> , <code>fltClose</code>)
<code>int, i</code>	integer
<code>bln, b</code>	boolean
<code>str, s</code>	string
<code>vnt, v</code>	variant
<code>rcd, r</code>	record type
<code>lst, l</code>	TList object
<code>pne, p</code>	pane reference (integer)
<code>hdl, h</code>	Price Series handle (integer)

2.4.4 Data Types

A variable must be declared as one of the following **data types**. For typical syntax, see the [Assignment Statements](#)^[14] topic.

integer

Stores whole number values. Values can range from -2,147,483,648 to 2,147,483,647. You can perform mathematical [Operations](#)^[18] on integer variables.

float

Stores floating point values. The WealthScript engine treats declared floats (and arrays of type float) with double-precision, which have 14 to 15 digits of significance. Approximate valid ranges are as follows:

Negative values: -1.7×10^{308} to -4.9×10^{-324}
 Positive values: 4.9×10^{-324} to 1.7×10^{308}

You can perform mathematical [Operations](#)^[18] on float variables.

Note: Price Series values are stored as *single-precision* floating point values, which maintain 7 to 8 significant digits and can range from 1.5×10^{-45} to 3.4×10^{38} . For more information, see Data Precision Considerations in the User Guide.

string

Can store textual data of any length. You can perform string [Operations](#)^[18] on string variables.

boolean

Can contain one of two logical values: **true** or **false**. You can perform logical [Operations](#)^[18] on boolean variables.

variant

A special type of variable that can be assigned to any basic data type. A variant can be useful if you need to use the same variable for multiple types at run time.

datetime (not supported)

In WealthScript code, dates are accessed as integer values, allowing date comparison using standard arithmetic operators. For more information, see `GetDate` and all the Date/Time functions in the WealthScript Function Reference.

See Also: [Record Types](#)^[12] [Object Type Declarations](#)^[118] [TList Object](#)^[126]

2.4.5 Record Types

A useful structure for organizing a related set of data is a user-defined **Record Type**. Record Types are multi-dimensional variables that can be used in passing data between procedures, for example, to make long parameter lists saner.

Although they are not necessary for programming in WealthScript, it's nice to know these types of structures are available if you need them.

Note: Records cannot be added to a [TList object](#)^[126]. Instead, you can add an object using the AddObject method. [Objects](#)^[117] can contain different data elements just like a record type.

Syntax

type

```
rtypename = record
  vlistname1 : datatype;
  vlistname2 : datatype;
  :
  :
  vlistnameN : datatype;
end;
```

Item	Description
<i>rtypename</i>	A valid ^[10] variable name.
<i>vlistnameN</i>	A single variable name, or a comma-separated list of variables that follow the variable-naming rules ^[10] .
<i>datatype</i>	A data type ^[11] expression (e.g., integer) or an array declaration (e.g., array[0 .. 0] of float)

Example

```
{ define a record type named PriceData having
  1 datetime, 4 floats, 1 integer, and 1 boolean }
type
  PriceData = record
    dT: integer;
    O, H, L, C: float;
    V: integer;
    IsIndex: boolean;
  end;

{ function to convert a boolean to a string }
function BlnToStr( bln: boolean ): string;
begin
  if bln then
    Result := 'True'
  else
    Result := 'False';
end;

{ declare variables as the record type PriceData }
var pd1, pd2: PriceData;
const fmtPd = '#.00';
```

```

pd1.dT := 20030520;
pd1.O := 12.10;
pd1.H := 14.31;
pd1.L := 11.92;
pd1.C := 14.24;
pd1.V := 1023500;
pd1.IsIndex := False;

{ copy the data to another PriceData type }
pd2 := pd1;

Print( IntToStr(pd2.dT) + ', '
      + FormatFloat(fmtPd, pd2.O) + ', '
      + FormatFloat(fmtPd, pd2.H) + ', '
      + FormatFloat(fmtPd, pd2.L) + ', '
      + FormatFloat(fmtPd, pd2.C) + ', '
      + IntToStr(pd2.V) + ', '
      + BlnToStr(pd2.IsIndex) );

{ just for practice, let's do the same with an array of a Record Type }
var pda: array[0..1] of PriceData;

pda[0] := pd2;
pda[1] := pda[0];

Print( 'Second array contents:' );
Print( IntToStr(pda[1].dT) + ', '
      + FormatFloat(fmtPd, pda[1].O) + ', '
      + FormatFloat(fmtPd, pda[1].H) + ', '
      + FormatFloat(fmtPd, pda[1].L) + ', '
      + FormatFloat(fmtPd, pda[1].C) + ', '
      + IntToStr(pda[1].V) + ', '
      + BlnToStr(pda[1].IsIndex) );

```

2.4.6 Enumerated Types

The **Enumerated Type** is a special data type that contains a list of distinct values. You create a distinct label for each possible value of an Enumerated Type. Enumerated Types can be used to make your code more self-descriptive. For example, your trading system might look for a complex sequence of events before triggering a signal. Rather than using an integer variable to store the system's state, you could use an Enumerated Type. The script is then easier to understand because the labels of the Enumerated Type values are descriptive.

Syntax

```
type TMyType = ( valOne [, valTwo] ...[, valLast] );
```

Item	Description
<i>TMyType</i>	A valid ^[10] variable type name.
<i>valOne - valLast</i>	Each possible value of the Enumerated Type must be provided a unique valid ^[10] label. By convention, each label begins with the same brief prefix. You must provide at least one label.

Enumerated Type Values

A variable that is an Enumerated Type can only contain a value that was defined in the Enumerated Type's list. Internally, the values are stored as integers. You can convert an Enumerated Type variable to an integer by **casting** it as an integer value.

Example

```
type TEnum = ( enumZero, enumOne, enumTwo );
var n: integer;
var et: TEnum;
et := enumOne;
n := integer( et );
ShowMessage( IntToStr( n ) );
```

See Also: [Creating Synchronized Arrays](#) ⁴⁵

State Machines

The example below is a simple trading system "state machine". The system can be in one of three different states. The state is controlled by an Enumerated Type variable.

Example

```
type
  TSystemState = ( ssSetup, ssTactical, ssPinpoint );

var Bar: integer;
var State: TSystemState;

InstallStopLoss( 5 );
InstallProfitTarget( 10 );

for Bar := 20 to BarCount - 1 do
begin
  ApplyAutoStops( Bar );
  if not LastPositionActive then
  begin
    case State of
      ssSetup:
        if CumDown( Bar, #Close, 4 ) >= 9 then
          State := ssTactical;
      ssTactical:
        if RSI( Bar, #Close, 14 ) < 40 then
          State := ssPinpoint;
      ssPinpoint:
        if CumDown( Bar, #Close, 2 ) >= 3 then
        begin
          BuyAtMarket( Bar + 1, '' );
          State := ssSetup;
        end;
    end;
  end;
end;
end;
```

2.5 Assignment Statements

Use **assignment statements** to place values into your variables. Assignment statements use the **assignment operator**, which is typed as a colon immediately

followed by an equal sign.

Example

```
var n: integer;
n := 100;

var s: string;
{ Note that a string cannot extend across multiple lines in the Editor }
s := 'My name is Smith';

var f: float;
f := 3.1415;

var b: boolean;
b := true;
```

It's illegal to assign the wrong data type into a variable. The following examples will generate an error.

Example

```
var n: integer;
n := 1.234;

var s: string;
s := 200;

var f: float;
f := 'Illegal';
```

You can also assign the value from one variable into another.

Example

```
var var1, var2: integer;
var1 := 2001;
var2 := var1;
```

Initializing Variables

Generally, you should *initialize* variables, i.e., assign known values to variables, before using them for the first time in a calculation. Note that the previous examples use separate statements for declarations and assignments to initialize a variable.

Another spacing-saving technique involves [declaring](#)^[10] and initializing a variable in a single var statement. In some cases, such as within procedures or functions for example, this type of combined declaration/initialization may make your code more clear or readable. The expression on the right side of the assignment can also be a [function](#)^[35].

Example

```
var Yr, MyDay: integer;
var Img: string;
Yr := 2001;
MyDay := 16;
Img := 'RedDiamond';
```

Can be coded equivalently as follows. Note that an equals sign is used, not the assignment operator.

```
var Yr: integer = 2001;
var MyDay: integer = 16;
var Img: string = 'RedDiamond';
```

2.6 Constants

A constant is a numeric or string value in a script that will *never change*. Using constants can save you from having to repeat the same values multiple times in a script with the added advantage of making your code more concise and readable. Since constants are not variable they are never used on the left side of the [assignment operator](#)^[14].

For example, you might use a format string to format a value for debug printing. Rather than specifying the format argument each time you use a **Print** statement, you could define it as a constant and then use the constant as the format argument in each statement.

Declaring a Constant

To declare a constant, use the keyword **const** followed by the equal sign and then the value of the constant. In the example below, the constant *FMT* is set to a string, and therefore may be used in any function requiring a parameter of type **string**. You can, however, declare a constant with a numeric value (integer or float) as well.

Example

```
const FMT = '$#,##0.00';
var Bar: integer;
Bar := BarCount - 1;
DrawLabel( 'Open = ' + FormatFloat( FMT, PriceOpen( Bar ) ), 0 );
DrawLabel( 'High = ' + FormatFloat( FMT, PriceHigh( Bar ) ), 0 );
DrawLabel( 'Low = ' + FormatFloat( FMT, PriceLow( Bar ) ), 0 );
DrawLabel( 'Close = ' + FormatFloat( FMT, PriceClose( Bar ) ), 0 );
```

Pre-defined Constants

WealthScript has several constants available for you to use that will improve your code's readability. For more information, click the links.

[Price Series constants](#)^[50]

#Open, #High, #Low, #Close, #Volume, #OpenInterest, #Average, #AverageC
 #Equity ([PerfScripts](#)^[106] only)

Color value constants^[69]

#Black, #Maroon, #Green, #Olive, #Navy, #Purple, #Teal, #Gray, #Silver, #Red, #Lime, #Yellow, #Blue, #Fuchsia, #Aqua, #White, and finally #WinLoss, which is used primarily for PerfScripts^[106].

Light colors, normally used for shading the chart background:

#RedBkg, #BlueBkg, #GreenBkg

Plot formatting constants^[67]:

#Thin, #Dotted, #Thick, #Histogram, #ThickHist, #Dots

Style parameter constants (see PlotSymbol):

#OHLC, #Candle, #Line

PerfScript Style parameter constants^[106]

#Bold, #Italic

Time Frame constants (see ChangeScale):

#Daily, #Weekly, #Monthly

Day of the Week constants (use with DayOfWeek function):

#Monday, #Tuesday, #Wednesday, #Thursday, #Friday

Current SimuScript Position^[110]:

#Current

Shortcut to Closing All Positions^[79]: (use with SellAt and CoverAt functions)

#All

ChartScript Optimization Variables

#OptVars are values that will be replaced with a range of different values during the optimization process. You can use up to 10 #OptVars, #OptVar1 through #OptVar10.

#OptVar1, #OptVar2, ..., #OptVar10

Set Mode constants

The **SetAutoStopMode** WealthScript function allows you to control how the parameter of AutoStops are interpreted.

#AsPercent (default), #AsPoint, #AsDollar

The first two constants are also used in the **SetPeakTroughMode** WealthScript function to control how the *Reversal* parameter of Peak and Trough functions are interpreted.

2.7 Operations

2.7.1 Overview

There are four different types of **operations** you can perform in WealthScript; mathematical, boolean, logical, and string.

[Mathematical Operations](#)^[18]

Use the standard mathematical operators to manipulate numeric expressions.

[Boolean Operations](#)^[19]

Test relationships between expressions using boolean operators.

[Logical Operations](#)^[21]

Make logical comparisons between two numeric expressions with this subset of boolean operators.

[String Operations](#)^[25]

Concatenate and compare string variables and expressions.

2.7.2 Mathematical Operations

Standard Operators

You can use the standard mathematical operators summarized in the table below in your WealthScript code.

Syntax

Result := *Operand1* **Operator** *Operand2*;

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division

Multiplication and division operations are evaluated first, otherwise expressions are evaluated from left to right. You can use parenthesis to modify the standard order of evaluation, where the innermost expression is evaluated first.

Example

```
var x: integer;
x := 1 / 2;
x := x * 5 + 1;
x := ( x - 5 ) / ( x * 2 );
x := x - ( 2 / ( 3 * x ) );
```

More advanced mathematical operations can be completed using the built-in Math Functions.

Modulo Operator

The **Mod** operator is used to divide two floating-point numbers, which are first rounded to integers, and returns only the *remainder* as type **float**. Although the divisor may be a negative number, the result will always maintain the sign of the dividend.

Syntax

Result := *dividend* **Mod** *divisor*;

Example

```
{ y will equal -5 and z equals 0 }  
var y, z: float;  
y := -21 Mod 7.8;  
z := 21 Mod 7.3;  
ShowMessage( FloatToStr(y) + #9 + FloatToStr(z) );
```

See Also:

ModX function and **IGArithm01** functions in the [Wealth-Lab Code Library](#) on the Wealth-Lab site.

Div Operator

There are times when you may want to be sure that the result of an integer division returns an integer. Whereas **Mod** returns a remainder, division with the **Div** operator returns an **integer** quotient (without a remainder).

Syntax

Result := *dividend* **Div** *divisor*;

Remarks

dividend and *divisor* must be integer expressions.

Example

```
{ i will be assigned the value -3 }  
var i, j: integer;  
j := -6;  
i := 21 Div -j;  
ShowMessage( IntToStr( i ) );
```

2.7.3 Boolean Operations

Nearly all programs require you to test [boolean] relationships between numeric variable and perhaps even string variables. For these tests you'll use the standard set of Pascal boolean operators found in the table below:

Syntax

Result := *Operand1* **Operator** *Operand2*;

Operator	Description
=	Equal to
<>	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

The result of comparing two expressions with the operators above is a boolean (True or False). Consequently, you'll often assign the result of relational operations to a boolean variable as in the example below:

Example

```
var b: boolean;
var x1, x2: float;
x1 := 10;
x2 := 20;
b := true;      {true}
b := false;     {false}

b := x1 = x2;    {false}
b := x1 <> x2;    {true}
b := x2 > x1;    {true}
```

You can also use boolean expressions whenever a boolean is required without assigning the result to a boolean variable, as in the **if/then** statement below.

Example

```
var b: boolean;
var x1, x2: float;
x1 := 10;
x2 := 20;
if x2 < x1 then
  x1 := x1 * x2;
b := ( x1 > x2 ) Or ( x1 > 1 );  {true}
```

Note that when using a logical operator you must group the individual expressions in parenthesis, as in the final assignment using **Or** in the example above.

See Also: [Logical Operations](#) 

2.7.4 Logical Operations

2.7.4.1 Summary

The following operators allow you to perform logical comparisons between two numeric expressions. With these operators, you have the additional capability to perform bitwise comparisons of two identically positioned bits in two numeric expressions.

And Operator

Perform logical *conjunctions* of expressions with the **And** operator.

Or Operator

Perform logical *disjunctions* of expressions with the **Or** operator.

Xor Operator

Perform logical *exclusions* of expressions with the **Xor** operator.

Not Operator

Perform logical *negations* of expressions with the **Not** operator.

Note: When using a logical operator to obtain the result of two boolean expressions, you must group the individual boolean expressions in parenthesis.

Example

```
var TestIsTrue: boolean;
TestIsTrue := ( 2 > 1 ) And ( 2 + 2 = 5 );
If TestIsTrue then
  ShowMessage( 'The expression is True!')
else
  ShowMessage( 'The expression is False!');
```

2.7.4.2 And Operator

You may perform logical conjunctions of expressions with the **And** operator.

Syntax

Result := *Expression1* **And** *Expression2*;

Item	Description
<i>Result</i>	A boolean variable.
<i>Expression1</i>	Any boolean expression. Expressions including operators should be enclosed in parentheses.
<i>Expression2</i>	Any boolean expression. Expressions including operators should be enclosed in parentheses.

Result Summary:

Expression1	Expression2	Result
False	False	False
False	True	False
True	False	False
True	True	True

Example

```

var b1, b2, bR: boolean;
b1 := True;  b2 := False;
bR := b1 And b2;  { bR is assigned False }

b1 := 20 < 23;  b2 := 30 > 29;
bR := b1 And b2;  { bR is assigned True }

```

Integer Bitwise Comparison

Likewise, you may also use the **And** operator to compare two identically positioned bits in two numeric expressions.

And Bitwise Comparison Result Summary:

bit in Expression1	bit in Expression2	Result
0	0	0
1	0	0
0	1	0
1	1	1

Example

```

var x, y, z: integer;
x := 9;  y := 1;
z := x And y;  { z equals 1; 1001 And 0001 = 0001 }

x := 7;  y := 12;
z := x And y;  { z equals 4; 0111 And 1100 = 0100 }

```

2.7.4.3 Or Operator

You may perform logical disjunctions of expressions with the **Or** operator.

Syntax

Result := Expression1 Or Expression2;

Item	Description
<i>Result</i>	A boolean variable
<i>Expression1</i>	Any boolean expression. Expressions including operators should be enclosed in parentheses.
<i>Expression2</i>	Any boolean expression. Expressions including operators should be enclosed in parentheses.

Result Summary:

Expression1	Expression2	Result
False	False	False
False	True	True
True	False	True
True	True	True

Example

```
var b1, b2, bR: boolean;
b1 := True;  b2 := False;
bR := b1 Or b2; { bR is assigned True }

b1 := 20 < 23;  b2 := 30 > 29;
bR := b1 Or b2; { bR is assigned True }
```

Integer Bitwise Comparison

Likewise, you may also use the Or operator to compare two identically positioned bits in two numeric expressions.

Or Bitwise Comparison Result Summary:

bit in Expression1	bit in Expression2	Result
0	0	0
0	1	1
1	0	1
1	1	1

Example

```
var x, y, z: integer;
x := 9;  y := 3;
z := x Or y; { z equals 11; 1001 And 0011 = 1011 }

x := 7;  y := 8;
z := x Or y; { z equals 15; 0111 And 1000 = 1111 }
```

2.7.4.4 Xor Operator

You may perform logical exclusions of expressions with the **Xor** operator.

Syntax

Result := *Expression1* **Xor** *Expression2*;

Item	Description
<i>Result</i>	A boolean variable
<i>Expression1</i>	Any boolean expression. Expressions including operators should be enclosed in parentheses.
<i>Expression2</i>	Any boolean expression. Expressions including operators should be enclosed in parentheses.

Result Summary:

Expression1	Expression2	Result
False	False	False
False	True	True
True	False	True
True	True	False

Example

```
var b1, b2, bR: boolean;
b1 := True;  b2 := False;
bR := b1 Xor b2; { bR is assigned True }

b1 := 20 < 23;  b2 := 30 > 29;
bR := b1 Xor b2; { bR is assigned False }
```

Integer Bitwise Comparison

Likewise, you may also use the Xor operator to compare two identically positioned bits in two numeric expressions.

Xor Bitwise Comparison Result Summary:

bit in Expression1	bit in Expression2	Result
0	0	0
0	1	1
1	0	1
1	1	0

Example

```

var x, y, z: integer;
x := 9; y := 3;
z := x Xor y;    { z equals 10; 1001 And 0011 = 1010 }

x := 7; y := 8;
z := x Xor y;    { z equals 15; 0111 And 1000 = 1111 }

```

2.7.4.5 Not Operator

You may perform logical negations of expressions with the **Not** operator.

Syntax

Result := **Not** *Expression*;

Item	Description
<i>Result</i>	A boolean variable
<i>Expression</i>	Any boolean expression. Expressions including operators should be enclosed in parentheses.

Result Summary:

Expression	Result
False	True
True	False

Example

```

var b, bR: boolean;
b := True;
bR := Not b;    { bR is assigned False }

b := 20 > 23;
bR := Not b;    { bR is assigned True }

```

2.7.5 String Operations

The only valid string operation that changes the value of a string variable is **concatenation** (+), which appends multiple strings into a single string.

Example

```

var s1, s2, getty: string;
s1 := 'Four score and';
s2 := 'seven years ago';
getty := s1 + ' ' + s2;
ShowMessage( getty );
{ getty now holds the string 'Four score and seven years ago' }

```

For non-printable characters, use the **Chr**(*ascii*code) function instead of a literal

string, where *ascii*code is the decimal ASCII code of a printable or non-printable character. Alternatively, you may use the shorthand "#*ascii*code" notation. Below is a partial list of handy non-printable characters.

Decimal Code	Description
9	TAB, tab character
10	LF, line feed
13	CR, carriage return

Tip:

If you want to break a string into multiple lines, add carriage return and line break character codes to the location of the line break. In the example above, replace the string-assignment statement as follows:

```
getty := s1 + Chr(13) + Chr(10) + s2;
```

```
{ Or, using the shorthand notation: }  
getty := s1 + #13#10 + s2;
```

String Comparison

You may also make comparisons between string variables using the [boolean operators](#). A boolean operation on alphanumeric strings results in a binary (case-sensitive) comparison of the string expressions.

When comparing strings, characters are tested from left to right until an inequality is found. The value of a string character used for comparison is its associated ASCII code. Therefore, an alphanumeric character such as '3' having an ASCII code of 53, will evaluate as being less than any letter, which have ASCII codes starting at 65.

Example

```
var s1, s2: string;  
var b: boolean;  
s1 := 'OU812';  
s2 := 'Oh, me?';  
b := s1 < s2; { b is True }  
  
s2 := 'Ou812';  
b := s1 = s2; { b is False }
```

2.8 Conditional Statements

Conditional statements allow you to control the flow of execution in your WealthScript programs. You'll use the **if**, **then** and **else** statements for this purpose.

If/Then Statements

Use the **if/then** statement to perform logical tests. The program can branch to one set of statements if the test is true, and another if the result is false. You can use any of the logical operations in the **if/then** statement.

Syntax

```
if booleanexpression then  
  [begin]  
    statement;  
  [end];
```

Note that the **if/then** and the statements contained within it are considered as a single WealthScript statement, so you place a semicolon after the final statement executed, as shown below.

Example

```
var x: integer;  
x := 10;  
if x > 10 then  
  x := x + 1;    {will not execute}  
if x <= 10 then  
  x := x * 2;    {will execute}  
if ( x = 20 ) or ( x = 10 ) then  
  x := x / 3;    {will execute}
```

You can also test a boolean variable directly. This can make your code more readable if you creatively name your variables.

Example

```
var f1, f2: float;  
var IsTrue: boolean;  
f1 := 30.5;  
f2 := 29.0;  
IsTrue := f2 < f1;  
if IsTrue then  
  Print('Sell Now!');
```

Executing Multiple Statements After an If/Then

Often you'll want to execute more than one statement after an **if/then**. In this case you must use a **begin/end** statement pair to create a "code block" that encloses the statements. The **begin/end** code block concept is used in other areas of WealthScript, whenever a group of statements need to be treated as a single statement.

Syntax

```
if booleanexpression then  
  begin  
    statement1;  
    statement2;  
    :  
    :  
    statementX;  
  end;
```

The **begin/end** code block is considered a single statement, so the semicolon goes after the **end** portion of the pair. However, you can place as many other statements as you like within the **begin/end** code block. These individual statements within the **begin/end** should end with semicolons.

Example

```
var x: integer;
x := 10;
{ This code block contains no statements }
if x < 20 then
begin
end;

{ This if/then will execute 3 statements }
if x * 2 = 20 then
begin
  x := x * 2;
  x := x - 1;
  x := x / 10;
end;
```

Note that each of the 3 statements within the **begin/end** block ends with a semicolon.

The Else Statement

You can use the else statement to execute statements if the **if/then** test resolves to false. In this form, the **if/then/else** is considered a single statement, so the semicolon goes at the very end of the statement only.

Syntax

```
if booleanexpression then
  statement
else
  statement;
```

Example

```
var x: integer;
x := 10;
if x = 5 then
  x := x * 20
else
  x := x / 20;
```

Complex If/Then/Else with Begin/End

You can, of course, use begin/end code blocks in either or both portions of the **if/then/else** statement.

Example

```
{ if/then/else with begin/end blocks, no code in the blocks }
var x: integer;
x := 10;
if x < 10 then
begin
end
else
begin
end;
```

```

{ if/then/else with begin/end blocks, with code in the blocks }
var x: integer;
x := 10;
if x < 10 then
begin
    x := x * 2 + 1;
    x := x / 5;
end
else
begin
    x := x * x;
    x := x / 2;
end;

```

Note that there is no semicolon after the first **begin/end** pair in the **if/then/else** with code blocks. The semicolon appears after the last **end** only.

Nested If/Then If/Then/Else

You can "nest" one or more if/then/else statements within another.

Example

```

var x: integer;
x := 10;

{ These are two nested if/then statements }
if x = 10 then
    if x * 2 < 20 then
        begin
            x := x / 3;
            x := x + 2;
        end;

{ This is a nested if/then/else block. }
if x < 2 then
begin
    x := x * 10;
    x := x - 5;
end
else
if x > 5 then
begin
    x := x * 100;
    x := x * x;
end
else
begin
    x := ( x + 1 ) / x;
    x := x * 2;
end;

```

Note that the first if/then block in the example above is equivalent to the following if/then block that uses the **And** logical operator.

Example

```
var x: integer;
x := 10;
{ Reminder: boolean expressions must be grouped in parentheses
  when combined by a logical operator }
if ( x = 10 ) And ( x * 2 < 20 ) then
begin
  x := x / 3;
  x := x + 2;
end;
```

2.9 Case Statement

A case statement examines a variable and lets you execute a different statement or group of statements depending on its value. Each "case" can include a single value, a list of values separated by commas, or define a range between two values (included in the range) using a double-dot notation (..) between the values. Place a colon after the end of the value lists. After each case is defined, you can place a single statement to be executed, or a group of statements surrounded by a **begin/end** block.

Use the **else** statement to execute statements when a value doesn't fall within any of your pre-defined cases. The **begin/end** statements are optional after **else** in a **case** statement, even if you have multiple statements in the **else** block.

Note: You can use all comparative data types in the case instruction, i.e., including strings, floats, and even booleans; although use of floats and booleans in case statements are uncommon.

Single Value Case Statements

Syntax

```
case testexpression of
  casevalue1:
    [begin]
      statements;
    [end;]
  casevalue2, casevalue3, ..., casevalueX:
    statement;
  casevalueY:
    statement;
else
  [begin]
    statements;
  [end;]
end;
```

The example below shows a **case** statement that operates on single values only.

Example

```
var n: integer;
n := Round( Random * 5 ) + 1;
case n of
  1:
    ShowMessage( 'One' );
  2:
    ShowMessage( 'Two' );
  3:
    ShowMessage( 'Three' );
  4:
    ShowMessage( 'Four' );
  else
    ShowMessage( 'None of the Above' );
end;
```

Case Statements Using a List of Values

The example below uses a list of values for the cases. It also shows how to use **begin/end** blocks to execute multiple statements for a **case**.

Example

```
var n: integer;
n := Round( Random * 10 ) + 1;
case n of
  1, 2:
    begin
      ShowMessage( 'One, Two' );
      ShowMessage( 'Buckle my Shoe' );
    end;
  3, 4:
    begin
      ShowMessage( 'Three, Four' );
      ShowMessage( 'Trade Some More' );
    end;
  5..8:
    ShowMessage( 'Between 5 and 8, inclusive: ' + IntToStr( n ) );
  else
    ShowMessage( 'Collect your Profits now!' );
end;
```

2.10 Looping Statements

2.10.1 Summary

Use looping statements to repeat the execution of one or more statements numerous times. There are several types of looping techniques possible:

For Loop^[32]

The **for** loop uses an *index variable* to repeat a statement or block of statements.

While Loop^[33]

This type of loop continues to execute *while* a test condition evaluates True.

Repeat Loop^[34]

Similar to a the While loop, a Repeat loop makes sure that the statements within the loop are executed at least once.

Breaking Out of a Loop^[34]

It's not always necessary to run a loop to its completion. When the code inside a loop has served its purpose, use the **break** statement to terminate a loop to save processing time.

2.10.2 For Loop

The **for** loop uses an *index variable* to repeat a statement or block of statements. Within the repeated statement block you can access the value of the variable used to control the loop.

Syntax

```
for numericvariable := start to end do  
  [begin]  
    statements;  
  [end;]
```

If you want the **for** loop to repeat more than a single statement you must enclose the statements in a **begin/end** block.

Example

```
var n: integer;  
var x: float;  
x := 2;  
  
{ Repeat a single statement 10 times }  
for n := 1 to 10 do  
  x := x * 2;  
  
{ Repeat a group of statements 10 times }  
for n := 1 to 10 do  
begin  
  x := x * 2;  
  x := x + 5;  
end;
```

```
{ Use the index variable in the loop }  
for n := 1 to 10 do  
begin  
  x := x + n * 2;  
  x := x / n;  
end;
```

Counting Backward

You can count backwards instead of forward in your **for** loop by using **downto** instead of **to** in the loop.

Example

```
var n: integer;  
for n := 10 downto 1 do  
begin  
end;
```

2.10.3 While Loop

Use the **while** loop to execute statements **as long as** a certain boolean condition is true. The condition should be enclosed in parenthesis, and can be any value Boolean Operation.

Syntax

```
while booleanexpression do  
[begin]  
  statements;  
[end;]
```

Example

```
var n1, n2: integer;  
n1 := 10;  
n2 := 50;  
while ( n1 < n2 ) do  
begin  
  print( IntToStr( n1 ) );  
  n1 := n1 + 3;  
end;
```

2.10.4 Repeat Loop

The **repeat** loop will execute statements until the specified condition is true. This is similar to the **while** ³³ loop, but the **repeat** loop checks the condition after the first pass of the loop. This guarantees that the statements within the loop will execute at least once.

Note that this loop does *not* require **begin** and **end** for multiple statements since the block of loop code is fully contained within the **repeat..until** keywords.

Syntax

```
repeat  
  statements;  
until booleanexpression ;
```

Example

```
var n1, n2: integer;  
n1 := 10;  
n2 := 50;  
repeat  
  print( IntToStr( n1 ) );  
  n1 := n1 + 3;  
until ( n1 > n2 );
```

2.10.5 Breaking Out of a Loop

Sometimes it's necessary to break out of a loop before it completes. There are two ways you can do this.

The **break** statement takes you completely out of the loop, and resumes execution at the statement immediately after the loop.

Syntax

```
break;
```

Example

```
var i: integer;  
for i := 1 to 10 do  
  begin  
    if Random > 0.5 then  
      break;  
  end;  
  //Execution resumes here  
  Print( IntToStr( i ) );
```

The **continue** statement takes you back to the beginning of the loop and continues with the next iteration, skipping any statements after the **continue**.

Syntax

```
continue;
```

Example

```
var i, n: integer;
n := 0;
for i := 1 to 10 do
begin
  if Random > 0.5 then
    continue;
  n := n + i;
end;
```

2.11 Functions and Procedures

2.11.1 Overview

Functions and **procedures** are blocks of code that you can execute whenever needed from any point within your script. You give these code blocks their own "name", and can then execute the code by *calling* it by name.

A rule of thumb is that when you find yourself writing the same block of code more than once in your scripts, there's a good chance that you should convert that block of code into a function (if you need a value returned) or a procedure (to do some other repetitive operation, like drawing trendlines on a chart).

Declaring Procedures ³⁶

Procedures must be declared above the calling routine using the syntax found in this topic.

Declaring Functions ³⁷

The main distinction between a function and a procedure is that a function **returns a value** to the caller, while a procedure does not. Like procedures, functions must also be declared above the calling routine.

Calling Functions and Procedures ³⁸

Similar to double-clicking on a Windows shortcut to run a program, you call functions and procedures using the name in their declarations. When the name is encountered in code, the "small program" found within the function/procedure block is run. When the function/procedure completes its routine, program execution begins at the next statement following the call.

Passing Parameters ⁴⁰

More often than not, you'll want to pass values (or objects) to functions and procedures for further manipulation. Using the parameter list, you have the choice of passing arguments *by value* or *by reference*.

Scope of Variables ⁴²

It's possible for variables to be accessed in more than one routine. If you're not careful with the placement of your variable declarations you could unknowingly be modifying the value of a variable used in multiple routines.

Exiting a Procedure ⁴³

Use the **Exit** statement to terminate a function or procedure without executing any remaining statements.

[Native and Re-usable Functions](#)^[44]

One characteristic of functions and procedures is that they are re-usable. You can build your own library of functions and procedures and include them in different ChartScripts. WealthScript itself is made up of hundreds of other "native" functions and procedures. You'll call native functions just like a function you have written yourself, but since they are part of the WealthScript language, you don't have to declare them!

2.11.2 Declaring Procedures

Like functions, procedures must be declared before they can be referenced in your script. This means that they must appear towards the beginning of your script, above the main routine of your ChartScript code.

Use the **procedure** statement followed by a unique name to declare a procedure, as shown below. Procedures follow the same [naming rules](#)^[10] as normal variables.

Syntax

```
procedure procedurename ( ( [var] variablelist1: type1; [var] variablelist2: type2;
... [var] variablelistX: typeX) );
begin
  [ procedure-scope variable declarations ]
  [ statements ]
end;
```

Item	Description
<i>procedurename</i>	A valid name that follows variable naming rules ^[10] .
<i>variablelist</i>	A single variable name, or a comma-separated list of variables that follow the variable-naming rules ^[10] . When multiple <i>types</i> exist in the parameter list, they are separated by semicolons.
<i>type</i>	One of the valid data type names ^[11] .
<i>statements</i>	WealthScript function/procedure code

Remarks:

- Use procedures when you *do not* need to return a value to the caller.
- After the **procedure** statement is a **begin/end** block that contains the code that will execute when you call the procedure.
- By default, variables are passed *by value* to procedures and functions. Use the optional statement **var** within the argument list when you want to pass an argument *by reference*. For more information, see [Passing Parameters](#)^[40] in this chapter.
- Declarations of the variables in the parameter list are sufficient for their use throughout the procedure. In the *procedure-scope declarations*, declare only additional variables you need for use within the procedure; for interim calculations, for example.
- Excluding object and record types, *procedure-scope* variables can be declared in the procedure declaration, i.e., immediately after the **procedure** statement and before the first **begin**. For this method, use one **var** statement followed by

variablename: type; as shown below.

Example

```
{ This is a procedure and therefore has no return value }
procedure DoSomething;
var
  i, j: integer;
  f: float;
  str: string;
begin
  // Your "do something" procedure code would go here
end;
```

2.11.3 Declaring Functions

Like procedures, functions must be declared before they can be referenced in your script. This means that they must appear towards the beginning of your script, above the main body of your ChartScript code.

Use the **function** statement followed by a unique name to declare a function, as shown below. Functions follow the same [naming rules](#)^[10] as normal variables.

Syntax

```
function functionname[ ( [var] variablelist1: type1; [var] variablelist2: type2; ...
```

[var] *variablelistX*: *typeX*): *returntype*;

```
begin
  [ function-scope variable declarations ]
  [ statements ]
  [Result := expression ;]
end;
```

Item	Description
<i>functionname</i>	A valid name that follows variable naming rules ^[10] .
<i>procedurename</i>	A valid name that follows variable naming rules ^[10] .
<i>returntype</i>	One of the valid data type names ^[11] .
<i>variablelist</i>	A single variable name, or a comma-separated list of variables that follow the variable-naming rules ^[10] . When multiple <i>types</i> exists in the parameter list, they are separated by semicolons.
<i>type</i>	One of the valid data type names ^[11] .
<i>statements</i>	WealthScript function/procedure code
<i>expression</i>	An expression of type <i>returntype</i>

Remarks:

- Use functions when you need to return a value to the caller. Specify the data type (*returntype*) of the **return value** (*Result*) at the end of the **function** statement, preceded by a colon.
- After the **function** statement is a **begin/end** block that contains the code that will execute when you call the function.

- By default, variables are passed *by value* to procedures and functions. Use the optional statement **var** within the argument list when you want to pass an argument *by reference*. For more information, see [Passing Parameters](#)^[40] in this chapter.
- Declarations of the variables in the parameter list are sufficient for their use throughout the function. In the *function-scope declarations*, declare only additional variables you need for use within the procedure; for interim calculations, for example.
- Excluding object and record types, *function-scope* variables can be declared in the procedure declaration, i.e., immediately after the **function** statement and before the first **begin**. For this method, use one **var** statement followed by *variablename: type;* as shown below.

Example

```
{ This simple function returns the integer 1 }  
function MyFunction: integer;  
var  
    i: integer;  
    f: float;  
begin  
    i := 50;  
    f := 0.02;  
    Result := Round( i * f );  
end;  
  
{ And this one returns the string '<WLD>!' }  
function Func2: string;  
begin  
    Result := 'Wealth-Lab Developer 4.0!';  
end;  
  
// Call the functions and show the result  
ShowMessage( Func2 + ' is Number ' + IntToStr( MyFunction ) + '!' );
```

Function Return Values

Although it is optional, it makes little sense to declare a function that does not return a value. Notice in the above function examples that an expression of type *returntype* is assigned to a variable named *Result*. The *Result* variable is a special variable that is available only in functions. Always assign the return value of your functions to the *Result* variable. The assignment may be found at any point within the function block, although as it is a "result", this statement is often the last one.

Recursion

Recursion refers to the ability of a function to call itself. Using recursive techniques, you can write very compact and efficient code that performs tasks that might be otherwise unmanageable. Recursive, or "reentrant", functions may be programmed in WealthScript. A classic example of a recursive function is one that calculates the factorial of a number, **x!**.

Example

```
function Xfactorial( x: integer ): float;  
begin
```

```

var i: integer;
var f: float;
i := x - 1;
if i < 2 then    // No more calls!
    Result := x
else
    Result := x * Xfactorial( i );
end;

{ test the function }
var y: float;
var j: integer;
for j := 0 to 10 do
begin
    y := Xfactorial( j );
    Print( IntToStr( j ) + #9 + FloatToStr( y ) );
end;

```

2.11.4 Calling Functions and Procedures

Since procedures do not return values as do functions, some differences exist in the manner in which they can be called. In both cases, remember that the function or procedure must be declared before you can access it by name. Also, it's perfectly valid to call functions from within functions, procedures from functions, etc.

Procedure Calls

There's only one way to call a procedure - by using its name in your script code as a single statement. If the procedure has an argument list, you must supply properly-typed expressions for each argument in the procedure declaration.

Example

```

{ This procedure colors the volume histogram of all up bars green and
all down or flat bars red. It is included with your installation of
Wealth-Lab Developer 4.0 in the "Studies" ChartScript folder }
procedure VolumeColor;
var Bar: integer;
begin
    for Bar := 1 to BarCount - 1 do
        if PriceClose( Bar ) > PriceClose( Bar - 1 ) then
            SetSeriesBarColor( Bar, #Volume, #Green )
        else
            SetSeriesBarColor( Bar, #Volume, #Red );
    end;
end;

```

```
{ Execute the procedure by calling it }  
VolumeColor;
```

Function Calls

Because functions return a result, more possibilities exist. As with procedures, use the function's name and provide properly-typed expressions for each argument in the function declaration.

- Most commonly, you will use a function like an expression. If the function returns a boolean, for example, you can assign the function to a boolean variable. The function call appears on the *right side* of the assignment.
- Likewise, you may use the same boolean function in any expression that requires a boolean expression - as the conditional test expression in an If/Then statement for instance.
- If you do not care about the function's result, you may call the function in the same manner as a procedure - as a stand-alone statement. The function's processing will be the same whether or not you choose to use its result in an expression or store it in a variable.

Example

```
function MyFunc: integer;  
begin  
    Result := 100;  
end;  
  
var IntVar: integer;  
Print( MyFunc + MyFunc );    //Prints 200 to the debug window  
IntVar := MyFunc;            //IntVar now contains 100
```

2.11.5 Passing Parameters

You pass parameters to a function or procedure by defining a parameter list in the function or procedure declaration. The parameter list occurs after the function or procedure name, and contains a list of parameters enclosed in parenthesis. Each parameter is declared by name and data type separated by a colon. Parameter declarations with different data types should be separated by semicolons. You can declare multiple parameters of the same data type by separating them by commas.

The parameter list that appears in the function/procedure declaration is in itself a formal declaration of the variables that will be used in the function/procedure. Of course, if you need other variables for interim calculation within the routine, they must be declared using the [conventional notation](#)^[10].

Note: You may see some examples of function or procedure calls in which two empty parentheses are used for following the name, as in `BarCount()`, which is the WealthScript function to return the total number of bars in the chart. These are simply calls to functions/procedures with blank parameter lists. In Wealth-Lab Developer 4.0 you can be sure that calling such routines with or without the empty parentheses will yield the same result. However, calls to COM methods containing blank parameter lists *may require* the empty parentheses to be included.

Example

```

{ Declare Functions and Procedures }
function Cube( Param1: float ): float;
begin
    Result := Param1 * Param1 * Param1;
end;

procedure WriteIt( Bar, Value: integer );
begin
    Print( IntToStr( Bar ) + ': ' + IntToStr( Value ) );
end;

function MySMA( Bar, Series, Period: integer ): float;
begin
    var i: integer;
    var total: float;
    total := 0;
    for i := Bar downto Bar - Period + 1 do
        total := total + GetSeriesValue( i, Series );
    Result := total / Period;
end;

{ Now call them }
var n, x: integer;
n := BarCount - 1;
x := Round( Cube( MySMA( n, #Close, 20 ) ) );
WriteIt( n, x );

```

By Reference or By Value

When the **var** statement is *not* used in a variable declaration within the argument list of a procedure or function declaration, variable parameters are passed *by value*. This means that a copy of the variable's value is created and "passed" to the function/procedure for use. Changes made within the function/procedure to a variable passed by value will not affect the original value of the variable in the caller, or calling procedure.

The opposite is true when the **var** statement *is* used. In a function's or a procedure's parameter list, the **var** statement marks the variable(s) to be passed *by reference*. When passed by reference, changes to the variable within the function/procedure will affect the value of the variable in the calling procedure as demonstrated below. (In reality, the routine operates on the same variable and what is *passed* is actually a pointer to that variable in computer memory.)

Example

```

procedure PassParams( var ChangeMe: integer; WontChange: integer );
begin
    ChangeMe := 100;
    WontChange := 100;
end;

var OneInteger, TwoInteger: integer;

OneInteger := 1;
TwoInteger := 2;
PassParams(OneInteger, TwoInteger);
ShowMessage( 'OneInteger is now ' + IntToStr(OneInteger) );
ShowMessage( 'TwoInteger is (still) ' + IntToStr(TwoInteger) );

```

Note that although a procedure does not provide a return result, it's perfectly legal to use by-reference parameters in a procedure to alter variables in the calling routine. The downside is that this advanced coding technique can lead to equally complex problems that are difficult to trace since the same variables can be altered in more than one procedure.

2.11.6 Scope of Variables

Variable scope is the extent to which your code has access to declared variables. Depending on the location of a variable declaration, it may be accessible only by a function, a procedure, the main ChartScript routine, or all of the above! Variables declared for objects using [the Type statement](#)^[118] have their own special scope as described in the [Objects](#)^[117] chapter.

Generally speaking, three levels of scope exist in your ChartScripts:

- Script-wide scope: Variables declared at the top of a ChartScript can be referenced by any routine below the declaration.
- Procedure or function scope: Variables declared within a function or procedure can be accessed only by the function or procedure in which they are declared.
- Main routine scope: Like script-wide scope, variables are available to routines below the declaration, but because of its placement, these variables cannot be accessed by code above the declaration.

These concepts are illustrated in the following example. Note that if you try to use the *MainRoutineScope* variable within the *Scoping* procedure, an error would result.

Example

```
{ A variable declared here can be accessed by any
  procedure or routine below }
var ScriptScope: integer;

procedure Scoping();
{ Variables declared within a function or procedure can be
  accessed only by the function or procedure }
begin
var ProcedureScope: integer;

    ScriptScope := 100;
    ProcedureScope := 2;
end;

{ Variables declared below are not accessible by any routine above }
var MainRoutineScope: integer;

MainRoutineScope:= 1;
Scoping;
ShowMessage('ScriptScope set by Scoping procedure = ' +
            IntToStr( ScriptScope ));
ScriptScope := 200;
```

2.11.7 Exiting a Procedure

Functions and procedures exit automatically upon processing the last statement contained therein. To terminate a function or procedure prematurely so that Wealth-Lab does not execute any of the statements that follow, use the **exit** statement. When you call **exit**, program control is passed away from the current procedure immediately, and program control resumes with the next statement following the procedure call. If **exit** is found in the main body of the ChartScript (i.e., not within a function or procedure), it terminates script processing altogether.

Syntax

exit;

Example 1

```
{ Don't run the script on the symbol 'T' }
var Bar: integer;
if GetSymbol = 'T' then
    exit;

for Bar := 20 to BarCount - 1 do
begin
    { Trading system here }
end;
```

The next example demonstrates the optimization technique used for custom indicators in accessing their data. If the function has been called previously from elsewhere in the script, the former result is found and returned to the caller. In this case, **Exit** terminates the method immediately so as not to waste time recalculating all the indicators values.

Example 2

```
{ Typical indicator usage }
function InverseFisherSeries( Series: integer ): integer;
begin
    var Bar: integer;
    var sName: string;
    var Value, e2y, y: float;

    sName := 'InverseFisher(' + GetDescription( Series ) + ')';
    Result := FindNamedSeries( sName );
    if Result >= 0 then
        Exit;
    Result := CreateNamedSeries( sName );
    for Bar := 0 to BarCount - 1 do
    begin
        e2y := exp( 2 * GetSeriesValue( Bar, Series ) );
        Value := ( e2y - 1 ) / ( e2y + 1 );
        SetSeriesValue( Bar, Result, Value );
    end;
end;
```

2.11.8 Native and Re-usable Functions

Native Functions and Procedures

WealthScript contains hundreds of built-in functions and procedures that you'll use extensively in your scripts to control trading rules, plot indicators, and annotate the chart. You call these functions and procedures just as you'd call one that you'd created yourself. The WealthScript Function Reference⁵¹ and the Function QuickRef contain a full list of the native functions and procedures available in WealthScript.

Including Functions and Procedures

You can build your own library of re-useable functions and procedures and include them in different ChartScripts. This is a powerful capability that can save you hours of copy and pasting effort, and makes it much easier to maintain your code. See the Include Manager topic for more information.

2.12 Error Handling

When a ChartScript encounters a compilation or run-time error, processing stops and an error message appears below the ChartScript Editor. You can click on the error message to pinpoint the line of code that generated the error.

Other run-time and logic errors occurring in a function or procedure can be more difficult to isolate and solve. This is because the error in the ChartScript Editor will point to the statement calling the function or procedure. See the ChartScript Integrated Debugger topic in the Wealth-Lab Developer 4.0 Users Guide for information in troubleshooting these and other types of coding bugs.

Handling Errors

There might be cases where you expect that an error might occur, but you want to continue processing in the script regardless. WealthScript uses the concept of *structured exception handling* to let you handle errors.

Use the `try/except/end` statement block to enclose sections of code that might contain errors. If an error occurs anywhere within the try and except statements, program flow is transferred immediately into the first statement after the except. If you want to handle errors silently just don't write any statements between the except and the end.

In this example we try and store a value in a custom Price Series without having created the Price Series using CreateSeries. We trap and report the error and continue with execution of the script.

Example

```

var MySeries: integer;
try
  SetSeriesValue( 100, MySeries, 1.234 );
except
  ShowMessage( 'CreateSeries wasn't called!' );
end;
ShowMessage( 'but Script Continues to Execute' );

```

2.13 Arrays

An **array** is a collection of values of the same data type that you can access by **index** number. For example, you can create an array that can hold 100 integer values, or 25 string values. You access the **elements** of an array by their index numbers. See the COM Support Chapter in the Wealth-Lab User Guide for a description and examples on COM Variant Arrays.

Declaring Arrays

Arrays are declared with a special form of the **var** statement. You provide the name of the array and the upper and lower bounds, which must be a literal integer or a declared constant that appears before the array declaration.

Example

```

{ Declare an array that can hold 100 integers }
var MyArray: array[1..100] of integer;

{ Declare an array that can hold 25 strings }
var BunchOfStrings: array[1..25] of string;

{ Use a constant to specify the upper bound }
const UB = 10;
var RootBeerFloats: array[0..UB] of float;

```

To declare a multi-dimensional array, simply append **of array** statements as shown in the following example.

Example

```

{ Each element of this multi-dimensional array are assigned a value
  equal to the product of its indices }
var i, j: integer;
var arMulti: array[1..10] of array[1..20] of float;

{ Fill the array }
for i := 1 to 10 do
  for j := 1 to 20 do
    begin
      arMulti[ i, j ] := i * j;
      Print( '[ ' + IntToStr( i )
        + ', ' + IntToStr( j ) + ' ] = '
        + FloatToStr( arMulti[ i, j ] ) );
    end;
  end;
end;

```


Accessing Array Elements

Use the index number to access individual elements of the array. You can read the values from an array, and set values to an array.

Example

```
var MyArray: array[1..100] of integer;
var number: integer;

number := MyArray[1] + MyArray[2];
MyArray[3] := number;
```

Looping through Array Elements

The various Looping Statements in WealthScript provide a powerful way to work with array elements.

Example

```
var MyArray: array[1..100] of integer;
var i, number, TheSum: integer;
TheSum := 0;
for i := 1 to 100 do
    TheSum := TheSum + MyArray[i];
```

Creating Synchronized Arrays

The number of elements of an array must be specified in the **var** statement that declares an array, and it must be a constant value. However, you can create a special type of array that automatically contains the same number of elements as bars in your chart. Just specify zero as both the upper and lower bounds of the array.

Example

```
{ Create an array synchronized to the number of bars in the chart }
var SmoothedAverage: array[0..0] of float;
var i: integer;
SmoothedAverage[0] := ( PriceHigh(0) + PriceLow(0) ) / 2;
for i := 1 to BarCount - 1 do
    SmoothedAverage[i] := ( ( PriceHigh(i) + PriceLow(i) ) / 2 ) +
    SmoothedAverage[i - 1] / 2;
```

It can be useful to declare a synchronized array of an [enumerated type](#)¹³ to hold state data for a particular bar in the chart.

Example

```
var Bar, AvgHi: integer;
type HiCond = (HiRise, HiFlat, HiFall);
var HiMktCond: array[0..0] of HiCond;

AvgHi := SMASeries( #High, 20 );
for Bar := 20 to BarCount - 1 do
    if @AvgHi[bar] > @AvgHi[bar-1] then
        HiMktCond[bar] := HiRise
    else if @AvgHi[bar] = @AvgHi[bar-1] then
        HiMktCond[bar] := HiFlat
    else
```

```
HiMktCond[bar] := HiFall;
```

Passing Arrays as Parameters to Functions and Procedures

You can pass an array as a parameter to a Function or Procedure. To do this you must use the **type** statement (normally used when creating new [Object](#) types) that describes the type and bounds of the array. You then declare the array using this type. You use the same type within the function or procedure parameter list.

Note: Types must be defined outside of a *type* declaration. In other words, you cannot define a type within another *type*.

Ref: <http://www.wealth-lab.com/cgi-bin/WealthLab.DLL/topic?id=4691>

Example

```
type TMyArray = array[0..0] of float;

{ Note AnArray is passed by reference }
procedure ZeroArray( var AnArray: TMyArray );
begin
    var i: integer;
    for i := 0 to BarCount - 1 do
        AnArray[i] := 0;
    end;

var MyArray: TMyArray;
var Bar: integer;

{ Put values in the last 10 elements of MyArray }
for Bar := BarCount - 10 to BarCount - 1 do
begin
    MyArray[Bar] := Bar;
    Print( 'ZeroArray[' + IntToStr( Bar ) + '] = ' +
        FormatFloat( '#.0', MyArray[Bar] ) );
end;
ZeroArray( MyArray );
{ MyArray, which has the same number of elements as the chart,
  now has all of those elements initialized to zero. }
Print( ' ' );
for Bar := BarCount - 10 to BarCount - 1 do
    Print( 'ZeroArray[' + IntToStr( Bar ) + '] = ' +
        FormatFloat( '#.0', MyArray[Bar] ) );
```

Note: If the **var** statement were not included in the argument list of the procedure, the array would be passed *by value*. In other words, a copy of the array is made available to the procedure to which it is passed. Therefore, changes made to the copy of the array (AnArray) will not affect the original array (MyArray) in the calling procedure.

3 Working with Price Series

3.1 Introduction to Price Series

In every ChartScript you will in some way change, manipulate, test, etc. a Price Series. We recommend that you take some time to fully understand the concepts explained in this chapter.

[What is a Price Series?](#)⁴⁸

This special internal data structure, which always has the same number of elements as bars in the chart, provides quick access to your data and an easy way to refer to it.

[Handles to Price Series](#)⁴⁹

To use a Price Series you just need to get a handle on it. See how in the topics in this chapter.

[Creating Your Own Price Series](#)⁵⁴

Sometimes you'll want to generate a brand new series one element at a time.

[Accessing a Single Value of a Price Series](#)⁵⁵

These functions return individual values of standard price series.

[Using @ Syntax to Access Values from a Price Series](#)⁵⁷

Are you tired of typing the Set/GetSeriesValue function syntax? Use the @-symbol shorthand notation instead.

[Price Series Frequently-Asked Questions](#)⁶¹

... in case you still have some.

3.2 What is a Price Series?

A **Price Series** is a special type of internal data structure in WealthScript. Simply put, a Price Series is a sequence of values, one value for each bar in the chart. Consequently, you can think of a Price Series as a 1-dimensional array of values in which the index of the array are the bar numbers of the chart.

In ChartScripts, you will refer to Price Series using **handles**. A handle is an integer value used to reference a Price Series in memory. You don't have to worry about the values of handles (Wealth-Lab takes care of these details for you), rather, using WealthScript functions you will obtain and assign handles to your own well-named integer variables to remind you of the contents of series to which the handle refers, like "My15PeriodAvgSeries."

To learn more about handles, read the topics in the next section, Handles to Price Series.

Characteristics of all Price Series:

- A Price Series is a series of data values of type **float**. Each value is single precision, which has 7 to 8 significant digits.
- A Price Series always contains the same number of values as bars in your chart.
- A Price Series has a constant value, called a handle (of type **integer**) which you use to make reference to the *complete* series of values.

See Also:

[Standard Price Series and Their Constants](#) ⁵⁰
[Functions that Return a Price Series Handle](#) ⁵¹

3.3 Handles to Price Series

3.3.1 Overview

In Wealth-Lab Developer 4.0, the proper use of Price Series and their handles is essential to obtaining accurate back-testing results. Once you have mastered these concepts, you will be well on your way to understanding how to create trading systems as simple or complex as you like.

[Standard Price Series and Their Constants](#) ⁵⁰

Several pre-defined named constants provide access to Price Series that you will continually use in your ChartScripts. Find out which ones they are.

[Functions that Return a Price Series Handle](#) ⁵¹

WealthScript contains a great number of functions that return handles of new Price Series. With these functions, you can create indicators to your specification or even perform operations across complete Price Series with just one statement! However, to use a new Price Series, you'll have to designate your own handles.

[Functions that Accept a Price Series Handle](#) ⁵²

When a WealthScript function calls for a *Series* as an integer argument, you must insert a valid Price Series handle. By doing this, you're making reference to the Price Series on which the function will operate.

Checklist for Creating Price Series Handles

Still have doubts? Follow this handy checklist for using Price Series handles in ChartScripts. Note that the following is not necessary if you're going to use a Standard Price Series constant, such as **#Close** or **#Volume**.

- Step 1.** Declare an integer variable that you will use as the handle for your new Price Series.
 - Step 2.** Assign a function that returns a Price Series handle, an integer, to your variable. If this function is not **CreateSeries**, you're finished!
 - Step 3.** If you used **CreateSeries** in step 2, then you should use the **SetSeriesValue** function to assign values to your new series. If you don't do this, the series will hold the value zero (0.0) for every element.
-

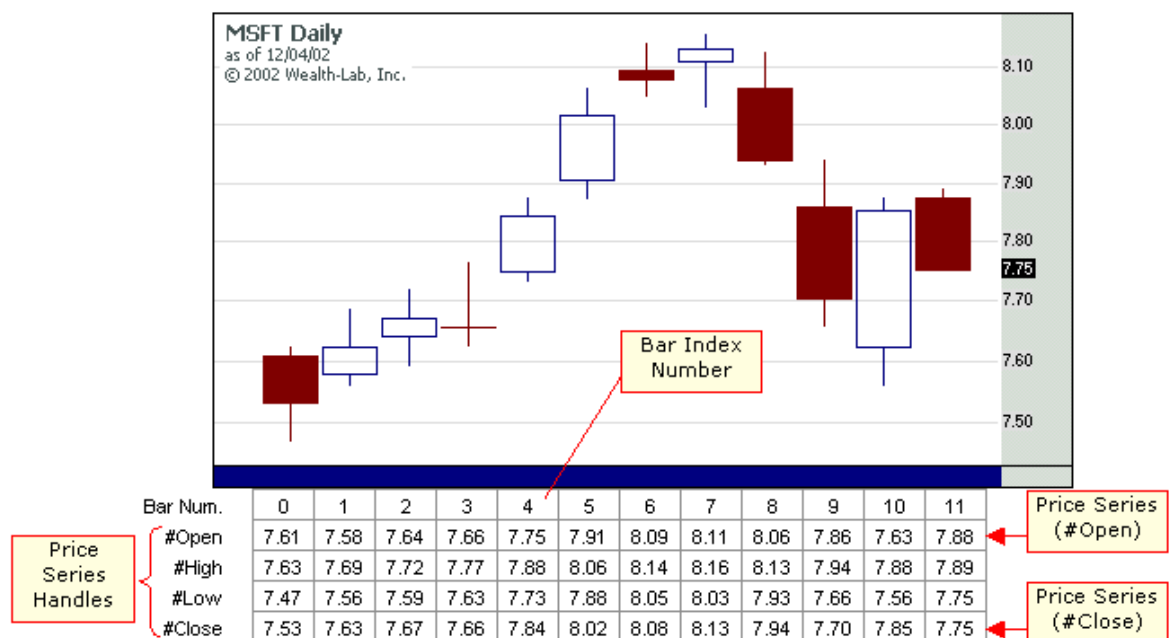
3.3.2 Standard Price Series and Their Constants

Some Price Series are ubiquitous, such as the series of *OHLC* values. These are referred to as the *Standard Price Series*. For these series, and for a few others described below, Wealth-Lab has established "constant handles" that you can use to rapidly access those series. Four of these pre-defined handles are **#Open**, **#High**, **#Low**, and **#Close**.

The handle **#Open** always refers to the series of all the *opening prices* of a chart's primary data source. Likewise, **#High** refers to the series of all the *high prices* of a chart's data source, and so on.

Remember, these handles make reference to the *entire series* and not just one particular value in the series. To find out how to obtain a single value from a Price Series, see the topic, [Accessing a Single Value of a Price Series](#)^[55].

Let's visualize what we have described by considering the following illustration.



Assuming that this chart's data source has no other values to the left or right (early or later in time, respectively), we can observe four of the Standard Price Series, each having 12 bars, which are numbered 0 to 11. Later, we'll show you how to [access a single value of a Price Series](#)^[55].

Completing the list of Standard Price Series we have **#Volume**, **#OpenInterest**, **#Average** and **#AverageC**. The last two, which are handles to the average of other Standard Price Series, merit a definition:

#Average

Returns the complete Average Price Series (all bars) as defined by the equation:

$$(\text{High} + \text{Low}) / 2$$

#AverageC

Returns the complete Average Price Series weighted by closing prices as defined by the equation:

$$(\text{High} + \text{Low} + \text{Close}) / 3$$

3.3.3 Functions that Return a Price Series Handle

It's impossible to show examples using Price Series without describing WealthScript functions that accept, and most often, return handles to Price Series. WealthScript has many such functions, which generally fall into two groups - Indicator Series Functions and Price Series Operator Functions.

Price Series Indicator Functions

By using WealthScript Indicator Series Functions, you will discover how easy it is to create a new Price Series of averages, oscillators, statistical measurements, etc. Let's demonstrate this by means of an example in which we create a new Price Series of the 5-period Weighted Moving Average of closing prices.

Example

```
{ Create a 5-period Weighted Moving Average Series
  from the series of closing prices }
var serWMA5: integer;
serWMA5 := WMASeries( #Close, 5 );

// Plot the new series
PlotSeries( serWMA5, 0, #Blue, #Thick );
```

What's going on here? After declaring one integer variable, serWMA5, to hold the handle of the new Price Series (the WMA series), we've created the new series using just one statement. The **WMASeries** statement returns the handle of the complete WMA series, which is assigned to serWMA5. Notice that one of the **WMASeries** arguments was the pre-defined handle of the Standard Price Series of closing values, **#Close**. Take a closer look at the example with typical values:

Bar Number	0	1	2	3	4	5	6	7	8	9	10	...
#Close	22.58	22.55	19.79	20.22	18.50	18.06	18.60	17.04	17.40	16.41	16.26	...
serWMA5	0.00	0.00	0.00	0.00	20.03	19.14	18.73	18.07	17.71	17.20	16.79	...

You'll notice that the first four values of the new series are zeroes. This is because the 5-period WMA series cannot be calculated until the fifth sample of data (Bar Number 4), therefore the initial samples are filled with zeroes. This is typical with indicators that require *seed data*, such as with any moving average function.

See Also:

The Technical Indicator Functions category of the WealthScript Function Reference [\[5\]](#) contains detailed information and examples of all the intrinsic indicator functions in Wealth-Lab Developer 4.0.

Price Series Operator Functions

Using Price Series Operator Functions you can perform operations on an existing Price Series and store the result in another. For example, you may want to rescale an entire series to normalize all of its values. The example below shows how you can create a new Price Series by dividing each value in the **#Close** series by a single value.

Example

```
{ Divide every bar's closing value by the value in the variable dvsr }  
var dvsr, serDivClose: integer;  
  
dvsr := 2;  
serDivClose := DivideSeriesValue( #Close, dvsr );  
PlotSeries( serDivClose, 0, #Red, #Thin );
```

Finally, here's an example of what **NOT** to do:

Example

```
{ Don't use handles in ordinary math operations! }  
var dvsr, serDivClose: integer;  
  
dvsr := 2;  
serDivClose := #Close / dvsr; // THIS IS A LOGIC ERROR!  
PlotSeries( serDivClose, 0, #Red, #Thin );
```

You might think this would accomplish the same thing as in the preceding example. Instead, the error "Not a valid Price Series" occurs when you try to refer to the new Price Series in the `PlotSeries` statement. This is because ordinary division does not create a new Price Series. You must use a WealthScript function that returns a Price Series integer handle as in the previous example with `DivideSeriesValue`.

See Also:

The WealthScript Function Reference [\[5\]](#) has detailed documentation for all Price Series functions.

3.3.4 Functions that Accept a Price Series Handle

WealthScript contains numerous built-in functions that provide access to common technical-analysis indicators. All of these functions, which may be applied to *any* Price Series, are well documented in the WealthScript Function Reference [\[5\]](#), but let's pick a familiar one to get a flavor for their use.

For example, you may like to obtain the simple moving average of a Price Series at a specific bar. Simple enough (no pun intended), you would choose the **SMA** function to return a Simple Moving Average. Here's the syntax for the SMA function that returns a single float value:

SMA(Bar, Series, Period);

Item	Description
<i>Bar</i>	Integer. Numeric expression representing the Bar Number of the chart for which you want to obtain the moving average value.
<i>Series</i>	Integer. The handle of a Price Series on which to base the moving average. You may use any of the following: <ul style="list-style-type: none"> • a Standard Price Series handle such as #Open, #Close, #Volume, etc • an integer variable to which a handle was previously assigned; from a WealthScript function for example • the complete syntax of any WealthScript function that returns a Price Series [integer handle]
<i>Period</i>	Integer. Numeric expression that is the <i>Period</i> of the moving average.

Recalling that functions return values, the following example shows how to get the 10-period SMA at the 51st bar and assign it to a variable of type float named *mySma*:

Example

```
var mySma: float;
mySma := SMA( 50, #Close, 10 );
ShowMessage( 'mySma = ' + FormatFloat('#0.00', mySma) );
```

You may be asking, "Why is the argument 50 and not 51?" In programming, arrays are typically, but not always, 0-based. Wealth-Lab internally uses 0-based arrays for Price Series, consequently the first bar number of a chart is actually 0, the second bar number is 1, and so on. This is an important tedious detail, but, in general you don't have to be conscious of it.

In the previous example and in those of the topic [Functions that Return a Price Series Handle](#)^[51], only the series of price closes, **#Close**, has been used. Note however, that *any* valid Price Series handle may be used for the *Series* argument in a WealthScript function. In the example below, we use the **SMASeries** function to return the handle of the complete 15-period SMA Price Series.

Example

```
{ Divide every bar's closing value by the value in the variable fd
  then take its 15-period Simple Moving Average }
var fd : float;
var serDivClose, serSMA: integer;

fd := 2.0;
serDivClose := DivideSeriesValue( #Close, fd );
serSMA := SMASeries( serDivClose, 15 );
// Plot the new series
PlotSeries( serSMA, 0, #Red, #Thin );
```

In a subtle way, another important aspect of WealthScript functions has just been introduced in the above examples. Nearly all indicators functions have two associated forms: one that returns the value of the indicator on a specific bar, like the **SMA** function, and, another that returns the complete Price Series of the indicator, as in the **SMASeries** function. This is explained in greater detail in the topic, [Working with Technical Indicator Functions](#)^[83].

3.4 Creating Your Own Price Series

You can create a new, blank Price Series and plug whatever values you need into it. You may be wondering why you would bother storing calculated values into a Price Series. Generally speaking, if you cannot find a WealthScript function or combination of functions that generates the series or indicator you're looking for, you'll have to resort to creating the series yourself.

If this sounds difficult, it's not. Simply use the **CreateSeries** function to *prepare* a new Price Series and then the **SetSeriesValue** to place values into it. Later, use the **GetSeriesValue** function to access the values within your newly created Price Series. The latter of these functions is covered in its own topic, [Accessing a Single Value of a Price Series](#)⁵⁵. For more information, refer to the [Checklist for Creating Price Series Handles](#)⁴⁹.

Here's the syntax of the **SetSeriesValue** procedure:

Syntax

SetSeriesValue(*Bar*, *Series*, *Value*) ;

Item	Description
<i>Bar</i>	Integer. Numeric expression representing the Bar Number of the chart for which <i>Value</i> is to be associated.
<i>Series</i>	Integer. The handle of a Price Series.
<i>Value</i>	Float. Numeric expression of the data to be assigned to <i>Bar</i> in <i>Series</i> .

In the example below the functions **PriceHigh** and **PriceLow** are used to retrieve values from the specified bar of the **#High** and **#Low** Standard Price Series. Using these values we can create a new Price Series that is equal to the current bar's midpoint between high and low. (You may recall this as being the **#Average** Standard Price Series, but we'll create our own new series for the sake of example.) As we loop through each of the chart's bars, a new value is calculated and inserted into the new series using the **SetSeriesValue** function.

Example

```
var MIDPOINT, BAR: integer;
var fValue: float;

{ The CreateSeries function creates and assigns the handle of a
  new Price Series that is initially filled with zeroes }
MidPoint := CreateSeries;
for Bar := 0 to BarCount - 1 do
begin
  fValue := PriceLow( Bar ) + ( PriceHigh( Bar ) - PriceLow( Bar ) ) /
  2 ;
  SetSeriesValue( Bar, MidPoint, fValue );
end;
// Plot the new series
PlotSeries( MidPoint, 0, #Blue, #Thin );
```

BarCount

In order to work with Price Series properly, you first need to know how many bars of data you have available in the chart. Use the **BarCount** function to return this information.

Example

```
var BarsAvailable: Integer;
BarsAvailable := BarCount;
```

The next example cycles through the chart data and accumulates the closing prices for "up" bars in one variable, and the closing prices for "down" bars in another variable, and then divides the result.

Example

```
var SUMUP, SUMDOWN, SUMUPDOWN: float;
var BAR: integer;
SumUp := 0;
SumDown := 0;
for Bar := 1 to BarCount - 1 do
begin
  if ( PriceClose( Bar ) >= PriceClose( Bar - 1 ) ) then
    SumUp := SumUp + PriceClose( Bar )
  else
    SumDown := SumDown + PriceClose( Bar );
end;
SumUpDown := SumUp / SumDown;
```

Note that in the example, the loop ends at **BarCount - 1**. This is because the first bar of a chart has an index number of 0, the second bar has index number 1, and so on. Consequently, you must terminate your loops at BarCount - 1, the last bar, or earlier.

You may also have noticed that the loop started at 1 instead of 0. This was necessary due to the argument of the second **PriceClose** statement: (Bar - 1). If Bar were allowed to be zero, the argument would have evaluated to -1, which does not refer to any bar of the chart, therefore a run-time error would have resulted.

See Also:

[Using "@" Syntax to Access Values from a Price Series](#)^[57]

3.5 Accessing a Single Value of a Price Series

Single Values of Standard Price Series

WealthScript has easy-to-remember functions that return the core price and volume values from your ChartScript data source. These functions are **PriceOpen**, **PriceHigh**, **PriceLow**, **PriceClose**, **Volume**, **OpenInterest**, **PriceAverage**, and **PriceAverageC**. They return a *single value at a specific bar* from the [Standard Price Series](#)^[50] that they describe, consequently, it is not necessary to specify the Price Series as a function argument.

The general syntax for this group of Data Access functions, all of which return a number of type **float**, is shown below.

Syntax (general)

```
functionname( Bar ) ;
```

Item	Description
functionname	Any one of the data access function names: PriceOpen, PriceHigh, PriceLow, PriceClose, Volume, OpenInterest, PriceAverage, or PriceAverageC.
<i>Bar</i>	Integer. Numeric expression representing the bar of the chart from which data is to be retrieved.

GetSeriesValue Function

To obtain a single price value from *any* series, use can use the **GetSeriesValue** function. Generally speaking, however, you will this function to obtain values from Price Series created using the **CreateSeries** function. As we have just seen, shorthand methods exist to retrieve [single values from Standard Price Series](#)^[55]. Later, you'll discover that Technical Indicators Functions also have a [more-intuitive method](#)^[83] to obtain their value at a specific bar.

GetSeriesValue returns a **float** value of the series at the *Bar* number.

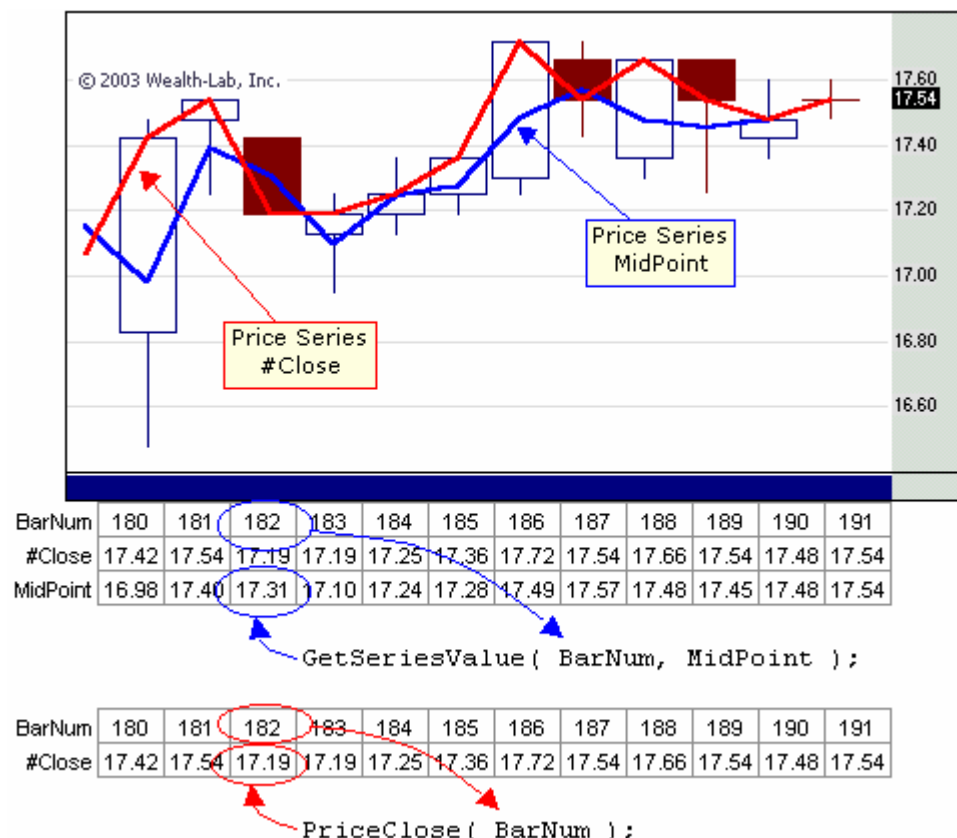
Syntax

```
GetSeriesValue( Bar, Series ) ;
```

Item	Description
<i>Bar</i>	Integer. Numeric expression representing the bar of the chart from which data is to be retrieved.
<i>Series</i>	Integer. The handle of a Price Series.

Note: It's perfectly legal to use **GetSeriesValue** to retrieve, for example, the closing price of *Bar* by passing **#Close** as the Price Series handle. However, **PriceClose** is a shorthand statement that *always* refers to the **#Close** Price Series and therefore gives your code better readability.

In the illustrations below, **GetSeriesValue** is used to obtain the values from the Price Series *MidPoint*. *MidPoint* is the handle to a Price Series we created in the example for [Creating Your Own Price Series](#)^[54].



The arrow diagram indicates that if the integer expression *BarNum* evaluates to 182, the *GetSeriesValue* function will return the value of 17.31 when *MidPoint* is specified. Likewise, the *PriceClose* statement evaluates to 17.19, which is the closing price at bar number 182.

3.6 Using @ Syntax to Access Values from a Price Series

A simpler method is available to access values in a Price Series. If you precede the Price Series handle variable with a "@", you can access the values in the Price Series as if it were an array. You can read and write values to a Price Series using this syntax. This eliminates (*see Note) the need to code **GetSeriesValue** and **SetSeriesValue**, and can substantially reduce the verbiage in a script's code.

Example

```
var Series: integer;
var x: float;
Series := CreateSeries;

SetSeriesValue( 0, Series, 123.45 );
{ becomes }
@Series[0] := 123.45;

x := GetSeriesValue( 0, Series )
{ becomes }
x := @Series[0];
```

Note: The @ syntax is not compatible with Price Series whose handles are stored in a declared array as the following example demonstrates. In this case, you *must*

use the **GetSeriesValue** or **SetSeriesValue** WealthScript functions, as required.

Example

```
var Bar, i: integer;
var h: array[0..1] of integer;

{ Create 2 price series and store their handles in the array }
h[0] := WMASeries( #Close, 5 );
h[1] := WMASeries( #Close, 20 );

{ Retrieve the value of each series on the last bar }
Bar := BarCount - 1;
for i := 0 to 1 do
    Print( FloatToStr( GetSeriesValue( Bar, h[i] ) ) );

{ This is not valid! }
for i := 0 to 1 do
    Print( FloatToStr( @h[i][Bar] ) );
```

3.7 Series Math

3.7.1 Practice

Let's take some time to drive home some points that some users seem to have trouble grasping (especially those coming from other technical analysis platforms). Please take the time to do these simple exercises and check your answers in the next topic.

Exercise 1

Imagine that you want to create a Price Series that holds the *change* in **closing price** relative to the first bar of the chart. To do this, you need to obtain the value of the first bar of the chart and subtract it from the closing prices from all of the remaining bars. How? Try to plot the new resultant series in a new pane.

Exercise 2

Similar to Exercise 1, create a Price Series that holds the *percentage change* in **closing price** relative to the first bar of the chart. Plot the resultant series. Use the following formula:

$$\text{PctChange} = 100 * (\text{CurrentPrice} / \text{ValueOnFirstBar} - 1)$$

Exercise 3

In both of the preceding examples, we performed math operations on Price Series by subtracting, multiplying, and dividing by a single constant value. Now let's use two different Price Series as the operands by finding the average closing price between the #High and #Low series. Use the following formula and plot the resultant series:

$$\text{AvgClosingPrice} = (\text{High} + \text{Low} + \text{Close}) / 3$$

Exercise 4

Create and plot the difference of the current closing price minus the closing price from 2 bars ago.

Exercise 5

After reviewing the answers to the above exercises, explain the main difference in the techniques used between the "Answer A's" and the "Answer B's".

3.7.2 Answers

Please try performing the exercises on your own first before peeking at the answers.

It's better to make the mistakes now!

Exercise 1

```
{ Answer A }
var DiffSer1: integer;
DiffSer1 := SubtractSeriesValue( #Close, PriceClose( 0 ) );

{ Answer B }
var Bar, DiffSer2: integer;
DiffSer2 := CreateSeries;
for Bar := 0 to BarCount - 1 do
  @DiffSer2[Bar] := PriceClose( Bar ) - PriceClose( 0 );

{ Create a pane to plot the new series }
var DiffPane: integer = CreatePane( 100, true, true );
PlotSeriesLabel( DiffSer1, DiffPane, #Blue, #Thick, 'Difference from
Bar 0' );
PlotSeriesLabel( DiffSer2, DiffPane, #Red, #Histogram, 'Difference from
Bar 0' );
```

As you can see there are at least 2 correct answers. Notice though, that Answer A utilizes a special WealthScript function, `SubtractSeriesValue`, to subtract a *single value* from each element in the Price Series identified in its first parameter. This results in fewer statements and code that executes faster.

Exercise 2

```
{ Answer A }
var PctSer1, DivSer, DiffSer: integer;
DivSer := DivideSeriesValue( #Close, PriceClose( 0 ) );
DiffSer := SubtractSeriesValue( DivSer, 1 );
PctSer1 := MultiplySeriesValue( DiffSer, 100 );

{ Answer B }
var Bar, PctSer2: integer;
PctSer2 := CreateSeries;
for Bar := 0 to BarCount - 1 do
  @PctSer2[Bar] := 100 * ( PriceClose( Bar ) / PriceClose( 0 ) - 1 );

{ Create a pane to plot the new series }
var PctPane: integer = CreatePane( 100, true, true );
PlotSeriesLabel( PctSer1, PctPane, #Blue, #Thick, 'Pct Change from Bar
0' );
PlotSeriesLabel( PctSer2, PctPane, #Red, #Histogram, 'Pct Change from
Bar 0' );
```

Again, we can perform the same calculation in at least two different ways - the choice is yours! Use whichever makes the most sense to you. Note that the solution in Answer A can be also written without the use of the interim variables. Below we use a block-formatting technique to help show the relationship of the parameters to their functions.

```
{ Answer A2 }
var PctSer1: integer;
PctSer1 := MultiplySeriesValue(
    SubtractSeriesValue(
        DivideSeriesValue( #Close, PriceClose( 0 ) ),
        1 ),
    100 );
```

Exercise 3

```
{ Answer A1 }
PlotSeries( #AverageC, 0, #Gray, #Thick );

{ Answer A2 }
var AvgCSer1: integer;
AvgCSer1 := DivideSeriesValue(
    AddSeries( AddSeries( #High, #Low ), #Close ),
    3 );

{ Answer B }
var Bar, AvgCSer2: integer;
AvgCSer2 := CreateSeries;
for Bar := 0 to BarCount - 1 do
    @AvgCSer2[Bar] := ( PriceHigh( Bar ) + PriceLow( Bar ) + PriceClose(
        Bar ) ) / 3;

{ Plot the new series in the Price Pane, 0 }
PlotSeriesLabel( AvgCSer1, 0, #Fuchsia, #Dotted, 'Avg Closing Price A1' );
PlotSeriesLabel( AvgCSer2, 0, #Blue, #Thin, 'Avg Closing Price B' );
```

Did you recognize this as the formula for the **#AverageC** [Standard Price Series](#)⁵⁰?

Exercise 4

```
{ Answer A1 }
var DiffSer1: integer;
DiffSer1 := MomentumSeries( #Close, 2 );

{ Answer A2 }
var DiffSer2: integer;
DiffSer2 := SubtractSeries( #Close, OffsetSeries( #Close, -2 ) );

{ Answer B }
var Bar, DiffSer3: integer;
DiffSer3 := CreateSeries;
for Bar := 2 to BarCount - 1 do
    @DiffSer3[Bar] := PriceClose( Bar ) - PriceClose( Bar - 2 );

{ Plot the new series in a new Pane }
var DiffPane: integer = CreatePane( 100, true, true );
PlotSeriesLabel( DiffSer1, DiffPane, #Gray, #ThickHist, 'Difference A1' );
PlotSeriesLabel( DiffSer2, DiffPane, #Fuchsia, #Histogram, 'Difference
```

```
A2' );
PlotSeriesLabel( DiffSer3, DiffPane, #Blue, #Thick, 'Difference B' );
```

- A1:** The WealthScript function `MomentumSeries` provides the easiest solution. Many of the WealthScript Indicator functions provide ready-made solutions for the most common operations, and only through experience can you get familiar with them.
- A2:** Here we *delay* the `#Close` series by 2 bars using `OffsetSeries`. After that, we simply perform a series difference operation on the original `#Close` series and its offset.
- B:** You can always fall back to doing the calculations one bar at a time and filling the Price Series created by you. Note that the loop must start at Bar #2 here. Why? Try putting 0 or 1 for the starting the loop index and execute the script again (F5). What happens?

Exercise 5

Answer A's technique:

1. Declare an *integer* variable to hold a *Price Series handle*, a reference.
2. Use the result of a WealthScript `*Series` function to assign a series to the variable.

Answer B's technique:

1. Declare an *integer* variable to hold a *Price Series handle*, a reference.
2. Use `CreateSeries` to assign a new blank price series (filled with zeroes) to the variable.
3. Loop over bars in the chart to fill the new series with values.

3.8 Price Series FAQs

How do I get the data from the Open, High, Low, Close, or Volume of a bar?

The preferred method is to use the WealthScript functions specifically designed for this purpose: `PriceOpen(Bar)`, `PriceHigh(Bar)`, `PriceLow(Bar)`, `PriceClose(Bar)`, `Volume(Bar)`, `OpenInterest(Bar)`, `Average(Bar)`, `AverageC(Bar)`, where *Bar* is the Bar Number of the bar for which you want the data.

Example

```
{ Get the opening price of the last bar in the chart }
var fOpen: float;
var Bar: integer;
Bar := BarCount - 1;
fOpen := PriceOpen(Bar);
```

Equally effective, you may use the general syntax for getting data from any Price Series, `GetSeriesValue` or its [shorthand @ syntax](#)^[57].

Example

```

{ Get the opening price of the last bar in the chart }
var fOpen: float;
var Bar: integer;
Bar := BarCount - 1;
fOpen := GetSeriesValue(Bar, #Open);

{ or use the shorthand @ syntax }
fOpen := @#Open[Bar];

```

How do I get an indicator's value at a specific bar?

Each indicator has a syntax form that is specifically designed for this purpose. If you know in advance which indicator you will be using, like **SMA**, **RSI**, **StochK**, etc. you can use its syntax. For more information see [Accessing Indicator Values](#)^[83] in the chapter [Working with Technical Indicator Functions](#)^[83].

Example

```

{ Find the 10-period Simple Moving Avg at Bar Number 50 }
var mySma: float;
mySma := SMA( 50, #Close, 10 );
ShowMessage( 'mySma at Bar Number 50 = ' + FormatFloat('#0.00',
mySma) );

```

Sometimes you will not know in advance which Price Series you will be using. This may sound strange, but it's the basis of the manner in which re-useable functions and procedures work. In this case, use the general syntax for getting data from any Price Series, `GetSeriesValue` or its [shorthand @ syntax](#)^[57].

Example

```

{ Returns the number of bars ending with EndBar since
  Series2 crossed over Series1. }
function BarsSinceCrossOver(EndBar, Series1, Series2: integer ):
integer;
begin
  var i, CntBar: integer;

  CntBar := 0;
  for i:= EndBar downto 0 do
    if GetSeriesValue(i, Series2) < GetSeriesValue(i, Series1) then
      break
    else
      CntBar := CntBar + 1;

  Result := CntBar;
end;

{ test the function by passing the handles from two different Price
Series }
var b, MySMAHandle, Bar: integer;

MySMAHandle := SMASeries( #Close, 20 );
PlotSeries(MySMAHandle, 0, #blue, #thin );
Bar := BarCount - 1;

{ Pass MySMAHandle as Series1 and #High as Series2 to the function }
b := BarsSinceCrossOver( Bar, MySMAHandle, #High );
Print( 'The number of bars since cross over is: ' + IntToStr(b) );

```

```
{ Draw a circle on the bar just prior to crossover }  
DrawCircle( 8, 0, Bar - b, PriceHigh(Bar - b), #red, #thick );
```

How can I access a Price Series from a symbol other than the one in my ChartScript?

You can obtain information from Price Series that are not part of the primary ChartScript Standard Price Series by using the [GetExternalSeries](#) function. See its entry in the WealthScript Function Reference^[5] for more information.

See Also: [Plotting Indicators Based on Other Symbols](#)^[67]

4 Painting the Chart

4.1 Overview

WealthScript provides a set of functions to control how your chart information is displayed. You can plot indicators, create new chart panes, add text, annotations, and drawing objects, or even draw bitmap images. Whenever you need to programmatically perform display tasks in the Chart window, open the Cosmetic Chart category to find the function that serves your purpose.

To view a chart, open a ChartScript by selecting File/New ChartScript. In the ChartScript window, the "Chart" tab is selected (by default) to view the chart of the item selected in the DataSource Tree.

[Chart Panes](#)^[65]

It is not essential that your code displays additional information on the chart. A trading system will function the same if you choose *not* to draw lines, add annotations, or plot indicators on the chart. However doing so can help you troubleshoot your scripts, give you visual affirmation that your system is functioning as designed, or even provide insight for improving methods.

[Creating New Panes](#)^[66]

The default panes for price and volume is just one possibility. See how to make new panes to plot additional indicators.

[Plotting an Indicator in a Pane](#)^[67]

Tell Wealth-Lab where and how you want to plot your Price Series by specifying the pane's index and drawing style. Also plot indicators based on symbols *other than* your primary symbol.

[Plotting Multiple Symbols](#)^[68]

Instead of drawing just a single #Close Price Series of another symbol, display all of the available information in full OHLC bar or candlestick representations.

[Specifying Colors](#)^[69]

Give life to your charts by adding color to bars, text, indicators, backgrounds, and more!

[Drawing Text in a Pane](#)^[70]

Did you forget what that blue line on the chart represents? Leave no doubt by adding colorful text legends and other useful annotations.

[Drawing Objects in a Pane](#)^[70]

Make your charts come to life by programmatically drawing objects such as lines, rectangles, and ellipses. You'll be amazed with the way some users are able to visually express even esoteric concepts in Wealth-Lab Developer 4.0 - like probability distributions and frequency spectrums!

4.2 Chart Panes

Panes refer to the subdivided areas of a chart in which price, volume, and other data is presented. A basic chart contains two panes - one that displays price information and another that displays volume. Because it contains less information, the volume Pane is typically smaller than the price Pane.

Useful Chart Panes facts:

1. When it does not make sense to plot an indicator's Price Series in the price or volume panes, you can [create new panes](#)⁶⁵ to display the additional information.
2. Conversely, if the volume pane is not important to you, use the [HideVolume](#) function to make more room for other panes.
3. You can manually resize chart panes by dragging the line that divides two adjacent panes.
4. Override an auto-scaling of a pane using the [SetPaneMinMax](#) WealthScript function.

Pane Indices

Each chart pane has an associated integer index that is used whenever a WealthScript function requires a *Pane* argument. Use this index to specify the pane that will receive the action of a drawing or plotting function. The two most common panes, the price and volume panes, always have indices of 0 and 1, respectively.

In previous chapters, we've already seen the [PlotSeries](#) statement used, so let's look at its syntax:

```
PlotSeries( Series, Pane, Color, Style);
```

With emphasis on the *Pane* argument, if we want to plot a series in the price pane we would pass the value 0 (or a numeric expression that evaluates to 0) as *Pane*.

Example

```
{ Connect the high values in the price pane by a thick blue line
  and envelope the volume histogram with a thick green line }
var PricePane: integer;
PricePane := 0;
{ The Pane argument can be a number or a numeric expression }
PlotSeries( #High, PricePane, #Blue, #Thick );
PlotSeries( #Volume, 1, #Green, #Thick );
```

4.3 Creating New Panes

You can create new chart panes to draw other indicators and Price Series using the [CreatePane](#) function. This WealthScript function returns the *integer* value of the new pane's index. Therefore you should assign [CreatePane](#) to an integer variable in order to prepare a new pane that you'll use later in a plotting or drawing function. Although the Price and Volume panes have indices of 0 and 1, respectively, you should **never** assume that newly-created panes will have indices of 2, 3, 4, etc. The function's syntax is:

```
CreatePane( Height, AbovePrices, ShowGrid ): integer;
```

Specify the *Height* as an integer in pixels of the new pane in the first parameter. A height of 100 is a good general-purpose value. Pass a boolean value as *AbovePrices* to indicate whether this pane will appear above (true) or below (false) the price pane. *ShowGrid* is another boolean used to control whether or not grid lines are drawn for the new pane. If you do not enable the standard grid lines you can call **DrawHorzLine** to manually draw your own lines at specific values on the pane.

Your new pane will automatically scale to the various Price Series that you plot within it. You can use the **SetPaneMinMax** function to override these calculated values.

In the following example a new Pane is created to draw RSI (Relative Strength Index). Since RSI can fluctuate between 0 and 100, the Pane is set to min/max at these values. The default grid is disabled, and we draw our own horizontal lines at the 70 and 30 levels.

Example

```
var MyRSI, PaneRSI: integer;
MyRSI := RSISeries( #Close, 14 );
PaneRSI := CreatePane( 100, true, false );
SetPaneMinMax( PaneRSI, 0, 100 );
PlotSeries( MyRSI, PaneRSI, #Navy, #Thick );
DrawHorzLine( 30, PaneRSI, #Silver, #Dotted );
DrawHorzLine( 70, PaneRSI, #Silver, #Dotted );
DrawHorzLine( 50, PaneRSI, #Gray, #Dotted );
DrawText( 'RSI 14 day', PaneRSI, 4, 4, #Navy, 8 );
```



Notice that each time a new pane is added, it is automatically separated from the others by a thin horizontal line. For aesthetic reasons you may wish to remove these lines. To do so, simply add the **HidePaneLines** function to your ChartScript code.

4.4 Plotting an Indicator in a Pane

As we've seen, the `PlotSeries` function plots the specified Price Series in the desired Pane. Since `PlotSeries` can accept any Price Series, you now see the value of being able to obtain the handle to an indicator, and to creating your own custom Price Series. For reference, the syntax is as follows:

PlotSeries(*Series, Pane, Color, Style*);

After specifying the Price *Series* and the *Pane*, you then pass the desired *Color* (using the color formulated as described in the topic [Specifying Colors](#)^[69]) and a drawing style. The drawing style can be one of the following constants:

<code>#Thin</code>	Normal Solid Line
<code>#Dotted</code>	Dotted Line
<code>#Thick</code>	Thick Solid Line
<code>#Histogram</code>	Histogram Format
<code>#ThickHist</code>	Thick Histogram
<code>#Dots</code>	Dots

Example

```
{ Plot a 30 day SMA in green and a 200 day SMA in red }
PlotSeries( SMASeries( #Close, 30 ), 0, #Green, #Thin );
PlotSeries( SMASeries( #Close, 200 ), 0, #Red, #Thin );
```

Tip: Pass -1 to the function `SetSeriesBarColor` to prevent drawing of an indicator for the specified bar(s). This may be useful during periods when an indicator's value is invalid or in transition.

Plotting Indicators Based on Other Symbols

After working with WealthScript a short time, you'll see that plotting a symbol's Price Series and indicators based on those Price Series is simply a matter of accessing their data, and more precisely, obtaining a handle to their data. Once you have a reference to the data series, you can use it for plotting, and more importantly for calculations that lead to trading decisions.

Either of the WealthScript functions, `GetExternalSeries` or `SetPrimarySeries`, can be used to access data for a symbol other than the one selected in a ChartScript's DataSource tree. Generally speaking, `SetPrimarySeries` is used when you want to be able to access all the [Standard Price Series](#)^[50] of another symbol, or perhaps to generate a trading signal on the selected symbol while looping through a WatchList. `GetExternalSeries` returns a handle to only a single named Series for another symbol, and is sufficient for most operations requiring the use of external data.

The following two examples, which highlight the use of these functions, produce identical results.

Example

```
{ Plot HPQ and its 50-day SMA along with my chart using
GetExternalSeries }
var HPQClose, HPQ50, HPQPane: integer;
HPQClose := GetExternalSeries( 'HPQ', #Close );
HPQ50 := SMASeries( HPQClose, 50 );
HPQPane := CreatePane( 150, true, true );
PlotSeries( HPQClose, HPQPane, #Blue, #Thick );
PlotSeries( HPQ50, HPQPane, #Black, #Dotted );
```

Example

```
{ Plot HPQ and its 50-day SMA along with my chart using
SetPrimarySeries }
var HPQClose, HPQ50, HPQPane: integer;
SetPrimarySeries( 'HPQ' );
HPQClose := #Close;
HPQ50 := SMASeries( HPQClose, 50 );
RestorePrimarySeries;
HPQPane := CreatePane( 150, true, true );
PlotSeries( HPQClose, HPQPane, #Blue, #Thick );
PlotSeries( HPQ50, HPQPane, #Black, #Dotted );
```

4.5 Plotting Multiple Symbols

It's easy to plot a full OHLC representation of a symbol other than the one selected in a ChartScript's DataSource tree. Using PlotSymbol allows you to create a reference plot of another symbol, however it does not provide access to its data. To access Price Series data of an external symbol, see the topic [Plotting Indicators Based on Other Symbols](#)⁶⁷.

Syntax

PlotSymbol(*Symbol*, *Pane*, *Color*, *Style*);

Example

```
var P: integer;
PlotSymbol( 'MSFT', 0, #Silver, #Candle );
P := CreatePane( 100, true, true );
PlotSymbol( 'IBM', P, #Blue, #OHLC );
```

Conversely, you may have data for four Price Series that are not associated with a symbol. These could be data that you have created based on calculations from an indicator, for example. To combine these Price Series into a single OHLC representation, use the PlotSyntheticSymbol function.

Syntax

PlotSyntheticSymbol(*Symbol*, *OpenSeries*, *HighSeries*, *LowSeries*, *CloseSeries*, *Pane*, *Color*, *Style*)

Example

```

var RSIO, RSIH, RSIL, RSIC, RSIPANE: integer;
RSIO := RSISeries( #Open, 14 );
RSIH := RSISeries( #High, 14 );
RSIL := RSISeries( #Low, 14 );
RSIC := RSISeries( #Close, 14 );
RSIPane := CreatePane( 100, true, true );
PlotSyntheticSymbol( 'RSICandle', RSIO, RSIH, RSIL, RSIC, RSIPane,
#Red, #Candle );
DrawLabel( 'RSI Candles', RSIPane );

```

If you run the example above, notice that the candles will often times appear incorrect. This is because the RSI of low prices is not always less than the RSI to open, high and close, so the candle values do not always form into traditionally correct candles.

4.6 Specifying Colors

Many of the WealthScript Chart functions require parameter types that describe a *Color* value. WealthScript uses a simple mechanism to pass color information. Colors are broken down into different intensities for red, green and blue (RGB). Each intensity level can have a value between 0 (no intensity) to 9 (full intensity). A single 3 digit number expresses a complete color value.

900 = Bright Red	090 = Bright Green	009 = Bright Blue
550 = Olive	050 = Dark Green	444 = Dark Gray

You can also use the following special constants to specify color values:

#Black, #Maroon, #Green, #Olive, #Navy, #Purple, #Teal, #Gray, #Silver, #Red, #Lime, #Yellow, #Blue, #Fuchsia, #Aqua, #White, and finally #WinLoss (used primarily for [PerfScripts](#)^[106])

Finally, the three constants that follow refer to lightly-shaded colors often used with [SetBackgroundColor](#) to fill the chart's background or [SetPaneBackgroundColor](#) to color a pane's background. Nevertheless, these too can be passed as a value to any WealthScript function argument requiring *Color*.

#RedBkg, #BlueBkg, #GreenBkg

Coloring Bars

You can color individual bars using the [SetBarColor](#) function. Or, specify the colors used for up versus down bars with the [SetBarColors](#) function.

Example

```

{ Color all bars that reach a 20-day high green }
var Bar: integer;
for Bar := 20 to BarCount - 1 do
  if PriceHigh( Bar ) = Highest( Bar, #High, 20 ) then
    SetBarColor( Bar, #Green );

```


See Also:

SetColorScheme and SetSeriesBarColor in the WealthScript Function Reference.

4.7 Drawing Text in a Pane

Use the `DrawText` function to annotate a Chart Pane with text.

DrawText(*Value, Pane, X, Y, Color, Size*);

The first parameter is a **string** expression that contains the text to be drawn. The second parameter is the pane's index. The next two parameters specify the x, y position of the text, in pixels. A position of 0, 0 represents the top left of the Pane. The next parameter specifies the color to use. The last parameter indicates the font size. A value of 8 is standard size for drawing text.

Since you specify an absolute position within the pane, this function is most useful to create a legend for indicators and other Price Series.

Example

```
PlotSeries( #High, 0, #Blue, #Thick );
PlotSeries( #Low, 0, #Red, #Thick );

{ Make a legend for the plotted series }
DrawText( '#High Series', 0, 100, 4, #Blue, 8 );
DrawText( '#Low Series', 0, 100, 16, #Red, 8 );
```

WealthScript contains additional functions with which to annotate your charts with text using varying degrees of control. For more information, see their descriptions in the WealthScript Function Reference:

AnnotateBar, AnnotateChart, DrawLabel

4.8 Drawing Objects in a Pane

Wealth-Lab Developer 4.0 provides two toolbars, *Drawing* and *Drawing2*, which are filled with drawing objects to employ in *manually* annotate charts. The Drawing2 toolbar contains the Trendline Tool, whose values at any bar can be found by your WealthScript code even when drawn manually! For more information, see TrendLineValue in the Function Reference or visit the Chart Drawing Tools discussion in the User Guide.

If drawing chart objects manually doesn't appeal to you, then of course Wealth-Lab Developer 4.0 has a programmatic solution with WealthScript. Using functions from the Cosmetic Chart category, you have the ability to draw circles, diamonds, ellipses, lines, rectangles, and polygons (diamonds and triangles).

The example below provides a sampling of a few of the drawing objects available to you in ChartScripts. Run the example on a Daily DataSource by clicking on several different symbols in the ChartScript window.

Example

```
var BAR, Bar1, Bar2, Radius: integer;
```

```

var x1, x2, y1, y2, l1, l2, Price1, Price2: float;

{ Circle any 100 day Low }
for Bar := 100 to BarCount - 1 do
begin
    if PriceLow( Bar ) = Lowest( Bar, #Low, 100 ) then
        DrawCircle( 8, 0, Bar, PriceLow( Bar ), #Red, #Thick );
    end;

    { Draw a circle around an arbitrary point }
    Bar1 := BarCount - 40;
    Bar2 := BarCount - 20;
    SetBarColor( Bar1, #Blue );
    SetBarColor( Bar2, #Blue );

    y1 := PriceClose( Bar1 );
    y2 := PriceClose( Bar2 );
    DrawCircle2( Bar1, y1, Bar2, y2, 0, #Blue, #Thin );

    { Find the last 13% reversal peak and trough }
    Bar1 := TroughBar( BarCount - 1, #Low, 13 );
    Price1 := PriceLow( Bar1 );
    Bar2 := PeakBar( BarCount - 1, #High, 13 );
    Price2 := PriceHigh( Bar2 );

    { Draw a diagonal line and a rectangle between the previous peak and
    trough }
    DrawLine( Bar1, Price1, Bar2, Price2, 0, #Red, #Thick );
    DrawRectangle( Bar1, Price1, Bar2, Price2, 0, #Green, #Thin, #GreenBkg,
    True );

    { Draw ellipses to highlight peaks and troughs }
    DrawEllipse( Bar1 - 4, Price1 * 1.02, Bar1 + 4, Price1 * 0.98, 0,
    #RedBkg, #Thin, #RedBkg, true );
    DrawEllipse( Bar2 - 4, Price2 * 1.02, Bar2 + 4, Price2 * 0.98, 0,
    #BlueBkg, #Thin, #BlueBkg, true );

```

5 Writing Your Trading System Rules

5.1 Overview

A Trading System has steadfast rules that are carried out independently of what you *think* may be the outcome of the trade. Consequently, Trading System rules are the conditions under which you decide to enter and exit a trade.

Invariably, when deciding to program a new Trading System, you'll start by either verbalizing rules, drawing pictures, or writing *pseudo code* to collect your thoughts. (For this purpose, and also for documenting your scripts, you can make use of the ChartScript Description view.) Once you become proficient with WealthScript, you may find that it's just as easy to put down your thoughts in code directly into the ChartScript code editor!

[Scripting Trading Rules](#)^[72]

If you can imagine it, almost certainly you can test it in Wealth-Lab Developer 4.0. First, you'll have to translate your ideas into code. This chapter introduces the most important functions necessary to simulate real-life trading orders.

[Implementing Trading System Rules](#)^[78]

You can use whatever logic based on price, technical indicators, date information, or whatever else you can think of in your entry and exit rules. However, if you discover a trading system that achieves unimaginable returns, it's quite likely your code includes a *postdictive error*.

[Managing Multiple Positions](#)^[79]

Wealth-Lab Developer 4.0 assigns a number to each new Position entered. You can access this information with the functions presented in this topic, and in doing so, you'll be able to write trading systems that add Positions by averaging up or down. As always, it's your choice!

5.2 Scripting Trading Rules

5.2.1 Overview

The most powerful feature of WealthScript is the ability to embed Trading System rules in your ChartScripts. Whenever your ChartScript executes, Wealth-Lab Developer 4.0 displays all of the trades that your System generated in clear buy and sell arrows on the chart, provided that they are enabled. The ChartScript Performance Results view also lists the overall System performance, and the Trades view contains a detailed listing of all trades that were generated.

If you're just becoming familiar with WealthScript, the following topics provide an introduction to programming Trading Systems in a cumulative fashion.

[The Main Loop](#)^[73]

Each time a script is executed, generally it should start from the first bar at which all your indicators are valid and continue to the final bar in the chart.

[Triggering a Market Buy Order](#)^[74]

Learn how to simulate entering long positions with Market orders.

Triggering a Limit or Stop Buy Order^[75]

Limit orders offer more control over the entry price of a trade. In addition, stop orders permit you to enter a trade with confirmation that the market for the security is moving in favor of the trade's intended direction.

Checking for Open Positions^[75]

Your trading strategy will change once you have entered a position. In single-Position-only trading systems, you'll be looking for the exits if you're already holding a position.

Using Automated Stops^[76]

Let Wealth-Lab Developer 4.0 worry about getting you out of a trade. It's as simple as adding 2 statements.

Selling Short^[77]

Theoretically, it's true that selling short carries unlimited risk. However, just as you may test long strategies in Wealth-Lab Developer 4.0 without risking real cash, you may do the same with short or mixed strategies.

5.2.2 The Main Loop

Every Trading System must have a main loop that cycles through the data in your chart. This is accomplished with a typical "**for** loop", as shown below. Although the first bar number of a ChartScript's DataSource is 0, your for loop should start from *at least 1* to ensure that Market orders can be placed. In order to place a Market order the system needs a "basis price," which is the closing price of the previous bar.

Example

```
var BAR: integer;
{ ChartScript Main Loop }
for Bar := 30 to BarCount - 1 do
begin
{ Trading rules go here, and are executed once for every bar in the
chart }
end;
```

Here, the **for** loop starts at the 30th bar of data. You should set your main loop's starting point based on the technical indicators that you're using. For example, if you use a 30 bar SMA and a 14 bar RSI in your rules, choose the longer of the two indicators and set your starting **for** loop value to 30. The main loop typically ends at **BarCount - 1**, which is the number of the last bar in all Price Series. For more discussion, see the article on [Stability of Indicators](#) in the Wealth-Lab.com [Knowledge Base](#).

Finally, notice that no reference to time exists in controlling the loop, only consecutive bar numbers. Unless you use WealthScript Time Frame functions to specifically manipulate a DataSource's native time frame, you the same ChartScript will work equally well with Daily bars as for Intraday bars, for example.

5.2.3 Triggering a Market Buy Order

Use the **BuyAtMarket** function to trigger a market buy order.

BuyAtMarket(*Bar*, *SignalName*);

The first parameter contains the *Bar* number to execute the trade. WealthScript Trading System signals are usually triggered using the bar's *closing* value, consequently the order should be placed on the bar following the one that generated the signal (**Bar + 1**). When trading on a daily basis, this is analogous to receiving a signal based on the closing price of one day and buying at the open of the next day.

The second function parameter is a *SignalName* that you choose to identify the trading signal that triggered the trade. This data is later found in the Trades view and is useful in your system analysis if you have a system that uses more than one type of entry. If you don't need a signal name, just pass a blank string as shown in the Example.

Example

```
var BAR: integer;
{ This simple ChartScript buys when prices cross above a 14 day SMA }
for Bar := 15 to BarCount - 1 do
begin
  if PriceClose( Bar ) > SMA( Bar, #Close, 14 ) then
    if PriceClose( Bar - 1 ) <= SMA( Bar - 1, #Close, 14 ) then
      BuyAtMarket( Bar + 1, '' );
end;
```

Notice above that the main loop begins at 15, even though we're using a 14 day SMA. This is because we're also looking 1 bar back in the chart (**Bar - 1**) so we bumped up our starting index to compensate for this.

Tip! Instead of testing the crossover condition manually as in the script above, use the WealthScript **CrossOver** function to determine if one series crosses over another on a specific bar. The script then becomes:

```
var BAR: integer;
for Bar := 15 to BarCount - 1 do
begin
  if CrossOver( Bar, #Close, SMASeries( #Close, 14 ) ) then
    BuyAtMarket( Bar + 1, '' );
end;
```

Simulating Market-On-Close

The WealthScript function **BuyAtClose** allows you to simulate a Buy Market-On-Close order.

BuyAtClose(*Bar*, *SignalName*);

In this case, the trade will enter long at the closing value of *Bar*. In actual trading, a broker cannot guarantee that a Market-On-Close order will execute at exactly the closing price. Consequently, you may wish to make use the Slippage feature found under **Tools|Options (F12)|Trading Costs/Control**.

5.2.4 Triggering a Limit or Stop Buy Order

You can also use the `BuyAtStop` and `BuyAtLimit` to simulate stop and limit buy orders. A price must be specified with these order types, therefore an additional argument exists for that purpose.

BuyAtStop(*Bar*, *StopPrice*, *SignalName*);

StopPrice is the price at which a simulated *market order* is placed. A position is established at the *StopPrice* if the opening price of *Bar* is less than or equal to *StopPrice* **and** the bar's high value is greater than or equal to *StopPrice*. However, if the opening price of *Bar* gaps above *StopPrice*, a position is established at the **opening price** of *Bar*. If neither of these conditions are true, then no position is established.

BuyAtLimit(*Bar*, *LimitPrice*, *SignalName*);

For `BuyAtLimit` a position is established at *LimitPrice*, if the open of *Bar* is greater than *LimitPrice* **and** the low of *Bar* is less than *LimitPrice*. If the opening price of *Bar* is less than or equal to *LimitPrice*, a long position is established at the **opening price** of *Bar*. Finally, if the prices fail to fall low enough to the limit objective, no position is established.

There is a chance that these orders might not be fulfilled, so these functions return boolean values indicating whether or not the trades were placed.

Example

```
var BAR: integer;
{ Issue a Limit Order to buy at current bar's close or lower only }
for Bar := 15 to BarCount - 1 do
begin
  if CrossOver( Bar, #Close, SMAseries( #Close, 14 ) ) then
    BuyAtLimit( Bar + 1, PriceClose( Bar ), '' );
end;
```

5.2.5 Checking for Open Positions

It's fine that we can now use our Trading Systems to open long Positions, but how do we close the Positions? WealthScript offers the capability to construct single-Position-only trading systems, or systems that can manage multiple Positions. We'll go over single-Position-only systems for now since it's a simpler concept.

In a single-Position trading system, we need to see if our last Position is active. Use the `LastPositionActive` function to get this information. If our last Position is active, we can branch to our sell rules, otherwise we can see if our buy rules present the opportunity to initiate a new Position.

Example

```
var BAR: integer;
{ A simple, single-position Trading System }
for Bar := 15 to BarCount - 1 do
begin
  if LastPositionActive then
  begin
```

```

    if PriceClose( Bar ) <= SMA( Bar, #Close, 14 ) then
        SellAtMarket( Bar + 1, LastPosition, '' );
    end
else
begin
    if CrossOver( Bar, #Close, SMASeries( #Close, 14 ) ) then
        BuyAtLimit( Bar + 1, PriceClose( Bar ), '' );
    end;
end;
end;

```

Closing Out a Position

You can see from the example above that the **SellAtMarket** function is one way to close out an open long Position. The function takes three parameters.

SellAtMarket(*Bar*, *Position*, *SignalName*);

The first parameter is the *Bar* in which to close out the Position. The second parameter is the Position Number that we want to sell. Since WealthScript can support trading systems that manage multiple positions we need a way to tell the exit rule which position we want to sell. For systems that manage a single Position at a time we can use the **LastPosition** function to return the Position number of the open Position.

Just as **BuyAtMarket** has the counterparts **BuyAtClose**, **BuyAtStop** and **BuyAtLimit**, so does **SellAtMarket** have **SellAtClose**, **SellAtStop** and **SellAtLimit**. Notice the four arguments in the syntax of the latter functions:

SellAtStop(*Bar*, *StopPrice*, *Position*, *SignalName*);

SellAtLimit(*Bar*, *LimitPrice*, *Position*, *SignalName*);

The *StopPrice* and *LimitPrice* arguments retain the same significance as in their "Buy" counterparts. Be aware that if you use the stop or limit sell functions, prices may not reach your stop or limit price, so the trade may not execute.

5.2.6 Using Automated Stops

WealthScript provides six functions that let you apply automated exits to your trading systems. You can call one or more of these functions at the start of your script to install these "automated stops", or simply AutoStops. At the start of your main trading loop, call the **ApplyAutoStops** function to execute the stops. WealthScript will cycle through your open Positions and apply the stops for you automatically. For more information, see the article *Using Automated Exits* on the Wealth-Lab.com site.

The function **SetAutoStopMode** allows you to control how the parameters of AutoStops are interpreted: as percentage (default), point, or dollar values. This is specified using one of the following three constants in the function's argument: **#AsPercent**, **#AsPoint**, or **#AsDollar**. See its entry in the Function Reference for more information.

The syntax of the applicable AutoStop functions and their abridged descriptions are found below. Note that [percentage] is the default interpretation if the **SetAutoStopMode** is not employed in the ChartScript.

InstallStopLoss(*StopLevel*);

StopLevel expresses the maximum [percentage] of loss for an open Position. A gap in prices may result in a loss greater than percentage of *StopLevel*.

InstallTrailingStop(*Trigger*, *StopLevel*);

Trigger is the Position's profit [percentage] that must be reached on a closing basis to activate the stop, and, *StopLevel* is the percentage of the total profit that must be lost (pull back) to trigger the stop. *StopLevel* is always expressed as a percentage and is not affected by **SetAutoStopMode**.

InstallBreakEvenStop(*Trigger*);

Trigger is the Position profit [percentage] that must be reached to activate the breakeven stop.

InstallReverseBreakEvenStop(*LossLevel*);

LossLevel is the [percentage] loss that must be reached to activate a breakeven stop limit.

InstallProfitTarget(*TargetLevel*);

TargetLevel, the profit target level, expresses the [percentage] profit desired to trigger an automatic exit of an open Position.

InstallTimeBasedExit(*Bars*);

Bars represents the number of bars after which a position is automatically closed.

Example

```
var BAR: integer;
{ Use automated Stops to close out the position }
InstallStopLoss( 20 );
InstallProfitTarget( 40 );
InstallTrailingStop( 20, 50 );
InstallBreakEvenStop( 10 );
InstallReverseBreakEvenStop( 20 );
PlotStops;
for Bar := 15 to BarCount - 1 do
begin
  if LastPositionActive then
    ApplyAutoStops( Bar )
  else
    begin
      if CrossOver( Bar, #Close, SMASeries( #Close, 14 ) ) then
        BuyAtLimit( Bar + 1, PriceClose( Bar ), '' );
      end;
    end;
end;
```

5.2.7 Selling Short

Each buy and sell function has a corresponding function for going short and covering a short Position. Replace the "Buy" with "Short" in the function name to initiate a short Position. Replace "Sell" with "Cover" to close a short Position. The Automated Stops can be used for short Positions as well as long.

For more information on these functions, see their entries in the Trading System

Control chapter in the WealthScript Function Reference.

Entering Long

BuyAtClose
BuyAtLimit
BuyAtMarket
BuyAtStop

Entering Short

ShortAtClose
ShortAtLimit
ShortAtMarket
ShortAtStop

Exiting Long Positions

SellAtClose
SellAtLimit
SellAtMarket
SellAtStop

Exiting Short Positions

CoverAtClose
CoverAtLimit
CoverAtMarket
CoverAtStop

5.3 Implementing Trading System Rules

You can use whatever logic based on price, technical indicators, date information, or whatever else you can think of in your entry and exit rules. Get as complicated and creative as you like, but be careful; often times the simpler the trading system, the more robust it will be. Consult the WealthScript Function Reference⁵¹ for a complete list of functions that you can use in your system rules, or with the main icon toolbar visible, **View|Icon Bar**, type **Ctrl+K** to open the QuickRef.

Example

```
var BAR: integer;

{ Buy if Price is higher than the Price of 3 days ago, and 100 day EMA
  is moving up. Sell if Price is lower than the price of 3 days ago, or
  100 day EMA is moving down. }
for Bar := 101 to BarCount - 1 do
begin
  if LastPositionActive then
  begin
    if ( PriceClose( Bar ) < PriceClose( Bar - 3 ) )
      or ( EMA( Bar, #Close, 100 ) < EMA( Bar - 1, #Close, 100 ) ) then
      SellAtMarket( Bar + 1, LastPosition, '' );
    end
  else
  begin
    if ( PriceClose( Bar ) > PriceClose( Bar - 3 ) )
      and ( EMA( Bar, #Close, 100 ) > EMA( Bar - 1, #Close, 100 ) ) then
      BuyAtMarket( Bar + 1, '' );
    end;
  end;
end;
```

Never Look Ahead!

Be sure that your trading system doesn't take advantage of information that it would have no way of accessing in the real world! For example, don't look ahead at Price Series or indicator values. Also, be sure to execute your entry and exit orders at the following bar (typically Bar + 1) to avoid using information from the current bar that you'd have no way of knowing at market open. In system testing these types of errors are termed *peeking* or *postdictive errors*.

Example

```

{ This System takes advantage of future information! It buys and sells
  at the market open on the same bar that it examines closing price! }
var BAR: integer;
for Bar := 1 to BarCount - 1 do
begin
  if LastPositionActive then
  begin
    if PriceClose( Bar ) < PriceClose( Bar - 1 ) then
      SellAtMarket( Bar, LastPosition, '' );
    end
  else
  begin
    if PriceClose( Bar ) > PriceClose( Bar - 1 ) then
      BuyAtMarket( Bar, '' );
    end;
  end;
end;

```

The Trading System above would give you an idea of how well you could do in the market if you had access to supernatural abilities. Although the violation is subtle, you'd be surprised at how much it can impact the bottom line of trading system evaluation!

5.4 Managing Multiple Positions

WealthScript provides the capability to create trading systems that can manage multiple open Positions. You can use this feature to write systems that average down, for example.

Several functions, which found in the Position Management chapter of the WealthScript Function Reference^[5], are available to help you work with information about System Positions. Some of the most important functions are described here.

PositionCount;

returns the total number of Positions that have been created.

LastPosition;

returns the Position number of the last-entered Position. Position numbers range from 0 to **PositionCount** - 1. Note that **LastPosition** = **PositionCount** - 1.

LastLongPositionActive;

returns the Position number of the last long Position.

LastShortPositionActive;

returns the Position number of the last short Position.

Other WealthScript Position functions return information about a specific Position. You pass a Position number to these functions:

PositionActive(Position);

returns *True* if the *Position* is currently open.

PositionEntryPrice(Position);

returns the entry price of the *Position*. See also **PositionExitPrice**.

PositionEntryBar(Position);

returns the integer Bar number on which the *Position* was established. See also

PositionExitBar.**PositionLong(*Position*);**

returns *True* if the *Position* is long and *False* if it is short. See also

PositionShort.

When working with multiple Positions, you typically have a secondary loop within your main loop that goes through each Position and determines whether it should be closed out.

Place your Position closing loop **above** any system entry trading rules. This will prevent the sell logic from being applied to Positions that are opened on the very same bar.

Example

```
{ This Trading System buys whenever RSI crosses above 30, and closes
all open positions when it crosses below 70. }
var BAR, P: integer;
for Bar := 15 to BarCount - 1 do
begin
  if RSI( Bar, #Close, 14 ) < 70 then
    if RSI( Bar - 1, #Close, 14 ) >= 70 then
      begin
        for p := 0 to PositionCount - 1 do
          if PositionActive( p ) then
            SellAtMarket( Bar + 1, p, '' );
        end;
      if RSI( Bar, #Close, 14 ) > 30 then
        if RSI( Bar - 1, #Close, 14 ) <= 30 then
          BuyAtMarket( Bar + 1, '' );
        end;
      end;
end;
```

Splitting Positions

If your strategy includes the purchase (or short sale) of a single position and then closing off parts of it in multiple separate trades, you can split the original position using the SplitPosition function. See the [SplitPosition tutorial](#) in the [Wealth-Lab Knowledge Base](#).

Note: Currently, separate positions cannot be merged or combined.

Shortcut to Closing all Open Positions

Instead of looping through the individual Positions in a multiple-Position strategy, you can use the special #All constant in place of a parameter that requires a *Position* number to close all open long or short Positions at once.

Example

```
var BAR: integer;
for Bar := 20 to BarCount - 1 do
begin
  if CrossOverValue( Bar, CMOSeries( #Close, 20 ), -40 ) then
    BuyAtMarket( Bar + 1, 'CMO' )
  else if CrossUnderValue( Bar, CMOSeries( #Close, 20 ), 40 ) then
    SellAtMarket( Bar + 1, #All, 'CMO' );
end;
```

Optimizing Processing of Active Positions

For sure, the **#All** shortcut is quick to deal with one or more Positions that will be exited at the same time. Some trading systems, however, have different stops, profit targets, or other exit logic for each individual Position. In these cases, it is necessary to process each active Position, one at a time, to execute the intended logic. To find the active positions, however, it is not necessary to loop over *all* the Positions as shown in the RSI trading system above. Doing so can significantly slow down ChartScripts that create many Positions.

Two design patterns are frequently used that optimize processing speed in these cases.

Design Pattern 1

Here, interim variables are declared to hold the `ActivePositionCount` and the number of active Positions that have been processed. Since Positions that are opened earlier are usually closed out first, the `PositionCount` loop counts backwards, starting with the most-recent Position. As each active Position is found, the *Processed* variable is incremented and compared to *APCount*, and, when equal the loop is terminated since all known active Positions have been processed.

```
var p, APCount, Processed: integer;

Processed := 0;
APCount := ActivePositionCount;
for p := PositionCount - 1 downto 0 do
begin
  if PositionActive( p ) then
  begin

    { do something with the active position p here }

    Inc( Processed );
    if Processed = APCount then
      break;
    end;
  end;
end;
```

For an example of pattern 1 in action, see [Interacting Dynamically with Portfolio Level Equity](#) at the Wealth-Lab.com Knowledge Base.

Design Pattern 2

In this ingenious pattern, the variable *p* is initialized to the number of the most-recent active Position + 1. The `repeat/until`^[34] nested loop then finds the very next active position by decrementing *p* by one. The process is repeated by the outer loop *only* for the number of active positions.

```
var a, p: integer; // (in variable declarations)

p := LastActivePosition + 1; { * In a SimuScript replace with p :=
PositionCount; *}
for a := 1 to ActivePositionCount do
begin
  repeat
    Dec( p );
  until PositionActive( p );
```

```
{ do something with the active position p here }  
end;
```

6 Working with Technical Indicator Functions

6.1 Overview

WealthScript provides native functions for dozens of technical indicators. Each of these technical indicator functions has two different syntax forms. The first form returns the value of an indicator at a specific bar in the chart. The second form returns the Price Series handle of the indicator, which you can then pass to functions such as [PlotSeries](#) or [AddSeries](#).

[Accessing Indicator Values](#)^[83]

You *could* use the [GetSeriesValue](#) function to access values of a technical indicator series, but there's a much more intuitive way using the first form of indicator syntax.

[Accessing Indicator Price Series Handles](#)^[84]

As outlined in the chapter [Working with Price Series](#)^[48], handles are used to refer to a complete Price Series. The value that the second form of indicator syntax returns is, in fact, a handle.

For further reference:

The Technical Indicator Functions section of the WealthScript Function Reference^[5] contains a complete listing of the technical indicator functions available. Also, the Wealth-Lab Developer 4.0 Function QuickRef contains a complete example of each native indicator in use.

6.2 Accessing Indicator Values

Use the first form of the indicator function to return the indicator's value at a specific bar in the chart. This form of syntax is commonly an abbreviation, or possibly an acronym, which describes the indicator.

Syntax (Indicator first form, general syntax):

```
indicatorabbr( Bar[, Series[, ParameterList ] ] );
```

Although *Bar* is always required when using the first form, the brackets [] indicate optional arguments that depend on the parameters particular to the indicator. For a typical example, let's turn our attention to the Simple Moving Average value function. Its syntax abbreviation (*indicatorabbr*) is [SMA](#) and the function takes 3 parameters:

```
SMA( Bar, Series, Period );
```

<i>Bar</i>	The <i>Bar</i> number at which we're interested in obtaining the Simple Moving Average.
<i>Series</i>	The handle of the Price <i>Series</i> (or WealthScript function that returns a Price Series handle) of which we want to obtain the moving average.
<i>Period</i>	The <i>Period</i> of the moving average.

The example below prints the Simple Moving Average value for each bar to the Debug

Window. If you run this script and examine the output in the Debug Window you'll notice that the first 29 lines are zero. This is because we're requesting the value of the 30 day moving average, so the first indicator value isn't available until the 30th bar.

Example

```
var Bar: integer;
var SMA_Value: float;
for Bar := 0 to BarCount - 1 do
begin
    SMA_Value := SMA( Bar, #Close, 30 );
    Print( FloatToStr( SMA_Value ) );
end;
```

Indicator Calculation

The first time you call one of the native technical indicator functions or a properly-formed custom indicator, Wealth-Lab calculates the indicator **across the complete Price Series**. Subsequent calls to the indicator function return pre-calculated values. Because of this, you can be sure that repeated calls to access indicator values will be as quick as possible and that unnecessary recalculation is not performed.

6.3 Accessing Indicator Price Series Handles

The second form of the indicator function returns the handle to the complete indicator Price Series (see the [Working with Price Series](#)^[46] topic). These functions are always named the same as their first-form counterparts, but with the word "Series" appended. So, for example, the Simple Moving Average function is named **SMASeries**. Since these functions return the handle that refers to the complete Price Series, they do not include the *Bar* parameter.

Syntax (Indicator second form, general):

```
indicatorabbrSeries( [ Series[, ParameterList ] ] );
```

Now, we complete our explanation with the syntax of the most well-known indicator.

```
SMASeries( Series, Period );
```

<i>Series</i>	The handle of the Price <i>Series</i> (or WealthScript function that returns a Price Series handle) of which we want to obtain the moving average.
<i>Period</i>	The <i>Period</i> of the moving average.

You can pass an indicator Price Series handle to any WealthScript function that is looking for a Price Series parameter. For example, **PlotSeries**. The example below obtains the handle to the Simple Moving Average and plots it on the chart.

Example

```
var nHandle: integer;
nHandle := SMASeries( #Close, 30 );
```

```
PlotSeries( nHandle, 0, #Red, #Thick );
```


7 Accessing Data from Files

7.1 Overview

You can access data from external text files from within your WealthScript code by using the File Access Functions. The File Access Functions provide a way to create, read from, and write to external files.

[Creating and Opening Files](#)^[86]

Just as Price Series use handles to refer to the data entire series, you'll need a *file handle* to point to a file on your computer. The two functions that create and open files return an value that you assign to an integer variable, which is then used as the file handle.

[Reading and Writing](#)^[87]

Once you have a file handle saved in an integer variable, you use it as a reference to read and write from the file.

[Closing Files](#)^[88]

Wealth-Lab automatically closes files that you open from within a script, but you may do it yourself if you like. Read this topic to discover some subtleties in file-access operations during WatchList Scans and \$imulations.

7.2 Creating and Opening Files

Each of the functions described below return an **integer** "File Handle" that is used in subsequent File Access function calls.

Syntax:

FileCreate(*FileName*);

FileOpen(*FileName*);

FileName is a string expression that describes the full path of the file to be created and/or opened. If *FileName* does not include a path, then the file will be created/opened from the Wealth-Lab Developer 4.0's main directory. If a directory path is specified, it must exist otherwise an error will result.

Use the **FileCreate** function to create and open a new, empty file. If *FileName* already exists, it will be deleted and a new file created in its place. See important aspects of **FileCreate** when used in WatchList Scans or \$imulations under the topic [Closing Files](#)^[88].

The **FileOpen** function is used to open an existing file. Nevertheless, if *FileName* does not exist, **FileOpen** will create it.

Example

```
var NewFile, OldFile: integer;
NewFile := FileCreate( 'c:\windows\temp\wltemp.txt' );
OldFile := FileOpen( 'c:\windows\win.ini' );
```

To create a file that includes the symbol name in use by the ChartScript, you can use the **GetSymbol** function as shown in the next example.

Example

```
var NewFile: integer;  
NewFile := FileCreate( 'c:\windows\temp\' + GetSymbol + '.txt' );
```

7.3 Reading and Writing

Use the **FileRead** function to read a line from a file, and the **FileWrite** function to write a line to a file.

Syntax:

FileWrite(*File*, *Line*);

FileRead(*File*);

File is the File Handle that was returned from either **FileCreate** or **FileOpen**. *Line* is a string expression of the data to be "written" or output to *File*.

FileWrite always appends the data string specified in the *Line* parameter to the end of the file. Additionally, the write operation automatically appends carriage return and line feed characters, Chr(13) + Chr(10), to *Line*.

Each time **FileRead** is encountered in your script, it reads the next line from *File* and returns the data as a string. Consequently, you normally find a **FileRead** statement within a loop that continues until the end of file is encountered.

Read and write file operations maintain separate file pointers, so you can even read from a file created with **FileOpen** and write to the same File Handle without disrupting the read.

Use the **FileEOF** (end of file) function to determine if there are any more lines of data to be read from a file. The function returns a **boolean True** if the file pointer has encountered the end of file.

Syntax:

FileEOF(*File*);

Example

```
{ Create a copy of the Win.ini file in the Temp directory }  
var NewFile, OldFile: integer;  
var s: string;  
NewFile := FileCreate( 'c:\windows\temp\wltemp.txt' );  
OldFile := FileOpen( 'c:\windows\win.ini' );  
while not FileEOF( OldFile ) do  
begin  
    s := FileRead( OldFile );  
    FileWrite( NewFile, s );  
end;
```

7.4 Closing Files

Files are automatically closed after the script completes processing. During WatchList Scans or \$imulations, files are automatically closed after the complete Scan or \$imulation. Consequently, when opening a file using **FileCreate**, each individual ChartScript run during a Scan or \$imulation can append lines of data to a single output file without deleting the file that was created at the beginning of the Scan or \$imulation.

You have the option, nevertheless, to close the file explicitly via the **FileClose** function.

Syntax:

FileClose(*File*);

File is the integer File Handle that was returned from either **FileCreate** or **FileOpen**.

Since files are closed automatically after the script completes, this function has limited use. During Scans or \$imulations if you truly want **FileCreate** to delete the previously created file of the same name, include the **FileClose** function in the script.

8 Understanding Time Frames

8.1 Overview

WealthScript provides a set of special functions for accessing data from higher time frames. You can easily create weekly or monthly data from a daily chart. Likewise, you can access daily data from an intraday chart. You can also access higher-time-frame intraday data from an intraday chart, provided that the higher-time-frame data can be created from the lower level bars. For example, a chart of 10-minute bars can be created using 1 or 5-minute bars, but not with 4-minute bars.

It may not be immediately obvious why you would want to use higher-time-frame data when data of greater *granularity* (lower time frame) is available. Imagine though, that you'd like your trade setup to be based on a strong underlying trend turning positive, such as a moving average of weekly bars. When this condition is true, you might trigger the trade Wealth-Lab Developer 4.0 based on some pre-determined Daily price movement. In Wealth-Lab Developer 4.0 you can do this task with the same single set of Daily price bars!

As the following topics are very closely linked, it's best to review them in order.

[Accessing a Higher Time Frame](#)^[89]

Depending on the time frame of your underlying data, different functions are utilized to scale your data in other time frames.

[Expanding the Series](#)^[91]

Once you have the Price Series in a higher time frame, it will be necessary to synchronize it with the original time scale to be useful in ChartScript plotting functions, for example. After you've done this conversion, you can use the new series just like any other Price Series in the original time frame.

[Accessing Higher Time Frame Data by Bar](#)^[93]

You may forego the rather simplistic operation of expanding the entire series and use another set of functions to find the corresponding bar number of the higher-time-frame series in the original Price Series.

[Scaling and Trading](#)^[94]

The technique of compressing data is used to create indicators that you later project back to the original base time frame in which your trades are executed. Do not confuse the purposes of WealthScript Time Frame functions with Wealth-Lab's scaling tools.

See Also: Synchronization Options in the Wealth-Lab Developer 4.0 User Guide.

8.2 Accessing a Higher Time Frame

The first step in accessing data from a higher time frame is to use one of the special "SetScale" functions to change to the desired time scale. WealthScript provides [SetScaleWeekly](#) and [SetScaleMonthly](#) that can be called from a daily chart, and [SetScaleDaily](#) and [SetScaleCompressed](#) that can be called from an intraday chart. You'll receive a compilation error if you try to use one of these functions with data of an incompatible time frame.

Summary of Time Frame Compression Options

<u>Base Time Frame</u>	<u>Compression Function</u>	<u>Resulting Time Frame</u>
Intraday	SetScaleCompressed	Intraday (higher time frame)
Intraday	SetScaleDaily	Daily
Daily	SetScaleWeekly	Weekly
Daily	SetScaleMonthly	Monthly

Example

```
{ Obtain the weekly closing prices from a daily chart }
var WeeklyClose: integer;
SetScaleWeekly;
WeeklyClose := #Close;
RestorePrimarySeries;
```

Note the call to [RestorePrimarySeries](#) at the end of the script. You should always call [RestorePrimarySeries](#) after you're finished operating in the higher time frame.

To get an idea of what's going on behind the scenes, let's inspect the [#Close](#) and WeeklyClose series of a typical, albeit very small, data sample.

	Mon	Tue	Wed	Thur	Fri	Mon	Tue	Thur	Fri
Date	12/16	12/17	12/18	12/19	12/20	12/23	12/24	12/26	12/27
#Close (Daily)	37.32	36.66	36.37	36.11	37.06	36.83	36.81	37.11	36.54
WeeklyClose	37.06					36.54			

The WeeklyClose series contains roughly 1/5 the number of values as the primary Daily series, and, the data values are taken from the last calendar day of the week - the weekly close - which in this case is Friday. If you looped through the bars before the call to [RestorePrimarySeries](#) you would find that the bars retain the calendar day of the first calendar day of the week (Monday). This is really immaterial, and you'll see why when you [Expand the Series](#)^[91] to use its data in your ChartScript.

Expanding the Weekly Series

The example above returned the weekly closing Price Series for our daily data. If our daily chart had 1000 bars, the "WeeklyClose" Price Series would roughly contain only 200 bars (5 trading days per week). If you tried to use this Price Series in a function such as [PlotSeries](#), you'd receive an error (or no result), because the weekly Price Series has fewer bars than the daily Price Series. There are two ways to "expand" the higher-time-frame data and make it available for use from within the lower level chart: [Expanding the \[entire\] Series](#)^[91] and [Accessing Higher Time Frame Data by Bar](#)^[93].

8.3 Expanding the Series

The first method of accessing the higher time frame data is arguably simpler. WealthScript provides special functions to automatically expand the higher-time-frame data. You can use the [DailyFromWeekly](#) and [DailyFromMonthly](#) functions in daily charts, and the [IntradayFromDaily](#) or [IntradayFromCompressed](#) functions in intraday charts. After calling the appropriate function to expand the higher-time-frame Price Series, use [GetSeriesValue](#) to obtain the value of the converted series at a particular Bar Number.

Summary of Timeframe Expansion Options

<u>Base Time Frame</u>	<u>Expansion Function</u>	<u>Use After Compression With</u>
Intraday	IntradayFromCompressed	SetScaleCompressed
Intraday	IntradayFromDaily	SetScaleDaily
Daily	DailyFromWeekly	SetScaleWeekly
Daily	DailyFromMonthly	SetScaleMonthly

These functions return a new Price Series that is synchronized to the lower time frame data. The expanded Series contains a number of repeated values. For example, a weekly series converted to a daily series generally will have 5 repeated values in a row, one for each day of the week.

Note: Upon expansion, alignment of compressed data is greatly affected by the [Compressed Price Series Alignment Option](#)^[91].

The example below shows how to convert the weekly data to a daily series for plotting. This effectively overlays the weekly close over the daily chart.

Example

```
var WeeklyClose, WeeklySynched: integer;
SetScaleWeekly;
WeeklyClose := #Close;
RestorePrimarySeries;
WeeklySynched := DailyFromWeekly( WeeklyClose );
PlotSeries( WeeklySynched, 0, #Red, #Dotted );
```

At this point you can use the WeeklySynched series the same as any other Price Series in the Daily time frame.

Apply indicators, such as a Weighted Moving Average, to the compressed (higher-time-frame) Price Series *prior to* using an expansion function. This may be done before or after the call to [RestorePrimarySeries](#). Adding to our previous example, we demonstrate how to do this.

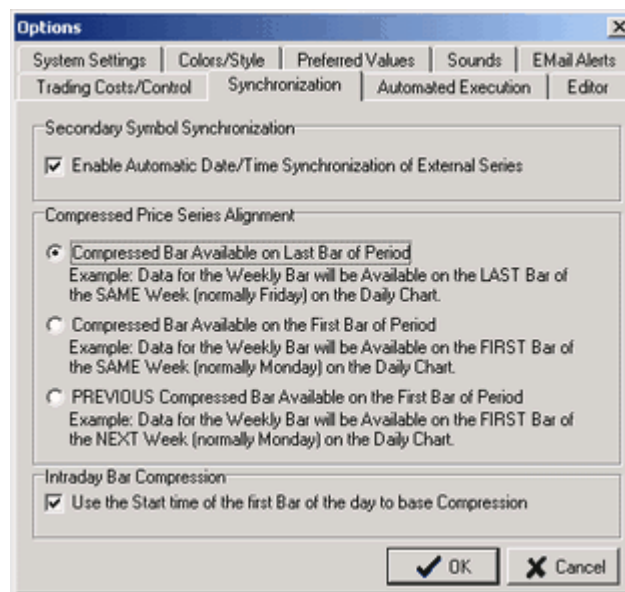
Example

```
{ Create a 5-period Weighted Moving Avg of the weekly price series
  derived
  from daily data and then use it in the Daily time frame. }
var WeeklyClose, WeeklySynched, AvgWeekly, AvgWeeklySynched: integer;
SetScaleWeekly;
WeeklyClose := #Close;
AvgWeekly := WMASeries( WeeklyClose, 5 );
RestorePrimarySeries;
WeeklySynched := DailyFromWeekly( WeeklyClose );
AvgWeeklySynched := DailyFromWeekly( AvgWeekly );
PlotSeries( WeeklySynched, 0, #Red, #Dotted );
PlotSeries( AvgWeeklySynched, 0, #Blue, #Dotted );
```

Compressed Price Series Alignment Option

It's important not to "look ahead" while back testing trading systems as this will cause postdictive errors in your scripting that usually leads to overly-inflated profits. If you're not careful, this can be easy to do when synchronizing an expanded Price Series.

You have control of how to align and display data from a compressed Price Series. This option is provided by selecting **Tools | Options** (or by striking the F12 function key) and then the Synchronization tab.



Considering again the previous examples of overlaying compressed Weekly data on top of its corresponding Daily Price Series, let's inspect the same sample data set to see how these options affect the outcome. Here, we observe the result of the repeated data values after the DailyFromWeekly function call based on the selection indicated. We'll refer to these as Options #1, #2, and #3 from top to bottom.

Compressed Bar Available on Last Bar of Period

(Option #1)	Mon	Tue	Wed	Thur	Fri	Mon	Tue	Thur	Fri
Date	12/16	12/17	12/18	12/19	12/20	12/23	12/24	12/26	12/27
#Close (Daily)	37.32	36.66	36.37	36.11	37.06	36.83	36.81	37.11	36.54
WeeklySynched	36.52	36.52	36.52	36.52	37.06	37.06	37.06	37.06	36.54

Compressed Bar Available on the First Bar of Period

(Option #2)	Mon	Tue	Wed	Thur	Fri	Mon	Tue	Thur	Fri
Date	12/16	12/17	12/18	12/19	12/20	12/23	12/24	12/26	12/27
#Close (Daily)	37.32	36.66	36.37	36.11	37.06	36.83	36.81	37.11	36.54
WeeklySynched	37.06	37.06	37.06	37.06	37.06	36.54	36.54	36.54	36.54

Ⓢ PREVIOUS Compressed Bar Available on First Bar of Period

(Option #3)	Mon	Tue	Wed	Thur	Fri	Mon	Tue	Thur	Fri	Mon
Date	12/16	12/17	12/18	12/19	12/20	12/23	12/24	12/26	12/27	12/30
#Close (Daily)	37.32	36.66	36.37	36.11	37.06	36.83	36.81	37.11	36.54	36.99
WeeklySynched	36.52	36.52	36.52	36.52	36.52	37.06	37.06	37.06	37.06	36.54

When testing a trading system using compressed data in a more granular time frame (i.e., expanded), it's clear from these illustrations that either Option #1 or #3 must be selected. The difference in the first (default) and third options is one of a self-imposed delay. In other words, if you were to run an end-of-day Scan with Option #1 on Friday (after the close), you could generate trading signals for Monday's open based on the current week's data. In contrast, with Option #3, this data would not be available until Monday night's Scan. Either method is acceptable and which one you choose depends on your methodologies.

If Option #2 were selected, you would incorrectly be using data only available at a later time in actual trading, as you can verify in the illustration above. Nevertheless, you may wish to see the data using the second convention for charting purposes, or to create some sort of idealized trading system. For these reasons it is available for your discretionary use.

For a graphical interpretation of this discussion using Daily/Intraday time frames, see the following Knowledge Base article: <http://www.wealth-lab.com/cgi-bin/WealthLab.DLL/kbase?id=77>

8.4 Accessing Higher Time Frame Data by Bar

The second method of accessing the higher-time-frame data is to determine the bar number in the higher-time-frame Price Series that corresponds to the bar number in the lower level Price Series. WealthScript provides special functions to do this: [GetWeeklyBar](#) and [GetMonthlyBar](#) for daily charts and [GetDailyBar](#) and [GetCompressedBar](#) for intraday charts. Once you have determined the corresponding bar number, you can use [GetSeriesValue](#) to obtain the value of the converted series at that bar.

Summary of Get Bar Options

Base Time Frame	"Get Bar" Function	Use After Compression With
Intraday	GetCompressedBar	SetScaleCompressed
Intraday	GetDailyBar	SetScaleDaily
Daily	GetWeeklyBar	SetScaleWeekly
Daily	GetMonthlyBar	SetScaleMonthly

The example below first grabs the weekly closing price series. It then goes through each bar of the daily chart and finds the corresponding weekly closing prices for the previous 2 weeks. If the previous week's close was higher than the close 2 weeks ago, the script colors the daily bar green. After running the example, a look at the Debug Window resulting from the [Print](#) statement will provide additional insight.

Example

```
var WeeklyClose, Bar, BarWeekly: integer;
SetScaleWeekly;
WeeklyClose := #Close;
RestorePrimarySeries;
```



```

for Bar := 12 to BarCount - 1 do
begin
  BarWeekly := GetWeeklyBar( Bar );
  Print( IntToStr(BarWeekly) + ', ' + IntToStr(Bar) );
  if GetSeriesValue( BarWeekly - 1, WeeklyClose )
    > GetSeriesValue( BarWeekly - 2, WeeklyClose ) then
    SetBarColor( Bar, #Green );
end;

```

Let's recap.

The difference between the two methods of accessing higher-time-frame data is subtle. After **RestorePrimarySeries**, in the "expansion" method, we simply create another new Price Series that contains repeated values synchronized with the original Price Series. Expanding the higher-time-frame series in this way is necessary if you want to plot its values using the **PlotSeries** function.

In the less-intuitive method above, the higher-time-frame series is never expanded. Its values are obtained by finding bar numbers that correspond between the two time frames. Since the repeated values associated with the series-expansion method do not exist, we have an advantage in memory savings.

8.5 Scaling and Trading

The Time Frame functions discussed in the preceding topics are probably the most difficult to understand of the WealthScript functions, yet once you have mastered them, you will see how easy it is to create complex trading systems based on data and indicators in other time frames.

Two concepts relating to time frames are necessary to understand. The first is that you can **Scale** the data in the primary series using the using the Scale toolbar for ChartScripts (**DWM**, **5**, **3**) and the Scale tab controls in the \$imulator, Rankings, and Scans tools. Scaling in this manner re-creates the data into a new base time frame, which allows you to generate trades in the new scale. Note that the **ChangeScale** function serves this same purpose, but it is useful only in the ChartScript window.

Unlike the aforementioned scaling features of Wealth-Lab, the Time Frame functions do not change the base time frame and therefore do not allow you to make trades on resultant Price Series. This group of functions allow you to create indicators in more compressed time frames that must be *restored* or *projected* back to the original base time frame.

Scaling and Time Frame Notes:

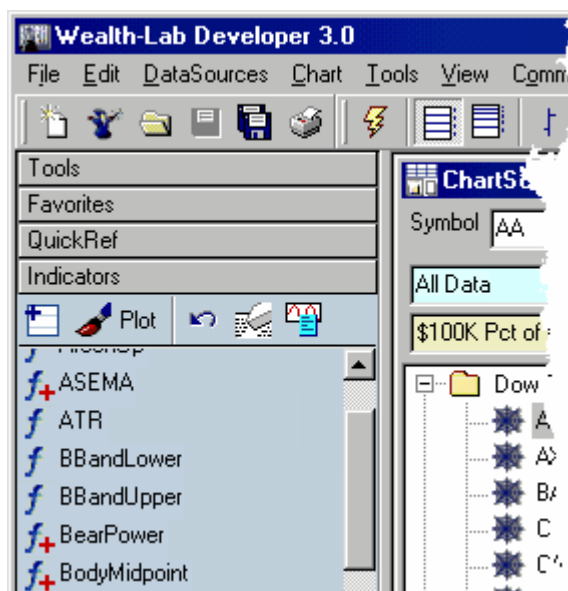
1. Transforming intraday data to multiples of its underlying interval using the Scale toolbar is currently available only for ChartScript windows. A similar *intraday scaling* feature does not exist for the \$imulator, Scans, and Rankings.
2. It is not possible to place trades on a Primary Series that has been time-compressed *from within a script* using **SetScaleCompressed** or **SetScaleDaily**, for example. These WealthScript Time Frame functions allow you only to generate indicators and other Price Series in a more compressed time frame that must be referenced back to the base time frame.

9 Creating a Custom Indicator

9.1 Overview

You can create custom technical indicators in Wealth-Lab Developer 4.0 that are treated just like native indicators. Custom indicators are scripts composed of two functions (as are native indicators). One function returns the value of the indicator at a specific bar. The second function returns a handle to the complete Price Series for the indicator.

A custom indicator is nothing more than a specially formed ChartScript that is stored in the "Indicators" folder. Custom indicators appear in the Indicator list within the main icon tool bar panel, which is docked on the left side of the screen. You can distinguish custom indicators from native WealthScript indicators because they have a red cross next to the function symbol.



[Using the New Indicator Wizard](#)^[96]

Even for experienced users, the New Indicator Wizard is a great place to start to generate the boilerplate code for your indicator. After defining a few parameters, you'll only have to program how the indicator is calculated. It's a snap!

[The Guts of a Custom Indicator](#)^[99]

If you're a code hound, you'll probably be interested in the details of how Wealth-Lab can calculate so quickly the value of your indicator each time you reference it in a ChartScript. (The secret is that it calculates the entire indicator series once only!)

[Other Possibilities](#)^[101]

There's always more than one way to code an idea. Coding an indicator is not an exception to this rule.

9.2 Using the New Indicator Wizard

You can use the New Indicator Wizard to help produce a new custom indicator. The Wizard generates the required boilerplate code for the indicator, and stores the ChartScript in the "Indicators" folder. Start the New Indicator Wizard from the file menu by selecting **File|New Indicator Wizard**, or by simply striking **Ctrl+I**.

Note: You must know how to [work with Price Series](#)^[48] before attempting to create custom indicators.

Step 1. Indicator Name

The first step of the Wizard is to select the new indicator's name. You cannot select a name of an existing ChartScript or a native WealthScript function. The Wizard uses the indicator name to create two user-defined functions in the resulting ChartScript code. The first function adopts the name of the indicator, and the second function appends the word "Series" to the indicator name.

Step 2. Indicator Parameters

The next step of the Wizard is to select the parameters that the indicator will accept. Here you are actually creating the parameter list that appears in the indicator's function declarations. Select one of the names provided in the Parameter Name drop down box, or type your own variable name.

Note: Do not use variables named *Bar*, *sName*, or *Value*. The Indicator Wizard reserves these names for its output.

Before clicking "Add Parameter", select the data type of your variable from the other drop down box. Continue this process for as many parameters as are necessary.

If any of the parameters is destined to be a Price Series handle, you should include the word "Series" in the parameter name and select "integer" as the data type. Wealth-Lab Developer 4.0 looks for the word "Series", and if found will provide the list of Price Series constants ([#Open](#), [#High](#), [#Low](#), [#Close](#), [#Volume](#), [#Average](#)) whenever the indicator's Properties Dialog is displayed, after dragging and dropping an indicator on a chart pane for instance.

When finished adding parameters, select the "OK" button to create the indicator script.

Step 3. New Indicator Wizard Output

The New Indicator Wizard uses the information you provided to create a new ChartScript and places it in the "Indicators" folder. This ChartScript contains the skeleton code that the custom indicator requires. You have to now fill in the portion of the code that actually calculates the indicator's value. The following code snippet is part of the resulting ChartScript:

```
Result := CreateNamedSeries( sName );
for Bar := Period to BarCount - 1 do
begin
{ Calculate your indicator value here }
Value := 0;
SetSeriesValue( Bar, Result, Value );
end;
```

Your job is to replace the statement, *Value := 0;* with code that calculates the value of the indicator. Depending on the complexity of your indicator, this may be a few or many statements. Note that this code is already within a for loop that cycles through each bar of the chart. Your code should ultimately assign a numeric expression (other than zero) to the variable *Value*, the value of your indicator at *Bar*, which is conclusively stored in the indicator's series using [SetSeriesValue](#). In other words, you're *filling* the blank series created by [CreateNamedSeries](#) bar by bar.

Note that the special variable *Result* is used as the handle to the indicator's Price Series. It's important that when setting the series value at each bar that you use the *Result* handle. When finished, don't forget to save your work!

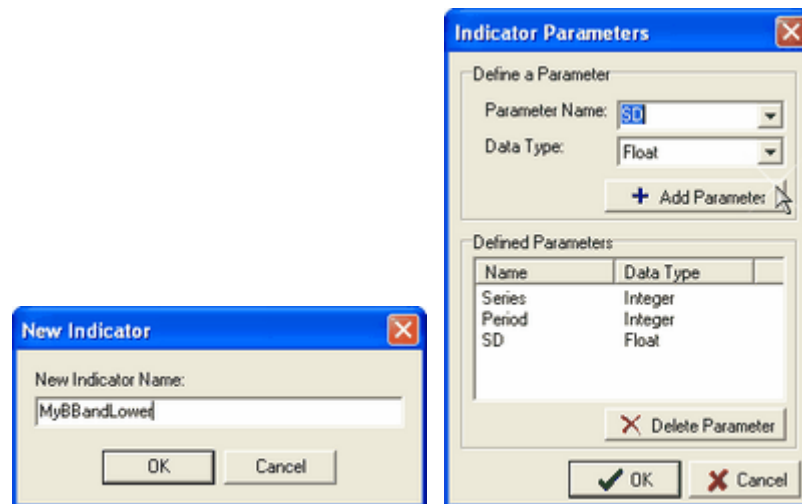
Note: Do not create another series using [CreateSeries](#) and then assign its handle to *Result*. This will have the effect of assigning an unnamed series to *Result*, and therefore subsequent calls to the indicator series or attempting to obtain a specific value at a single bar will return zero value.

Custom Indicators Derived from Other Indicators

As we've just seen, the Indicator Wizard provides a code template for creating indicators that are built *bar by bar*. However, many custom indicators can be derived more efficiently by combining existing indicators with [series math](#)^[58] using [Price Series operator functions](#)^[59] like [AddSeries](#), [MultiplySeries](#), [MultiplySeriesValue](#),

etc. In these cases, since the indicator is not built bar by bar in WealthScript, we need to make modifications to the wizard code's *Series function. The Indicator Wizard is still valuable because it generates the proper function declarations and parameter lists.

As an example, let's recreate the BBandLower indicator in custom indicator form. As the following image shows, we've invoked the Wizard by striking Ctrl+I (or from the File menu), named our indicator *MyBBandLower*, and added the required parameters and Data Types.



Creating a WealthScript version of the BBandLower indicator.

Upon clicking OK to the Parameters dialog, *MyBBandLower* is saved to the Indicators folder and is immediately registered in the main Indicators toolbar. Since we know that the lower Bollinger Band is calculated by subtracting the StdDevSeries multiplied by the specified standard deviations from the simple moving average of the same Period, we can express it as follows:

```
function MyBBandLowerSeries( Series: Integer; Period: Integer; SD:
Float ): integer;
begin
    var Diff: integer;
    var sName: string;

    sName := 'MyBBandLower(' + GetDescription( Series ) + ',' + IntToStr(
Period ) + ',' + FloatToStr( SD ) + ')';
    Result := FindNamedSeries( sName );
    if Result >= 0 then
        Exit;

    Diff := MultiplySeriesValue( StdDevSeries( Series, Period ), SD );
    Result := SubtractSeries( SMASeries( Series, Period ), Diff );
    SetDescription( Result, sName );
end;
```

Take note of the major changes to the MyBBandLowerSeries code generated by the Indicator Wizard:

- CreateNamedSeries is not necessary because our indicator is created as the result of another indicator function.
- The for/do loop is eliminated. It's not necessary to calculate the indicator's value on a bar by bar basis.
- SetDescription assigns a string name, *sName*, to the description of our final Result series. As explained in a [subsequent topic](#)⁹⁹, Wealth-Lab uses

descriptions to access indicators whose values have already been calculated.

Wealth-Lab, however, automatically creates unique internal descriptions for native indicators, and consequently it is actually not required to form the *sName* description and assign it to the result using `SetDescription`. Therefore, we can simplify the custom indicator code even further as follows:

```
function MyBBandLowerSeries( Series: Integer; Period: Integer; SD:
Float ): integer;
begin
  var Diff: integer;
  Diff := MultiplySeriesValue( StdDevSeries( Series, Period ), SD );
  Result := SubtractSeries( SMASeries( Series, Period ), Diff );
end;
```

Or simply,

```
function MyBBandLowerSeries( Series: Integer; Period: Integer; SD:
Float ): integer;
begin
  Result := SubtractSeries(
    SMASeries( Series, Period ),
    MultiplySeriesValue( StdDevSeries( Series, Period ), SD )
  );
end;
```

9.3 Deleting a Custom Indicator

Unless you're clairvoyant, not all the custom indicators that you create will be useful, and therefore you'll need a means to remove them. Since Custom Indicators (and Studies) are simply special ChartScripts saved in the "Indicators" folder, to delete a custom indicator you simply have to remove its ChartScript using normal Explorer procedures:

1. Open the ChartScript Explorer (Ctrl+O)
2. Navigate to the "Indicators" folder. (To remove a Study, go to the "Studies" folder.)
3. Locate the Custom Indicator or Study.
4. Click the item to highlight it, and strike the **Delete** key on the keyboard.

After confirming the deletion, the Custom Indicator will no longer appear in the main Icon Bar under the "Indicators" section.

9.4 The Guts of a Custom Indicator

The New Indicator Wizard does the work of setting up the custom indicator for you, but it may be helpful to understand how custom indicators work internally. The following information **is not required** to create a custom indicator, so read on only if you are interested in the details.

Like all native WealthScript indicator functions, a custom indicator is composed of two functions. The first function returns the value of the indicator at a specific bar. The second function (with the word "Series" appended to it) returns the handle to the complete Price Series.

For example, if we created a custom indicator called "Test", we wind up with two functions, one called `Test` and another called `TestSeries`.

```
function Test( Bar: integer; Series: Integer; Period: Integer ): float;
begin
    Result := GetSeriesValue( Bar, TestSeries( Series, Period ) );
end;
```

The implementation of `Test` just grabs the value at the desired bar by calling `GetSeriesValue`. The second parameter of `GetSeriesValue` is a Price Series handle. In this case, we pass the second custom indicator function. So, in essence, the `Test` function always passes control to the **TestSeries** function to actually obtain its value.

This means that all of the work to calculate the indicator values is accomplished in the **TestSeries** function. Here, we use some special WealthScript functions to make sure that we only construct the indicator Price Series once, the first time **Test** or **TestSeries** is referenced, which leads to increased performance of the script.

We'll assume that the **Test** indicator had 2 integer parameters, *Series* and *Period*. The first thing the **TestSeries** function does is create a string that uniquely identifies the requested indicator series.

```
sName := 'Test(' + GetDescription( Series ) + ',' + IntToStr( Period )
+ ')';
```

Now that the function has a unique string that identifies this Price Series, it can see if the Price Series was previously created.

```
Result := FindNamedSeries( sName );

if Result >= 0 then
    Exit;
```

The `FindNamedSeries` function looks for a Price Series with a certain internal name. If a Price Series with the specified name was found, the series was already calculated, so we just assign it to the *Result* variable and exit the function. If the Price Series wasn't created, then we need to create it and populate it with indicator values.

```
Result := CreateNamedSeries( sName );
for Bar := Period to BarCount - 1 do
begin
    { Calculate your indicator value here }
    Value := 0;
    SetSeriesValue( Bar, Result, Value );
end;
```

The `CreateNamedSeries` function is similar to the frequently used `CreateSeries`. It too creates an empty Price Series. The difference is that `CreateNamedSeries` associates an internal name to the Price Series. We can then use `FindNamedSeries` to retrieve the Price Series by that name.

Related Topic: `SetDescription`

9.5 Other Possibilities and FAQs

The method that the New Indicator Wizard uses to build a custom indicator is only one possibility. Another way to proceed would be to calculate and return the indicator value within the **Test** function itself. Then, in the **TestSeries** function, loop through each bar and call the **Test** function to assign the value. This method would be more optimal for ChartScripts that access indicator values sporadically and do not access the complete Price Series for the indicator.

If desired, [submit](#) your correctly-formed custom indicators to the [WealthScript Code Library](#). This will then be available as a custom indicator on the web site and in Wealth-Lab Developer 4.0 when users perform a "Download ChartScripts" action.

I saved a custom indicator to the 'Indicators' folder but it doesn't appear in the Indicators icon bar?

A new custom indicator will be added to the icon bar if:

- you used the [New Indicator Wizard](#)^[96] to create the custom indicator, and,
- the indicator was added automatically following a Community Download.

Otherwise, if you added the indicator by saving a ChartScript to the Indicators folder, the indicator will appear in the icon bar the next time you start Wealth-Lab Developer 4.0.

How do I use Custom Indicators?

Answer: *just like any other native [technical indicator](#)*^[83].

The main difference is that you must make your script *aware* of the custom indicator's code by placing a special "include" comment at the top of your ChartScript that identifies the *name of the custom indicator script*. The Include Manager can help in locating custom indicators and placing these special \$Include comment(s).

For example, assume that you wanted to use the custom indicator *VK_WH* which has been saved as *VK WH Band* in the Indicators folder. A typical process would be as follows:

1. Open a ChartScript (Ctrl+O) to which you wish to add the indicator or start with a new one (Ctrl+N).
2. Click the Editor view and strike F6 to launch the Include Manager.
3. Locate the *VK WH Band* script, place a check mark next to it and click OK. This action automatically places the required `{ $I 'VK WH Band' }` comment at the top of the ChartScript.
4. At this point, unless you're very familiar with the parameter list of this indicator, you'll need to refer to the indicator's code and Description to use it properly. The following code show is an example that simply plots the indicator. Note that the *VK_WHSeries* function is defined in the 'VK WH Band' indicator script.

Example

```
{ $I 'VK WH Band' }
var VK_WHSer: integer;
```

```
VK_WHSer := VK_WHSeries( #Close, 5, 20 );  
PlotSeriesLabel( VK_WHSer, 0, 009, #Thin, 'VK_WH(#Close,5,20)' );
```

Where can I use Custom Indicators?

You can use custom indicators in any ChartScript or IndexScript (Index-Lab). Custom Indicators are not valid in CommissionScripts, PerfScripts, or SimuScripts.

10 CommissionScripts

10.1 Overview

You should always include real-world trading costs to add fidelity to your backtesting. The Options Dialog, **Tools|Options (F12)**, includes a Trading Costs/Control tab that provides selections for commissions and slippage that you will experience in real-world trading.


If your broker uses a flat-fee commission for each trade, then you may select the "per Trade" One Way Commission option, which simply deducts a fixed amount from each trade in a simulation. Likewise, the "per Share" option reduces a trade's gross profit or loss by the number of shares multiplied by the value entered. Still, these simple commission options do not include other small adjustments that your broker can make on a per trade basis, such as the SEC fee for sale transactions in the U.S., which at the time of this writing is \$0.0468 per \$1,000.

Some brokers use graduated commission schedules or base their fees on a percentage of trade volume. CommissionScripts give you complete control over calculating simple to the most complex commission schedules used by brokers worldwide.

[CommissionScript Variables](#)^[103]

Wealth-Lab makes specific trade data available to your script through the use of special 'CM' variables. You'll need these in order to calculate commissions. You'll assign the final commission value to the **CMResult** variable, for example.


[Creating and Testing CommissionScripts](#)^[104]

CommissionScripts are a special type of ChartScript that contain logic only for calculating commissions. After completing the code, save the script to the special  **CommissionScripts** ChartScript folder. You'll then be able to select it for use as the CommissionScript in the Options Dialog.

10.2 CommissionScript Variables

CommissionScripts work by having access to the following special variables, which Wealth-Lab loads with values that apply to the trade being processed. Each item's return type is provided below and is further defined in the WealthScript Function Reference^[5] as well as in the QuickRef:

```
CMShares      : integer;
CMPrice       : float;
CMEntry       : boolean;
CMSymbol      : string;
CMDataSource  : string;
CMOrderType   : integer;
CMResult      : float;
```

Using these special 'CM' variables, you can emulate the your broker's calculation and assign the result to the **CMResult** variable. Once complete, save the script to the  **CommissionScripts** ChartScript folder. At this point, the script will be available as a selection in the CommissionScript dropdown control in the Options Dialog.

When using CommissionScripts, Wealth-Lab executes the selected CommissionScript

for each trade processed during a simulation - once for each entry and once for each exit signal. The value calculated and assigned to the **CMResult** variable will then be used as the trade's commission cost.

Wealth-Lab reduces the account equity by the commission amount on the bar on which the trade takes place. Net profit reported for each trade in the Trades view includes all entry and exit (if closed) commissions.

WealthScript Functions Compatible with CommissionScripts

You can declare any of the standard variable [data types](#) for use in a CommissionScript as well as object types. However, not all WealthScript functions are available for use in CommissionScripts. Generally speaking, in addition to the special 'CM' variables and [GetGlobal/SetGlobal](#) system functions, you may use only the Math and String categories of WealthScript functions in CommissionScripts. Though most commission calculations are expected to require only the most basic math functions, it should be clear that some Math functions cannot be utilized, including:


[Correlation](#), [LinearRegLine](#), [LineExtendX](#), [LineExtendY](#), and [TrendLineValue](#)

Finally, user-defined functions and procedures may be declared at the top of a CommissionScript, however, `{ $Includes }` cannot be used.

10.3 Creating and Testing CommissionScripts

Creating a CommissionScript

The procedure to make a CommissionScript is quite simple:

- Step 1:** Open a New ChartScript (Ctrl + N) and select the Editor Tab.
- Step 2:** The template (or skeleton) code will not be useful, so clear it to create a fresh workspace.
- Step 3:** Using the aforementioned 'CM' variables program your broker's logic and assign the final result to **CMResult**.
- Step 4:** Save the script (Ctrl + S) to the  **CommissionScripts** ChartScript folder.

The following sample CommissionScript is based on a commission structure with the following characteristics:

- 1¢ per share up to 500 shares
- ½¢ for shares over 500 shares
- \$1 minimum


Example

```
if CMShares <= 500 then
    CMResult := CMShares * 0.01
else
    CMResult := ( 500 * 0.01 ) + ( ( CMShares - 500 ) * 0.005 );

if CMResult < 1 then
    CMResult := 1;
```

Testing a CommissionScript

Most CommissionScripts will be straightforward in nature and will certainly be simplistic to all but the most novice programmer. Nevertheless, typos and other errors can slip into our code so it's necessary to exercise a CommissionScript prior to committing it to a large \$imulation process. The following guidelines may help in building confidence that your CommissionScript is functioning properly:


1. After saving the script to the  **CommissionScripts** folder, be sure to select the CommissionScript for use in the Options Dialog|Trading Costs/Control tab.
2. To more easily isolate trading costs due to commissions, disable Slippage.
3. Begin by executing the CommissionScript by itself in a ChartScript window. Though you will not be able to determine that your commission algorithm is functioning correctly, running the script gives you a chance to correct syntax errors. If you corrected errors, save and close the CommissionScript.
4. Open a ChartScript of your choice that contains trading system rules and execute it. Determine the gross profit of a trade based on the number of shares/contracts and entry/exit prices. Subtract the Net Profit provided in the Trades view from the calculated gross profit. The result will be the value(s) calculated by the CommissionScript.
5. Re-activate Slippage, if desired.

Remarks:

- If you find that no commissions are ever deducted when using your commission script, check the script for errors.
- Commissions are shared equally between split positions in the ChartScript window. However, due to the way the \$imulator operates internally, all commissions are retained by the initial position in a split.

11 PerfScripts

11.1 Overview

PerfScripts, or **Performance Scripts**, are *Scriptable Performance Reports*. You can customize Wealth-Lab Performance Reports to display whatever performance metrics that you can imagine using the PerfScript feature. Performance Scripts must be saved to the special  **PerfScripts** folder, where a sample named "Standard PerfScript" is included with your Wealth-Lab Developer 4.0 installation that duplicates the standard Wealth-Lab Performance Report.

When enabled on the Performance View tab of the ChartScript Window or \$imulator tools, Wealth-Lab will execute a PerfScript four times to process *All Trades* (*Long+Short*), *Long Only*, *Short Only*, and *Buy & Hold* positions. Since Wealth-Lab automatically makes the appropriate group of positions available to the PerfScript during each of the four runs, it's not necessary to write special code to test position types (long, short, etc.).

[PerfScript Functions](#)^[106]

[Creating PerfScripts](#)^[107]

[Using PerfScripts](#)^[108]

11.2 PerfScript Functions

PerfScripts have a repertoire of dedicated functions that the QuickRef and WealthScript Function Reference describe in detail. The first five functions enable you to control the format of the metrics that you add to Wealth-Lab's ChartScript and \$imulator Performance Views. Wealth-Lab calculates account exposure and facilitates its access through the **AccountExposure** function.

PerfAddCurrency
PerfAddPct
PerfAddString
PerfAddNumber
PerfAddBreak
AccountExposure
StartingCapital
CashInterest
MarginLoan
TotalCommission

In addition to these PerfScript-specific functions, you can also use WealthScript functions from the Data Access, Date/Time, File Access, Technical Indicators, Math, Position Management, Price Series, and String categories. It should be clear that *not all* functions in these categories lend themselves to PerfScript analysis, such as the "Set" Position Management functions.

Concept Note

A performance script processes equity curve data and the individual trading details from all symbols following a \$imulation or ChartScript Window execution. References to Standard Price Series like **#Close**, **#Volume**, etc. cannot be permitted in PerfScripts because the idea of a Primary Series does not exist. Consequently, Price Series functions like ATR, MFI, etc. that require Standard Price Series *cannot be used* in PerfScripts. Generally, only a custom series created during the execution of the PerfScript, e.g., the result of the CreateSeries function, can be used as an argument for Price Series functions that accept an integer Series argument, such as SMA, Momentum, BBandLower, etc.

PerfScript Constants

- #WinLoss** Each of the **PerfAdd** functions contains a *Color* parameter that controls the metric's text color in the report. **#WinLoss** paints positive values *green* and negative values *red*. In addition, you can use any of the standard color [constants](#)^[69] or a 3-digit number.
- #Bold, #Italic** Each of the **PerfAdd** functions contains a *Style* parameter that controls the appearance of the metric's Label in the report. You can pass **#Bold**, **#Italic**, or 0 for regular style.
- #Equity** Standard handle to the Equity curve series (PerfScript only) in Portfolio Simulation Mode. In Raw Profit Mode, this handle returns the Profit curve.

11.3 Creating PerfScripts

Of the PerfScript functions, four are used to add data to a *performance record*, which is simply a single row of text in the Performance Report. Each row must have a unique *Label*. Depending on the type of data to be displayed, you'll reference this *Label* in one of the PerfAdd functions: **PerfAddCurrency**, **PerfAddNumber**, **PerfAddPct**, or **PerfAddString**. Consequently, the same performance record can display different types of data as required for *All Trades*, *Long Only*, etc.

For example, for any performance metric that involves a division, you should include logic to detect if the divisor is zero prior to the division operation. If it is, then you can use **PerfAddString** to show 'INF'. Otherwise, use one the other functions to display a number with the appropriate format. Likewise, you can catch and [handle the error](#)^[44].

Start with the Standard

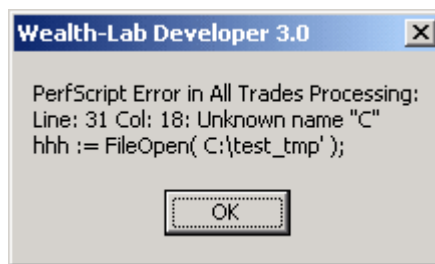
A sample PerfScript called "Standard PerfScript" that duplicates and adds to the standard Wealth-Lab Performance Report will be included in the PerfScripts ChartScript folder, where all PerfScripts must be maintained. A second sample, "Standard PerfScript with Interest" includes the use of the **CashInterest**, **MarginLoan**, and **TotalCommission** functions. The standard scripts are the best place to start when creating your customized PerfScript. Save one of the "standards" with a different name and start deleting or adding the calculations for metrics that you would like to see displayed. With the standard as a model, it's not likely that you'll

need any help to create custom formulas for new performance metrics.

PerfScript Errors

After writing a new PerfScript or editing an existing one, run the script in the ChartScript window by clicking any symbol. Doing this will allow you to correct syntax errors and most run-time errors in the script prior to actually [using it](#)^[108] to generate a report. Other PerfScript errors may not be caught in the "ChartScript Mode". For example, since Trading System functions are not compatible with PerfScripts, yours should not actually create trades. If it does, this mistake will not be detected until Wealth-Lab executes the script during a ChartScript's post processing to generate the actual performance report.

During a ChartScript's post processing, any irregularity in a PerfScript will generate an error dialog like the one below. It will identify in which of the four PerfScript runs the error occurred (*All Trades*, *Long Only*, *Short Only*, or *Buy & Hold*), the line number and error text, and finally the actual line of code. As a consequence of an error, a performance report will not be generated.



A dialog notifies you of run-time errors during the execution of a PerfScript.

Tip: Add metrics specific to Raw Profit and Portfolio Simulation Mode by testing if `StartingCapital = 0`. Like in the PerfScript sample, a boolean variable `bRawProfit` is set early on to control the output for the two modes.

Note that PerfScripts (like the sample) may contain several "BarCount" loops that cycle through all the bars in the chart to calculate various ratios or indices. Since a PerfScript is executed 4 times, it can take *several minutes* to complete processing if the DataSource has tens of thousands of bars. This is not an error!

11.4 Using PerfScripts

The controls for using PerfScripts are located at the top of the Performance View in the \$imulator and ChartScript Windows. The selections are independent between tools, and the last configuration in both is maintained for the next use.


Use a PerfScript

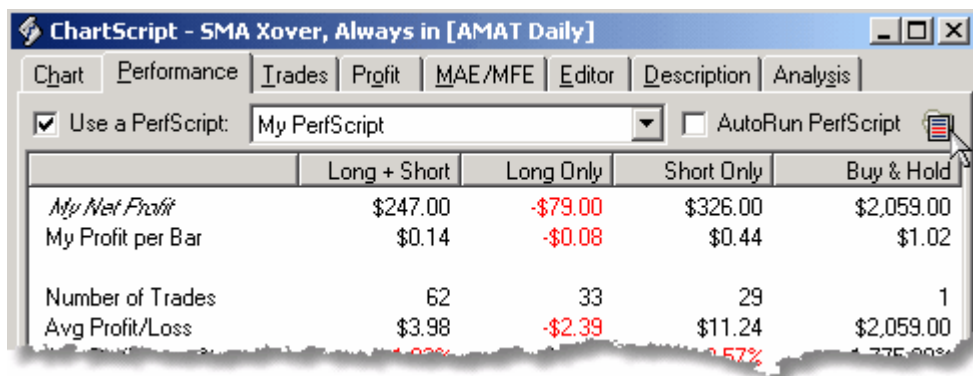
Check this option if you want to enable the use of the PerfScript selected in the drop down control immediately to the right. When selected, Wealth-Lab's usual performance reporting is disabled.

AutoRun PerfScript

Applies to: ChartScript Window

With "Use a PerfScript" selected, you can choose whether or not it is executed automatically (ChartScript Window only). When this option is not checked, the

Performance report will remain blank until you click the  PerfScript icon to run it on demand.



ChartScript Window PerfScript Control.
Click the icon to execute a PerfScript manually.

Tip: If you do not regularly look at the Performance Report, deselect "AutoRun PerfScripts" to optimize resources - especially when trading using Real-Time ChartScript Windows.

Usage Notes

Unless you need to calculate a metric that is not already included in the standard Wealth-Lab Performance Reports, there is no reason to even use PerfScripts. Due to the scripting "speed penalty", a sample PerfScript will take one to two orders longer to generate the same result as with the equivalent compiled code. Largely for this reason, you can uncheck AutoRun PerfScript so that you are not unnecessarily utilizing computer resources at times during system development and debugging when you're not likely to even look at the Performance View.


The "Script Timeout value" in **Options | System Settings | General Settings** does not pertain to PerfScripts. Consequently, you should give ample time for a PerfScript to complete its processing.

Warning! PerfScripts might take a long time (possibly several minutes) to compute on very large data sets.

12 SimuScripts

12.1 Overview

When you think "SimuScript", think "Position Sizing". Although Wealth-Lab Developer 4.0 provides four of the most popular position sizing methods for Portfolio \$imulations (Fixed Dollar/Margin, Fixed Share/Contract, Percent of Equity, and Maximum Risk Percent) you may have other ideas of how you would like to size your positions.

SimuScripts are an advanced feature of Wealth-Lab Developer 4.0 that let you experiment with your very own position-sizing rules in the \$imulator as well as in the ChartScript, Rankings, and Scans tools when Portfolio Simulation mode is selected. A SimuScript is a special type of ChartScript that must be stored in the  SimuScripts ChartScript folder.

[SimuScript Function Notes](#)^[110]

Only a subset of WealthScript functions are eligible for use in SimuScripts. However, SimuScripts have a special constant and dedicated functions that make it easy to write simple or complex algorithms to determine sizing for new positions.

[How do SimuScripts Work?](#)^[112]

The final result of a SimuScript will set the position size using one of three special SimuScript functions, which can size a position by percent of equity, fixed cash value, or by a specific number of shares.

[Creating a SimuScript](#)^[112]

In reality, a SimuScript is used like a procedure that is called each time your trading rules take a new position. If writing trading rules for ChartScripts is easy, then SimuScripts are almost child's play. A SimuScript can be as simple as one line of code!

[Testing a SimuScript](#)^[114]

Coding a SimuScript is arguably more simple than writing a ChartScript. Knowing a few more details about testing SimuScripts can make testing and debugging them simpler too.

Learn more about SimuScripts

A great way to learn more about SimuScripts is to review the SimuScript entries in the Function QuickRef. Each entry has a complete SimuScript example that will give you plenty of ideas. For a list of functions that are available see the SimuScript Functions topic in the WealthScript Function Reference^[5].

12.2 SimuScript Function Notes

SimuScripts support a subset of WealthScript functions, and include a collection of functions specific to position sizing. These include functions that return Portfolio Equity, Cash, DrawDown and many other values that may be useful in determining a position size. Availability of WealthScript functions for use in SimuScripts to include

the following categories of functions:

- Math Functions
- String Functions
- SimuScript-Specific Functions
- Data Access
- Date/Time
- File Access
- Indicators
- Position Management
- Price Series

Consequently, the following categories of functions *cannot* be used for SimuScripts:

- Alerts
 - Cosmetic Charts
 - System
 - Time Frame
 - Trading System
 - PerfScripts
 - CommissionScripts
-

SimuScript Use of BarCount

Generally speaking, SimuScript-specific functions that have WealthScript counterparts retain the same meaning when used in SimuScripts or in ChartScripts (e.g., [PositionLong](#), [PositionShort](#), etc.).

An exception worth noting is the slightly different meaning of the [BarCount](#) function when used in a SimuScript. While in a ChartScript [BarCount](#) returns the total number of bars in the chart, in a SimuScript the function returns the total number of bars *processed* at the time the SimuScript is executed. To return the current *Bar Number* on which the Position is being processed, use [BarCount](#) - 1 just as you do in ChartScripts.

The #Current Constant

In more complex SimuScripts you may want to retrieve data that are specific to the Position being processed. For example, you may have stored the value of an RSI indicator at the bar on which you entered the Position in your ChartScript using the [SetPositionData](#) function. In your SimuScript, you can access this data using the [GetPositionData](#) SimuScript function. You may then decide to take additional shares for more oversold values of RSI, for instance.

To recall the Position data that was stored for the Position *currently* being processed by the SimuScript, you pass the constant [#Current](#) to the [GetPositionData](#) function. In a similar way, you can use this constant for any variety of SimuScript functions that require a *Position* number as an argument.

Example

```
{ Risk half as many shares for short positions.
  Note: this is a complete SimuScript! }
if PositionShort(#Current) then
  SetPositionSizeShares( 100 )
else
```

```
SetPositionSizeShares( 200 );
```

12.3 How do SimuScripts Work?

Position sizing, no matter how simple, is an integral part of any trading system. If you do not wish to use one of the four position-sizing options offered by the Portfolio Simulator, you have the option to create a SimuScript to size your positions.

Select a specific SimuScript to use in the Portfolio Simulation control, which is a common control in both the Simulator and ChartScript windows. The selected SimuScript will be executed **once for each trade** generated during a simulation. You do not have to make a specific reference to a SimuScript in your ChartScript code. Wealth-Lab Developer 4.0 automatically executes the SimuScript whenever a "BuyAt" or "ShortAt" WealthScript function results in processing a new Position.

The goal of the SimuScript is to assign a position size to the current Position. The SimuScript does this by calling one of three functions during its execution:

SetPositionSizeFixed(*CashValue*);

Instructs the Portfolio Simulator to assign a fixed *CashValue* to a position. To eliminate a Position entirely, use this function by passing a zero value for *CashValue*.

SetPositionSizePct(*PercentOfEquity*);

Instructs the Portfolio Simulator to assign a percentage of total portfolio Equity to a position. To eliminate a Position entirely, use this function by passing a zero value for *PercentOfEquity*.

SetPositionSizeShares(*NumberOfShares*);

Instructs the Portfolio Simulator to assign a fixed number of shares to a position. To eliminate a Position entirely, use this function by passing a zero value for *NumberOfShares*.

Note: If your portfolio does not contain sufficient funds to acquire the full size of the position, the trade will not be placed. Your SimuScript can test for existing cash using the **Cash** function and reduce the position size, if desired, prior to calling one of the SetPositionSize functions.

The main thing to keep in mind when writing a SimuScript is that the script is processing only a **single Position**. The Portfolio Simulator calls the script one time for each Position that it needs to process.

12.4 Creating a SimuScript

You begin writing a SimuScript just as you would a normal ChartScript - starting with a New ChartScript Window. It's likely that you'll want to start fresh, so delete the template code in the ChartScript Editor if necessary. Only your position-sizing requirements and imagination can tell you how to proceed from this point. Your final SimuScript may be as simple as a single statement or even more complex than the ChartScript that will eventually use it!

Here we provide an example of a typical SimuScript with medium complexity. It

provides the same function as the Portfolio \$imulator's Maximum Risk Percent position-sizing model with an extra twist. It dynamically adjusts the percentage of risk based on the changing equity of a portfolio during a \$imulation. As equity grows, the SimuScript increases the percent of the equity risked on each trade, and vice versa. You can adjust the settings to your tastes by modifying the constant values and saving the script. Remember, all SimuScripts must be saved in the *SimuScripts* folder.

Example

```
{ SimuScript for increasing Percent Risk with growing Equity }
var fPctRisk, fEquity, CashSize: float;
var fStop, fBasis: float;
var Factor, FinalSize: integer;

{ These settings will increase the Risk by 0.2% for every $10,000 of
  equity growth }
const IncreaseRisk = 10000;
const RiskIncrement = 0.002;
const MinRiskCash = 75000;
const MinRisk = 0.005;      // Risk at least 0.5% on each trade
const MaxRisk = 0.06;       // Don't risk more than 6% on a single trade

{ Store values in variables for easy reference }
fEquity := Equity( BarCount - 1 );
fStop := GetPositionRiskStop( #Current );
fBasis := PositionBasisPrice( #Current );

if fEquity < MinRiskCash then
  fPctRisk := MinRisk
else
begin
  Factor := (fEquity - MinRiskCash) Div IncreaseRisk;
  fPctRisk := MinRisk + (RiskIncrement * Factor);
  if fPctRisk > MaxRisk then
    fPctRisk := MaxRisk;
end;

{ Calculate the size in shares, and then in cash }
FinalSize := Trunc( ( fEquity * fPctRisk ) / Abs( fBasis - fStop ) );
CashSize := FinalSize * fBasis;

{ If the position size is greater than the account equity,
  allow the trade to take place if fully in cash }
if CashSize > fEquity then
  FinalSize := Trunc( fEquity / fBasis );

SetPositionSizeShares( FinalSize );
```

Note the use of the function **GetPositionRiskStop** to retrieve the value of your stop level. To properly employ this SimuScript, you must use **SetPositionStopLevel** in your ChartScript code. Pass the price level of the initial stop immediately *before* entering a trade to this function. Only then can the SimuScript determine risk percentage with respect to your portfolio's equity level. See its QuickRef description for an example.

Note: **SetPositionStopLevel** supersedes **SetPositionRiskStop**.

Using this SimuScript on a winning system with Starting Capital of say, \$500,000, will yield the same results as the Portfolio \$imulator's Maximum Risk Percent with a 6% setting. With a losing system, this SimuScript could save you money!

See Also: [Only One Trade per Symbol](#) from the Wealth-Lab on-line articles archives.

12.5 Testing a SimuScript

Since SimuScripts only size positions and do not contain trading rules, it's not possible to know that they will function correctly by running the script by itself. They must be used in a Portfolio Simulation environment.

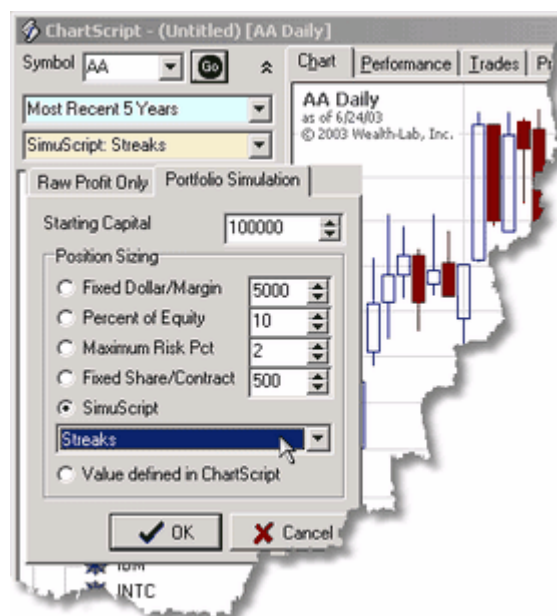
Guidelines to test and troubleshoot SimuScripts

1. Start by executing the SimuScript by itself. Although it's not likely that you can determine if the SimuScript sizing method functions in the manner in which you had intended, running the script gives you an opportunity to correct syntax errors.

Note: If using the **#Current** constant to refer to the current position being processed, you can expect the error, *List Index Out of Bounds (-1)*. At this point, the general syntax of the SimuScript is correct and you may proceed with system testing.

2. When your SimuScript's general syntax is correct, you're ready to test the SimuScript in the \$imulator or from another ChartScript using the common Position Sizing control in Portfolio Simulation mode as shown below.

Note: All functions are not equally available for SimuScripting in the ChartScript window as in the \$imulator. Refer to the WealthScript Function Reference^[5] or QuickRef for information on specific functions if in doubt.



Choosing a SimuScript for Portfolio Simulation mode in the ChartScript window.

3. Use the \$imulator or a ChartScript to build confidence that your SimuScript is functioning properly by initially testing one symbol only. After running the \$imulation, you can easily check to see if the first several trades are correctly sized by inspecting the Trades view.

4. If errors occur during a \$imulation, the Errors view will be shown automatically. Also, you'll likely see the message, "No Trades were generated by this \$imulation run". Check the Errors view for detailed information.

12.6 SimuScript FAQs

Can I use SimuScripts in the ChartScript Window?

Yes, however you cannot use a SimuScript that accesses Position data using `GetPositionData` from the ChartScript Window. See the description for `SetPositionData` for more information. If you need to pass Position data to a SimuScript in any tool other than the \$imulator, use `SetGlobal/GetGlobal`, or alternatively make use of the *SignalName* parameter of the `BuyAt` or `ShortAt` entry signals. In the latter case, retrieve the data with `PositionSignalName`.

Is it possible to use #OptVars in SimuScripts?

Not directly. Instead, you could write the current value of an `#OptVar` into global memory via `SetGlobal` at the start of ChartScript processing and retrieve the value in your SimuScript with `GetGlobal`.

I want to size differently according to the symbol. How?

Use `PositionSymbol` to test the `#Current` symbol. A [Case statement](#)^[30] is ideal here so that you can easily add different symbols to test.

Example

```
{* SimuScript *}
var sizeEqPct: float;

{ Assign sizing according to symbol }
case PositionSymbol( #Current ) of
  'GOOG':
    sizeEqPct := 8.0;
  'AAPL', 'MFST', 'INTC':
    sizeEqPct := 6.5;
  else
    sizeEqPct := 5.0; // 5% for any other symbol
end;
SetPositionSizePct( sizeEqPct );
```

How can I limit one Position per symbol?

Generally, ChartScripts are written to manage single Positions. But you may be dealing with a multi-Position script and want to analyze its return using a one-position-per-symbol strategy. A SimuScript would first need to determine if any *active Position* has the same symbol as the Positions currently being sized, and the solution is presented in the Knowledge Base:

[Allowing only one Trade per Symbol in the \\$imulator.](#)

I only want to allow 3 new entries per day. How?

Please see the [Max Entries per Day](#) SimuScript in the [Wealth-Lab Code Library](#).

13 Objects

13.1 Overview

WealthScript is a fully **object-oriented** scripting language, and support creation of **classes**, **inheritance**, and **polymorphism**. An *Object* is a type of variable that contains both *data items* and the *functions* and *procedures* (referred to collectively as *methods*) required to operate on the data.

To be sure, object-oriented programming (OOP) is not a trivial topic to grasp for novice programmers. Although the re-useable quality of objects make some tasks simple to accomplish, these techniques are not necessary to get a lot of mileage out of Wealth-Lab Developer 4.0. For those just becoming familiar with WealthScript and who are unfamiliar with OOP, mastering the use of the programming techniques described in the previous sections will provide you with plenty of capability in designing robust trading systems.

If you're already familiar with OOP, the topics in this chapter will introduce you to the proper WealthScript syntax to design, create, and destroy your objects. Visual Basic programmers familiar with OOP will discover the explicit object-declaration section that is hidden from them when creating their class objects, but otherwise the transition to using WealthScript objects should be quick. Please note that it is not our intention to teach OOP as many other in-depth resources are available on the subject.

Object Type Declarations^[118]

Generally speaking, *objects*, *types*, and *classes* are synonyms in programming terminology. A good part of the work in creating an object is declaring its parts.

Providing Access via Properties^[119]

Properties are those parts of which an object consists. Just as a car may be painted red, have a moon roof, and travel at 200 kph, an object has properties that define it. Depending on the manner in which you declare an object's properties, you can access or even change their values - just like you can change the color of a red car to blue.

Creating and Using Instances of a Type^[121]

Unlike red cars with moon roofs, it doesn't cost so much to create new objects. However, objects use memory and computer resources, consequently when they have served their purpose it's best to destroy them.

Putting it all Together^[122]

The complete script for the TProfitTracker is presented here with test code to put the object through its paces. Later, you can save the TProfitTracker object in the "Studies" folder and use it in any ChartScript that you wish by including it with the Include Manager.

Inheritance^[123]

You can create an object that *descends* from another one. The new object will inherit all of the variables, functions, procedures, and properties defined in the *ancestor*.

Polymorphism^[125]

You can create functions and procedures in a type that can change their behavior in descendant types. In object-oriented programming this type of feature is known as *polymorphism*.

The TList Object

Arrays are indispensable in programming, however, they may not always be the best choice for storing values of related items. The TList object is great when you don't know how many items you will be needing beforehand, therefore you may add to it, and remove from it, as you please. It's convenient since it takes care of all the "dimensioning" for you. Additionally, the TList has other properties and methods that would be very tedious to manage with plain vanilla arrays, and for this reason the TList is a good introduction to using objects - appropriate for even beginners!

13.2 Object Type Declarations

The Type Statement

You use the **type** statement to define a new type of *Object*. You can then create one or more instances of the object later in your code. The **type** statement contains three sections in which you can declare variables and functions/procedures.

private

Items declared in this section are available only to the Object's own functions or procedures.

protected

Items declared here are also available to Objects that are inherited from this Object.

public

Items declared in the public section are available anywhere.

Example

```
{ This is the skeleton for creating a new type of object
  When creating a new class, replace TMyObject with your class name }
type TMyObject = class
private
protected
public
end;
```

Variables in a Type

You can declare variables in any of the three sections in your type. Going forward our example will revolve around a new Object type that will know how to calculate and deliver information on the average profit generated per trade from your trading system.

Example

```
type TProfitTracker = class
private
  AvgProfit: integer;
protected
public
end;
```

Our new Object type, TProfitTracker, now contains one integer variable, AvgProfit. Notice that when you declare variables in any of the three sections of the **type** you

don't use the **var** statement.

Methods in a Type

Each of the three sections in a type can also contain functions or procedures, referred collectively as object *methods*. You declare the functions or procedures normally, then provide the implementation after the type declaration itself. Below we add a single new procedure to our type.

Note that the function is declared in the public section, and it is implemented after the end of the **type** statement. The syntax *TProfitTracker.Execute* lets WealthScript know that you're implementing the Execute procedure of the TProfitTracker type.

Also notice that the Execute procedure creates a new Price Series using **CreateSeries** (see [Creating Your Own Price Series](#)^[54]) and assigns it to the private variable "AvgProfit".

Example

```

type TProfitTracker = class
private
    AvgProfit: integer;
protected
public
    procedure Execute;
end;

procedure TProfitTracker.Execute;
var
    Bar, count, p: integer;
    profit: float;
begin
    AvgProfit := CreateSeries;
    for Bar := 0 to BarCount - 1 do
    begin
        count := 0;
        profit := 0;
        for p := 0 to PositionCount - 1 do
        begin
            if PositionExitBar( p ) <= Bar then
            begin
                Inc( count );
                profit := profit + PositionProfit( p );
            end;
        end;
        if count > 0 then
            SetSeriesValue( Bar, AvgProfit, profit / count );
        end;
    end;
end;

```

13.3 Providing Access via Properties

What are Properties?

Properties are a special feature of *objects*. A property can provide read-only or read-write access to data within its Object. You declare a property in any of the three type

sections (although it usually makes the most sense to declare them in the public section).

A property can be given a **read accessor** that specifies a function or variable to use to obtain the property's value. It can also be given a **write accessor** to specify a procedure to use for storing the property's value.

If a read accessor function is declared for a property, then whenever the property is referenced in code, the value is obtained by executing the read accessor function (or grabbing the value from the variable). Similarly, if a write accessor procedure is declared for a property, whenever the property is assigned a value, the value is passed through the write accessor procedure.

Our TProfitTracker creates a new Price Series, but stores it in a private variable. We can provide read-only access to this variable by creating a property that returns the value from the variable. Then, anyone using this object will be able to access the AvgProfit Price Series but will not be able to modify it.

Example

```
type TProfitTracker = class
private
  AvgProfit: integer;
protected
public
  procedure Execute;
  property AvgProfitSeries: integer read AvgProfit;
end;
```

Declaring Accessor Methods

The following sample creates a new type called TSample. TSample contains a single integer property, "Sample", with a read and write accessor methods. It also contains a private integer variable called "FSample" that stores that property value internally. This variable is often called the "Field variable" and is conventionally named the same as the property but preceded by an "F". The read accessor method simply returns the value from the FSample variable. Note that we could have eliminated the read accessor method in this case and replaced it with the variable itself as follows:

```
property Sample: integer read FSample write SetSample;
```

The write accessor, however, performs some special processing on the incoming value. It restricts the value to be within the range of 0 to 100 before assigning it to the underlying FSample variable.

```
type TSample = class
private
  FSample: integer;
  function GetSample: integer;
  procedure SetSample( n: integer );
protected
public
  property Sample: integer read GetSample write SetSample;
end;

function TSample.GetSample: integer;
begin
  Result := FSample;
end;
```

```
procedure TSample.SetSample( n: integer );
begin
    FSample := n;
    if FSample > 100 then
        FSample := 100;
    if FSample < 0 then
        FSample := 0;
end;
```

13.4 Creating and Using Instances of a Type

Creating an Object Instance

Now that we know how to declare types of Objects, we need to learn how to create instances of these types. You can create one or more instances of a type in your WealthScript code. Each instance maintains its own internal copy of any data elements declared within the type.

To create an instance of an Object you must first declare a variable to store the instance using a standard var statement. You can then create the instance of the Object using a new type of statement called the *constructor*. The *constructor* is simply the name of the type followed by a ".Create".

Example

```
var AProfitTracker: TProfitTracker;
AProfitTracker := TProfitTracker.Create;
```

Note that we first declared a variable of the type "TProfitTracker". We then assigned a value to the variable using the new constructor style statement.

If required, you can take advantage of the Create constructor method to initialize your object. The way to do it is to create your own Create constructor. The code you put in the constructor's method will be called whenever an instance of the class is created. The TProfitTracker class does not require a special initialization method, but we include the following example for completeness.

Example

```
type MyClass = class
    constructor Create;
end;

constructor MyClass.Create;
begin
    ShowMessage( 'Instance Created' );
end;

var Instance: MyClass;
Instance := MyClass.Create;
```

Freeing Instances

In Wealth-Lab Developer 4.0, WealthScript employs the programming concept of a *garbage collection* to clean up objects that are no longer being accessed. Destructors

need not be used and objects are freed automatically when they are no longer referenced, or at the end of a script. Consequently, there is no need to explicitly free, or destroy, object instances that you create.

Note: In previous versions of Wealth-Lab you were responsible for destroying object instances by calling their *Free* method. The *Free* method is no longer required, and you should remove calls to the *Free* method in your scripts.

Accessing Properties of Objects

Once you have one or more instances of your Object created, you can access their properties. Use the "Variable.Property" *dot-style* notation to access an object's properties.

Example

```
{ Access the Average Profit Series }  
var n: integer;  
n := AProfitTracker.AvgProfitSeries;  
  
{ This will trigger an error, since we didn't define a write accessor  
for the property }  
AProfitSeries := 0;
```

Executing Methods of an Object

Use the same dot-style notation to execute any functions or procedures defined within an Object's type.

Example

```
{ Tell the object to do its thing }  
AProfitTracker.Execute;
```

13.5 Putting it all Together

Below is the complete script for the TProfitTracker, and some test code to put the Object through its paces. Note that we've made the TProfitTracker more intelligent. The Object now tracks whether or not the average profit per trade Price Series was constructed using a private boolean variable "bExecuted". It then uses the read accessor method to construct the Price Series by calling the Execute method if required.

Example

```
type TProfitTracker = class  
private  
    FAvgProfitSeries: integer;  
    bExecuted: boolean;  
protected  
    function GetAvgProfit: integer;  
public  
    procedure Execute;  
    property AvgProfitSeries: integer read GetAvgProfit;
```

```

end;

function TProfitTracker.GetAvgProfit: integer;
begin
    if not bExecuted then
        Execute;
        Result := FAvgProfitSeries;
    end;

procedure TProfitTracker.Execute;
var
    Bar, count, p: integer;
    profit: float;
begin
    bExecuted := true;
    FAvgProfitSeries := CreateSeries;
    for Bar := 0 to BarCount - 1 do
        begin
            count := 0;
            profit := 0;
            for p := 0 to PositionCount - 1 do
                begin
                    if PositionExitBar( p ) <= Bar then
                        begin
                            Inc( count );

                            profit := profit + PositionProfit( p );
                        end;
                end;
            if count > 0 then
                SetSeriesValue( Bar, FAvgProfitSeries, profit / count );
            end;
        end;

{ A simple channel breakout system to test the object }
var Bar: integer;
for Bar := 4 to BarCount - 1 do
    begin
        if LastPositionActive then
            SellAtStop( Bar + 1, Lowest( Bar, #Low, 3 ), LastPosition, '' )
        else
            BuyAtStop( Bar + 1, Highest( Bar, #High, 3 ), '' );
        end;

{ Use the TProfitTracker object now }
var AProfitTracker: TProfitTracker;
var AvgProfitPane: integer;
AProfitTracker := TProfitTracker.Create;
AvgProfitPane := CreatePane( 100, true, true );
SetPaneMinMax( AvgProfitPane, 0, 0 );
PlotSeries( AProfitTracker.AvgProfitSeries, AvgProfitPane, #Green,
#ThickHist );

```

13.6 Inheritance

Deriving One Type from Another

You can create an object that *descends* from another one. The new object will inherit

all of the variables, functions, procedures, and properties defined in the *ancestor*. The new object will be able to access all of the items declared in the protected or public section of the ancestor, but not from the private section.

To specify that a type is derived from a parent, place the type of the ancestor in parenthesis after the type name in the type statement:

Example

```
type Ancestor = class
private
protected
public
end;

type TDescendant = class( TAncestor )
private
protected
public
end;
```

TObject Type

Actually, all types ultimately descend from a base type called TObject. TObject provides the default *constructor* and *destructor*. The system assumes that new types are derived from TObject even when no ancestor type is provided.

Example

```
{ The following 2 type statements are identical }
type TMyType = class
private
protected
public
end;

type TMyType = class( TObject )
private
protected
public
end;
```

Descendant Types Can Access Protected Items

You can access variables, functions and procedures that were declared in the protected section of an ancestor type from within the functions and procedures of the derived type.

Example

```
type TMyType = class
private
    var1: integer;
protected
    var2: integer;
public
    var3: integer;
end;
```

```

type TMyType2 = class( TMyType )
private
protected
public
    function GetResult: integer;
end;

function TMyType2.GetResult: integer;
begin
    Result := var3;      {Public ... this is legal}
    Result := var2;      {Protected ... this is legal}
    Result := var1;      {Private ... NOT ACCESSABLE}
end;

```

13.7 Polymorphism

Polymorphic Methods

You can create functions and procedures in a type that can change their behavior in descendant types. In object-oriented programming this type of feature is known as *polymorphism*. To flag a function or procedure as being polymorphic, add the keyword **virtual** after the declaration. Then, in your derived class, re-declare the function or procedure with the **override** keyword.

In this example we create a type that returns the average price at any given bar by adding the high and low and dividing by two. We then created an inherited type that changes the implementation of the function by factoring closing price into the calculation. The code at the bottom of the script illustrates the polymorphic behavior. We declare a variable of the type of the ancestor type (often called the *base class*), but use the *constructor* of the descendant type when creating the instance of the object. So, even though the object is stored in a variable type of the ancestor, it uses the descendant's function implementation when calculating the average price.

Example

```

type TAverager1 = class
private
protected
public
    function GetAvg( Bar: integer ): float; virtual;
end;

type TAverager2 = class( TAverager1 )
private
protected
public
    function GetAvg( Bar: integer ): float; override;
end;

function TAverager1.GetAvg( Bar: integer ): float;
begin
    Result := ( PriceHigh( Bar ) + PriceLow( Bar ) ) / 2;
end;

function TAverager2.GetAvg( Bar: integer ): float;
begin
    Result := ( PriceHigh( Bar ) + PriceLow( Bar ) + PriceClose( Bar ) )

```



```

/ 3;
end;

var Avg: TAverager1;
Avg := TAverager2.Create;

Print( FloatToStr( Avg.GetAvg( 0 ) ) );

```

Accessing the Inherited Behavior

Your polymorphic procedures and functions can access the behavior of the ancestor by using the **inherited** keyword. Here we change the implementation of the `GetAvg` function of the descendant class to access and then modify the result from the ancestor's function.

Example

```

function TAverager2.GetAvg( Bar: integer ): float;
begin
    Result := inherited GetAvg( Bar );
    Result := ( Result + PriceClose( Bar ) ) / 2;
end;

```

13.8 The TList Object

13.8.1 Overview

The `TList` class provides a list object. You can add and remove items to the list, as well as sort the items. You can access the items in the list by index number. The first item in the list is index zero, and the last item is index `Count - 1`.

A `TList` is great when you don't know how many items you will be needing beforehand, therefore you may add to it and remove from it as you please. It's convenient since it takes care of all the "dimensioning" for you. However, with these advantages, you will pay a small performance penalty in speed when compared to accessing an [array](#)^[45].

`TList` stores data items as [Variants](#)^[11], which is a special data type that can be used to store any other basic type, such as `string` or `float`. Consequently, items retrieved using the *Item* and *Data* methods from the `TList` are of type variant. You can also use the `TList` to store a collection of object types using the *AddObject* method. To retrieve the instance of an object, use the *Object* method.

The `TList` object is not available for use in SimuScripts.

The following example, which stores all of the closing values in the chart into a `TList` object, demonstrates the use of many of the `TList` methods. In the example, the `TList` object sorts its members, and finally, the sorted closing values are written to the Debug Window by iterating through the list.

Example

```

{ Declare Variables }
var Bar: integer;
var lst: TList;

```

```

var f: float;

{ Create an instance of a TList }
lst := TList.Create;

{ Fill the TList with Closing Values of the Chart }
for Bar := 0 to BarCount - 1 do
begin
    f := PriceClose( Bar );
    lst.Add( f );
end;
{ Sort the values }
lst.SortNumeric;

{ Print the sorted values to the Debug Window
  Note: here, lst.Count is equal to BarCount }
for Bar := 0 to lst.Count - 1 do
begin
    f := lst.Item( Bar );
    Print( FormatFloat( '#,###.00', f ) );
end;

```

You can also pass a TList to a procedure. When doing this, the TList object is passed *by reference* to the procedure. This means that any changes (Add, Delete, Clear, etc.) made to the the TList in the procedure will also effect the TList object in the calling procedure as demonstrated in the next example.

Example

```

function MySum(aTL: TList): integer;
var n: integer;
begin
    Result := 0;
    for n := 0 to aTL.Count - 1 do
        Result := Result + aTL.Item( n );
    { Delete the last item in the TList }
    aTL.Delete( aTL.Count - 1 );
end;

var lst: TList;
var isum: integer;

{ Program execution begins here }
lst := TList.Create;
lst.Add( 3 );
lst.Add( 5 );
lst.Add( 8 );
lst.Add( 13 );

isum := MySum( lst );
ShowMessage( 'The sum of the TList is ' + IntToStr( isum ) );
ShowMessage( 'The list now has ' + IntToStr( lst.Count ) + ' items' );

```

13.8.2 TList Functions

13.8.2.1 Add

The **Add** method returns an **integer** index of the added *Value*.

Syntax

```
object.Add( Value );
```

Item	Description
<i>object</i>	An object expression of type TList.
<i>Value</i>	Variant. A variable or expression of any of the primitive data types ^[11] .

Remarks

- Adds the specified item, *Value*, to the list.
- Returns the index number of the added *Value*. The **Add** method returns the index 0 for the first item added to a TList.
- Use the [Item](#)^[131] method with the integer index returned by the **Add** method to retrieve a *Value* in a TList.
- If the [Delete](#)^[136], [SortNumeric](#)^[137], or [SortString](#)^[137] methods are used after adding an item to a TList, it's likely that the index of the item returned by the **Add** will change.

Note:

You may implement this method as shown in the The TList Object [example](#)^[126], or alternatively by assigning the function to an integer variable.

FAQ: How can I add a [record type](#)^[12] to a TList?

You cannot add a record to a list, but you can add an object, which can contain different data elements just like a record type. For more information, see the [AddObject](#)^[129] method of a TList.

13.8.2.2 AddData

The **AddData** method returns an **integer** index of the added *Value* and associated *Data*.

Syntax

```
object.AddData( Value, Data );
```

Item	Description
<i>object</i>	An object expression of type TList
<i>Value</i>	Variant. A variable or expression of any of the primitive data types ^[11] .
<i>Data</i>	Variant. A variable or expression of any of the primitive data types ^[11] .

Remarks

- Returns the index number of the added *Value*.
- Adds the specified item, *Value*, to the list along with additional *Data*.
- Use the [Item](#)^[131] method with the integer index returned by the **AddData** method to

retrieve a *Value* in the TList.

- Access the *Data* at a later time using the [Data](#)^[137] method with the integer index returned by the **AddData** method.
- If the [Delete](#)^[136], [SortNumeric](#)^[137], or [SortString](#)^[137] methods are used after adding an item to a TList, it's likely that the index of the item returned by the **AddData** will change.

Tip:

You can easily store more than one value in either the *Value* or *Data* fields by using a delimited string variable or expression as shown in the example below. Later, you must parse the string to retrieve the individual values.

This example demonstrates how to stores a TList of 8% peaks containing the peak value as well as its date and bar number, which are stored as *Data* in the form of a comma delimited string.

Example:

```
var lst: TList;
var Bar, PkSe, i, dte: integer;
var f, fP: float;

lst := TList.Create;

{ Obtain a series of 8% Peaks and plot them on the chart }
PkSe := PeakSeries( #High, 8 );
PlotSeries( PkSe, 0, #Red, #Dots );

f := 0.0;
for Bar := 1 to BarCount - 1 do
begin
  { if a new peak is detected, add it to the list with its date value }
  dte := GetDate(Bar);
  fP := @PkSe[Bar];
  if f <> fP then begin
    lst.AddData( fP, IntToStr(dte) + ',' + IntToStr(Bar) );
    f := fP;
  end;
end;

{ Print the peak number, peak value, date, and bar
to the debug window }
for i := 0 to lst.Count - 1 do
  Print( 'Peak #' + IntToStr( i + 1 ) + ': ' +
    Chr(9) + FormatFloat('#.00', lst.Item( i )) +
    Chr(9) + lst.Data( i ) );
```

See Also:

[Item Method](#)^[137], [Data Method](#)^[137], Peak Indicator, Plot Series

13.8.2.3 AddObject

The **AddObject** method returns an **integer** index of the added *Value* and associated instance of *Object*.

Syntax

```
obj.AddObject( Value, Object );
```

Item	Description
<i>obj</i>	An object expression of type TList
<i>Value</i>	Variant. A variable or expression of any data type or an instance of an object.
<i>Object</i>	TObject. An instance of an Object type ^[118] to store in the TList.

Remarks

- Returns the index number of the added *Value*.
- Adds the specified item, *Value*, to the list along with the specified instance of *Object*.
- Use the [Item](#)^[131] method to retrieve the *Values* of the TList.
- Access the *Object* at a later time using the [Object](#)^[134] method with the integer index returned by the **AddObject** method.

Typical usage

See [Object method](#)^[134] example

13.8.2.4 Count

The **Count** method returns an **integer** of the number of items held in the list.

Syntax

```
object.Count( );
```

Item	Description
<i>object</i>	An object expression of type TList

Remarks

- Returns the the total number items that a currently held in the TList specified by *object*.

Typical usage

See [The TList Object example](#)^[126]

13.8.2.5 Create

The **Create** method returns an instance of a TList object

Syntax

```
TList.Create;
```

Remarks

- Creates an *instance* of a TList object.
- See this [TList Object Example](#)^[126] for typical usage.

13.8.2.6 Data

The **Data** method returns a **variant** data value that was stored in the list via [AddData](#)^[128].

Syntax

```
object.Data( Index );
```

Item	Description
<i>object</i>	An object expression of type TList.
<i>Index</i>	Integer variable or expression identifying the index of the data item in the TList.

Remarks

- Returns the data value that was stored in the list via [AddData](#)^[128].
- The item is returned as a **variant** data type, but you can assign this to a variable of the appropriate data type. For example, if the data were stored as 'AMGN', you cannot assign this to an **integer** or **float** type. Rather, it should be assigned to a **string** or another **variant**. On the other hand, if the data were stored as '34.22', you may assign this to a variable of type **float**, **string**, or **variant**. In the last case, you could also assign the **variant** type to an **integer** type (the variant number would be rounded when assigned), but you must be careful when making such assignments due to the possibility of an overflow.

Typical usage

See [AddData](#)^[128] example

13.8.2.7 Item

The **Item** method returns a **variant** value that was stored in the list as *Value* via [Add](#)^[127] or [AddData](#)^[128].

Syntax

```
object.Item( Index );
```

Item	Description
<i>object</i>	An object expression of type TList.
<i>Index</i>	Integer variable or expression identifying the index of the TList value.

Remarks

- Returns the value that was stored in the list via [Add](#)^[127] or [AddData](#)^[128].
- The item is returned as a **variant** data type, but you can assign this to a variable of the appropriate data type. For example, if the data were stored as 'AMGN', you cannot assign this to an **integer** or **float** type. Rather, it should be assigned to a **string** or another **variant**. On the other hand, if the data were stored as '34.22', you may assign this to a variable of type **float**, **string**, or **variant**. In the last case, you could also assign the **variant** type to an **integer** type (the variant number

would be rounded when assigned), but you must be careful when making such assignments due to the possibility of an overflow.

Typical usage

See [The TList Object](#)^[126] or [AddData](#)^[128] examples

13.8.2.8 IndexOf

The **IndexOf** method returns an **integer** value that is the index in the list for the item specified in the *Value* parameter.

Syntax

object.**IndexOf**(*Value*);

Item	Description
<i>object</i>	An object expression of type TList.
<i>Value</i>	Variant variable or expression identifying the value to be found in the TList.

Remarks

- Returns the index value for the specified Value. Values are added to the TList via the [Add](#)^[127], [AddData](#)^[128], or [AddObject](#)^[129] functions.
- The first item in the list has an index value of zero, and the last item has an index value of *object.Count* - 1.
- If the specified Value could not be found in the list, the function returns -1.

Example

```
var lst: TList;
var n: integer;

{ Create TList }
lst := TList.Create;

{ Fill list with text strings }
lst.Add( 'Zero' );
lst.Add( 'One' );
lst.Add( 'Two' );
lst.Add( 'Three' );
lst.Add( 'Four' );
lst.Add( 'Five' );

{ Sort the list }
lst.SortString;

{ Find the index of the specified string, will be last in the list
after alpha sort }
n := lst.IndexOf( 'Zero' );
ShowMessage( IntToStr( n ) );
```

13.8.2.9 IndexOfData

The **IndexOfData** method returns an **integer** value that is the index in the list for the

secondary data item specified in the *Value* parameter.

Syntax

```
object.IndexOfData( Value );
```

Item	Description
<i>object</i>	An object expression of type TList.
<i>Value</i>	Variant variable or expression identifying the secondary data value to be found in the TList.

Remarks

- Returns the index value for the specified secondary data Value. Secondary data Values are added to the TList via the [AddData](#)^[128] function.
- The first item in the list has an index value of zero, and the last item has an index value of *object.Count* - 1.
- If the specified secondary data Value could not be found in the list, the function returns -1.

Example

```
var lst: TList;
var n: integer;

{ Create TList }
lst := TList.Create;

{ Fill with symbols and PE ratios }
lst.AddData( 12.5, 'MSFT' );
lst.AddData( 17.6, 'GE' );
lst.AddData( 6.7, 'MCD' );
lst.AddData( 2.1, 'CSCO' );
lst.AddData( 8.4, 'SUNW' );
lst.AddData( -12.7, 'AOL' );

{ Find the PE for MSFT }
n := lst.IndexOfData( 'MSFT' );
ShowMessage( FloatToStr( lst.Item( n ) ) );
```

13.8.2.10 IndexOfObject

The **IndexOfObject** method returns an **integer** value that is the index in the list for the object instance specified in the *Value* parameter.

Syntax

```
obj.IndexOfObject( Value );
```

Item	Description
<i>obj</i>	An object expression of type TList.
<i>Value</i>	An instance of an object type ^[118] to be found in the TList.

Remarks

- Returns the index value for the specified object instance. Objects are added to the TList via the [AddObject](#)^[129] function.
- The first item in the list has an index value of zero, and the last item has an index value of *obj.Count* - 1.
- If the specified object instance could not be found in the list, the function returns -1.

Typical usage

See [IndexOfData](#)^[132] example

13.8.2.11 Object

The **Object** method returns the object instance that was previously added to a TList by the [AddObject](#)^[129] method.

Syntax

obj.**Object**(*Index*);

Item	Description
<i>obj</i>	An object expression of type TList
<i>Index</i>	Integer variable or expression identifying the index of the TObject in the list.

Remarks

- When retrieving the TObject of *Index* from the TList, use the **as** operator to convert the return value to its original class.

Example

```

type TMyObject = class( TObject )
private
protected
public
    procedure Shout;
end;

procedure TMyObject.Shout;
begin
    ShowMessage( 'Arrrggghhh!' );
end;

var lst: TList;
var mo: TMyObject;
lst := TList.Create;
mo := TMyObject.Create;
lst.AddObject( 123.45, mo );
mo := lst.Object( 0 ) as TMyObject;
mo.Shout;

```

13.8.3 TList Procedures

13.8.3.1 Changeltem

Syntax

object.**ChangeItem**(*Index*, *Value*);

Item	Description
<i>object</i>	An object expression of type TList.
<i>Index</i>	Integer variable or expression identifying the index of the item to change in the TList.
<i>Value</i>	Variant variable or expression of the new <i>Value</i> to be stored in the TList at <i>Index</i> .

Remarks

- Changes the initial value that was stored in the list via [Add](#)^[127] or [AddData](#)^[128] to a new *Value*.

Note

The example demonstrates that **ChangeItem** operates on equally well on items added through [Add](#)^[127] or [AddData](#)^[128]. You normally create TLists that are *collections* of closely-related items, and therefore you should use *either* [Add](#)^[127] or [AddData](#)^[128] throughout the TList. Otherwise, attempting to access non-existent data could lead to unpredictable results.

Example

```
var lst: TList;
var i1, i2: integer;

lst := TList.Create;

i1 := lst.Add( 'SUNW' );
i2 := lst.AddData( 'AMGN', 9.15 );
Print( lst.Item(i1) );
Print( lst.Item(i2) + ', ' + FloatToStr(lst.Data(i2)) );

{ Whoops, I meant CSCO! }
lst.ChangeItem( i1, 'CSCO' );
lst.ChangeItem( i2, 'CSCO' );
Print( lst.Item(i1) );
Print( lst.Item(i2) + ', ' + FloatToStr(lst.Data(i2)) );

lst.Free;
```

13.8.3.2 Clear

Syntax

object.**Clear**();

Item	Description
<i>object</i>	An object expression of type TList

Remarks

- Clears the contents of the list

Note

If you attempt to access a *non-existent* TList item or data, immediately following the **Clear** method for example, an *out of bounds* error will occur.

13.8.3.3 Delete**Syntax**

object.**Delete**(*Index*);

Item**Description**

object

An object expression of type TList.

Index

Integer variable or expression identifying the index of the item to delete in the TList.

Remarks

- Deletes the item in the list specified by *Index*.
- Following the **Delete** method, the indices of all TList items that appear after the deleted item are decremented by one.

Example

```
var lst: TList;
var i: integer;
var symbol: string;

lst := TList.Create;

lst.Add('SUNW');
lst.Add('T');
lst.Add('BA');
lst.Add('MSFT');
lst.Add('GM');

{ Find 'BA' in the list and Delete it }
for i := 0 to lst.Count - 1 do
begin
    if lst.Item( i ) = 'BA' then begin
        lst.Delete( i );
        break;    // break out of loop
    end;
end;

{ Print the list in the debug window }
for i := 0 to lst.Count - 1 do
    print( lst.Item( i ) );

lst.Free;
```

13.8.3.4 Free**Syntax**

object.**Free**;

Item	Description
------	-------------

<i>object</i>	An object expression of type TList.
---------------	-------------------------------------

Remarks

- Destroys the TList object to free resources previously allocated to the TList *object*.
- Due to the introduction of the [garbage collection](#)^[127] in Wealth-Lab Developer 4.0, it is no longer necessary to explicitly destroy objects, such as TLists, through the use of the **Free** method.

13.8.3.5 SortNumeric**Syntax**

```
object.SortNumeric( );
```

Item	Description
------	-------------

<i>object</i>	An object expression of type TList
---------------	------------------------------------

Remarks

- Sorts the values in the list as numbers from least to greatest.
- You should ensure that integers or floats were added to the list, otherwise the results could be unpredictable.

Example

```
{ create a list of ascending closing prices of the last 10 chart bars }
var lst: TList;
var Bar, n: integer;

lst := TList.Create;
for Bar := 0 to BarCount - 1 do
    lst.Add( PriceClose( Bar ) );

lst.SortNumeric;

Print( 'Ascending' );
for n := 0 to lst.Count - 1 do
    Print( FormatFloat( '#.00', lst.Item( n ) ) );

Print( ' ' );
Print( 'Descending' );
for n := lst.Count - 1 downto 0 do
    Print( FormatFloat( '#.00', lst.Item( n ) ) );
```

13.8.3.6 SortString**Syntax**

```
object.SortString( );
```

Item	Description
------	-------------

*object*An object expression of type TList

Remarks

- Sorts the values in the list as strings.
- The sort order is determined by a case-sensitive string comparison (binary compare) of all items in the list, from least to greatest.

In the example, the string 'ba' will be sorted to the end of the list since lowercase characters have greater ASCII codes than uppercase characters.

Example

```
var lst: TList;
var i: integer;
var symbol: string;

lst := TList.Create;

lst.Add('SUNW');
lst.Add('ba');
lst.Add('BA');
lst.Add('MSFT');
lst.Add('GM');

lst.SortString;

{ Print the list in the debug window }
for i := 0 to lst.Count - 1 do
    print( lst.Item( i ) );

lst.Free;
```

Index

- # -

#All constant 79
 #AsDollar 76
 #AsPercent 76
 #AsPoint 76
 #Average 50
 #AverageC 50
 #Bold 106
 #Close 50
 #Color 69
 #ColorBkg 69
 #Current constant 110
 #Dots (dotted line style) 67
 #Dotted (dotted line style) 67
 #Equity 106
 #High 50
 #Histogram (histogram plot style) 67
 #Italic 106
 #Low 50
 #Open 50
 #Thick (thick line style) 67
 #ThickHist (thick histogram plot style) 67
 #Thin (thin line style) 67
 #Volume 50
 #WinLoss 106

- @ -

@ syntax 57
 GetSeriesValue 57
 SetSeriesValue 57

- A -

AnnotateBar 70
 AnnotateChart 70
 ApplyAutoStops 76
 arrays 45
 accessing 45
 array 45
 declaring 45
 multi-dimensional 45
 synchronized 45
 AutoRun PerfScript 108

- B -

BarCount 54
 Use in SimuScript 110
 boolean 11
 break 34
 BuyAtClose 74
 BuyAtLimit 75
 BuyAtMarket 74
 BuyAtStop 75
 by reference 40
 by value 40

- C -

casting 13
 chart 64
 painting 64
 panes 65
 plotting 64
 ChartScript Editor 5
 closing positions 79
 CMDDataSource 103
 CMEntry 103
 CMOrderType 103
 CMPPrice 103
 CMResult 103
 CMShares 103
 CMSymbol 103
 colors 69
 specifying 69
 COM Support 5
 combining positions 79
 comments 8
 CommissionScript 103
 CMDDataSource 103
 CMEntry 103
 CMOrderType 103
 CMPPrice 103
 CMResult 103
 CMShares 103
 CMSymbol 103
 compatibility 103
 variables 103
 creating 104
 testing 104
 CommissionScripts Overview 103
 constants 16
 # 16
 declaring 16

constants 16
 pre-defined 16
constructor 121
CreatePane 65

- D -

datetime 11
declaring 14
delimiters 8
drawing 70
 objects (programmatically) 70
DrawLabel 70
DrawText 70

- E -

enumerated types 13
error 44
 handling 44
exceptions 44
exit 43

- F -

FAQs 115
 SimuScripts 115
float 11
for (looping statement) 32
Free 121
functions 35
 arguments 40
 calling 39
 declaring 37
 executing 39
 parameters 40
 syntax 37

- G -

garbage collection 121
GetSeriesValue 55

- H -

handle 51
HidePaneLines 65

- I -

indicator 96
 custom 96
inheritance 123
InstallBreakEvenStop 76
InstallProfitTarget 76
InstallReverseBreakEvenStop 76
InstallStopLoss 76
InstallTimeBasedExit 76
InstallTrailingStop 76
instances of objects 121
integer 11

- K -

Knowledge Base 79

- L -

LastLongPositionActive 79
LastPosition 79
LastShortPositionActive 79
looping statements (summary) 32

- M -

Max Entries per Day 115
merging positions 79

- N -

New Indicator Wizard 96

- O -

object oriented programming 117
objects 117
 accessing properties 121
 constructor 121
 creating 121
 declaring 118
 freeing 121
 functions and procedures 118
 inheritance 123
 instances 121
 methods 118
 overview 117

- objects 117
 - polymorphism 125
 - properties 119
 - read accessor 119
 - variables 118
 - write accessor 119
- OOP 117
- operations 18
 - boolean 19
 - logical 21
 - mathematical 18
 - string 25
- operator 18
 - and 21
 - div 18
 - modulo 18
 - not 25
 - or 22
 - standard 18
 - xor 24
 - assignment 14
- orders 75
 - limit 75
 - market 74
 - market-on-close 74
 - selling short 77
 - stop 75
- Overview 103
 - CommissionScripts 103

- P -

- painting the chart 64
- panes 65
 - creating 65
 - hide lines 65
 - hide volume 65
- peeking 78
- PerfScript 106
 - constants 106
 - creating 107
 - errors 107
 - functions 106
 - Overview 106
 - using 108
- PlotSeries 67
- PlotSymbol 68
- PlotSyntheticSymbol 68
- plotting 64
 - external symbols 68
 - indicators 67

- objects (programmatically) 70
- style 67
- synthetic symbols 68
- polymorphism 125
- position sizing 110
- PositionActive 79
- PositionCount 79
- PositionEntryBar 79
- PositionEntryPrice 79
- PositionLong 79
- positions 75
 - closing 75, 79
 - combining 79
 - merging 79
 - multiple 79
 - open 75
 - split 79
 - splitting 79
- Price Series 48
 - accessing values 61
 - accessing values from 55
 - alignment 91
 - characteristics 48
 - constant handles 50
 - creating 54
 - expanding 91
 - external 61
 - functions that accept 52
 - handle 51
 - overview 48, 49
 - standard 50
 - synchronization 91
- procedures 35
 - arguments 40
 - calling 39
 - declaring 36
 - executing 39
 - parameters 40
 - syntax 36

- R -

- record types 12
- recursion 37
- recursive functions 37
- reentrant functions 37
- repeat (looping statement) 34
- return values (functions) 37

- S -

- Scale 65
- scripting 72
 - main loop 73
 - overview 72
 - trading rules 72
- SellAtClose 75
- SellAtLimit 75
- SellAtMarket 75
- SellAtStop 75
- semicolon 8
- Series Math 59
 - answers 59
 - practice 58
- SetAutoStopMode 76
- SetDescription 96
- SetSeriesBarColor 67
- SetSeriesValue 55
- SimuScript 110
 - #Current 110
 - BarCount 110
 - coding 112
 - creating 112
 - errors 114
 - Functions 110
 - how they work 112
 - testing 114
- SimuScripts 110
 - FAQs 115
 - Overview 110
 - Portfolio \$imulator 110
 - position sizing 110
- slash 8
 - double 8
- splitting positions 79
- Stability of Indicators 73
- state machines 13
- statements 8, 26
 - break 34
 - case 30
 - conditional 26
 - for loop 32
 - if/then 26
 - if/then/else 26
 - repeat loop 34
 - while loop 33
- stops 76
 - automated 76
- string 11, 25

- # shorthand 25
- Chr 25
- comparison 25
- style 67
- syntax 7
 - @ symbol 57

- T -

- TList 126
- TList methods 126
 - Add 127
 - AddData 128
 - AddObject 129
 - ChangeItem 134
 - Clear 135
 - Count 130
 - Create 130
 - Data 131
 - Delete 136
 - Free 136
 - IndexOf 132
 - IndexOfData 132
 - IndexOfObject 133
 - Item 131
 - Object 134
 - SortNumeric 137
 - SortString 137
- TProfitTracker 122
- trading rules 72
 - implementation 78
 - looking ahead 78

- U -

- Use a PerfScript 108

- V -

- variables 9
 - assigning 14
 - data types 11
 - declaring 10
 - enumerated types 13
 - initializing 14
 - naming rules 10
 - record types 12
 - scope 42
- variant 11
- Volume 65
 - hide pane 65

- W -

WealthScript 5
 definition 5
 AnnotateBar 70
 AnnotateChart 70
 ApplyAutoStops 76
 BarCount 54
 BuyAtClose 74
 BuyAtLimit 75
 BuyAtMarket 74
 BuyAtStop 75
 CreatePane 65
 CreateSeries 54
 DrawCircle 70
 DrawCircle2 70
 DrawEllipse 70
 DrawLabel 70
 DrawLine 70
 DrawText 70
 GetSeriesValue 55
 HidePaneLines 65
 InstallBreakEvenStop 76
 InstallProfitTarget 76
 InstallReverseBreakEvenStop 76
 InstallStopLoss 76
 InstallTimeBasedExit 76
 InstallTrailingStop 76
 LastLongPositionActive 79
 LastPosition 79
 LastShortPositionActive 79
 PeakBar 70
 PlotSeries 67
 PlotSymbol 68
 PlotSyntheticSymbol 68
 PositionActive 79
 PositionCount 79
 PositionEntryBar 79
 PositionEntryPrice 79
 PositionLong 79
 SellAtClose 75
 SellAtLimit 75
 SellAtMarket 75
 SellAtStop 75
 SetAutoStopMode 76
 SetSeriesBarColor 67
 SetSeriesValue 55
 TroughBar 70
 while (looping statement) 33