# Improvian Language

User Guide

Version 3.0

Alyuda Research, Inc.

***Improvian Language User Guide***

# Contents

# Chapter 1

# Introduction

Welcome to the Tradecision's ***Improvian Guide***, a step-by-step guide that covers the basics of creating trading techniques with the help of the Improvian trading language.

## Notation Conventions

The following conventions are used in this Guide:

| Element | Meaning |
|---|---|
| *Italic* | Used for code characters |
| **Bold** | Indicates emphasis or a Tradecision's command or dialog box |
| ***Note:*** | Refers to significant information inside a given topic. |

## Is it Easy to Learn a Language?

Do you think that you have to spend years learning programming before you can write technical indicators or trading systems? You are wrong! You can start writing trading techniques as soon as you have learned several simple commands.

Of course, you will need some time before you can create a trading masterpiece, for example, a constantly profitable trading strategy. However, this does not mean that you will not be able to check your ideas, reduce risks or stabilize profits by way of creating a useful trading strategy, indicator or a function as early as now. As in any other endeavor, the first step is the most important. It is your luck that you have already made this first step by reading this comprehensive Guide which will help you master the Improvian trading language.

In this Guide you will find such methods for creating trading techniques that can be applied by any trader using the Tradecision application irrespective of his or her qualifications. You will see that acquiring these skills is rather easy. Furthermore, if you are patient enough and willing to spend some time on comprehending the main principles, the process of studying the techniques will become an easy and pleasant task.

By the time you are through studying this Guide, you will have the skills and knowledge to develop profitable trading techniques. Moreover, you will reach the crossing where you will be able to decide whether you want to be one of the gurus and continue honing your skills in creating profitable trading and analysis techniques.

## Who Should Read This Guide?

This Guide is for all those who want to start creating trading and analysis techniques in Tradecision using the Improvian trading language.

You do not need any preliminary training to use this Guide. All the main programming methods are explained on practical examples. All you need to understand the contents of the Guide is the basic computer skills, such as the use of the keyboard and mouse.

It is recognized that emotions cause traders to make poor trading decisions and thus lose part of their money or even the whole of their account. That is why many successful traders believe that using system-based trading or rules can be their competitive advantage.

Therefore, if you want to use the discipline approach in your trading, Improvian can help you implement your trading ideas.

## What is a Trading Technique?

Essentially, trading and analysis techniques created by traders using any trading software are relatively simple computer programs.

A computer program is a set of instructions in the form of words, codes, diagrams, symbols, and so on, expressed in a loadable form suitable for a computer and activated by it in order to achieve certain goal or results. Most programs consist of a loadable set of instructions that determine how the computer will react to user input when the program is running, i.e. when the instructions are 'loaded.'

## What is Improvian?

Improvian is a special programming language for traders. It gives you a variety of opportunities to implement your own trading ideas, for example, design a trading system or improve an existing study or indicator.

It is can be easy if you are ready to spend some time to explore the Improvian syntax. For example, a trading system is composed of one or more trading rules. In your strategy you just need to indicate what you want to be your buy, sell, sell short and other conditions.

Improvian can be accessed from the **Improvian Editor**, a special word processor that allows you to write your own ideas in the form of a strategy, indicator, study, and so on, using specific syntax.

Improvian is comprised of a combination of reserved words, operators, functions, parameters and punctuation.

Tradecision's Improvian Dictionary includes a collection of built-in functions covering many of the most popular trading techniques. These include all types of Moving Averages, Volatility, Momentum, chart patterns, to name but a few.

All language constructions are not case sensitive, i. e. "IBM" is the equivalent of "ibm".

## What Can You Create?

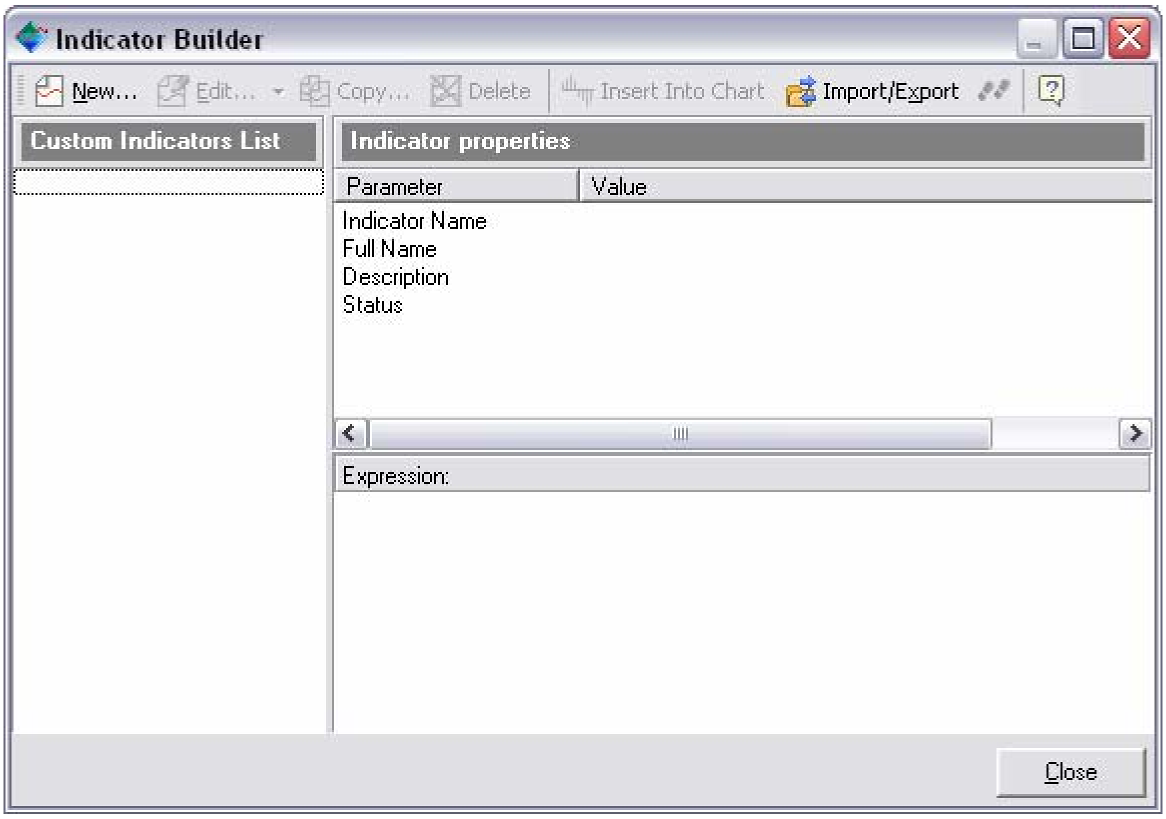
Using the Improvian language, you can create analysis techniques that can help you make a profitable trading decision. Inventing your own techniques isn't easy. First of all, you need an idea. Then you need to write that idea down. Improvian will allow you to write down any kind of idea you can only come up with, and Tradecision will inform you about how well your trading idea performs.

With Improvian you can create**:**

- Trading system rules (using the Strategy Builder capability);
- Inputs for Neural Models;
- Indicators (using the Indicator Builder capability);
- Custom Functions;
- Alerts (using the Alert Builder capability);
- Studies (using the Study Builder capability);
- Custom scans (using NeatScan market scanner).

## Technical Indicators

Indicators perform mathematical calculations for each bar. The results are presented in the form of a diagram which enables visual comparison with price data or other indicators based on instructions that were created using the Improvian language. Indicators are built in **Indicator Builder,** a special tool for indicator building and management.

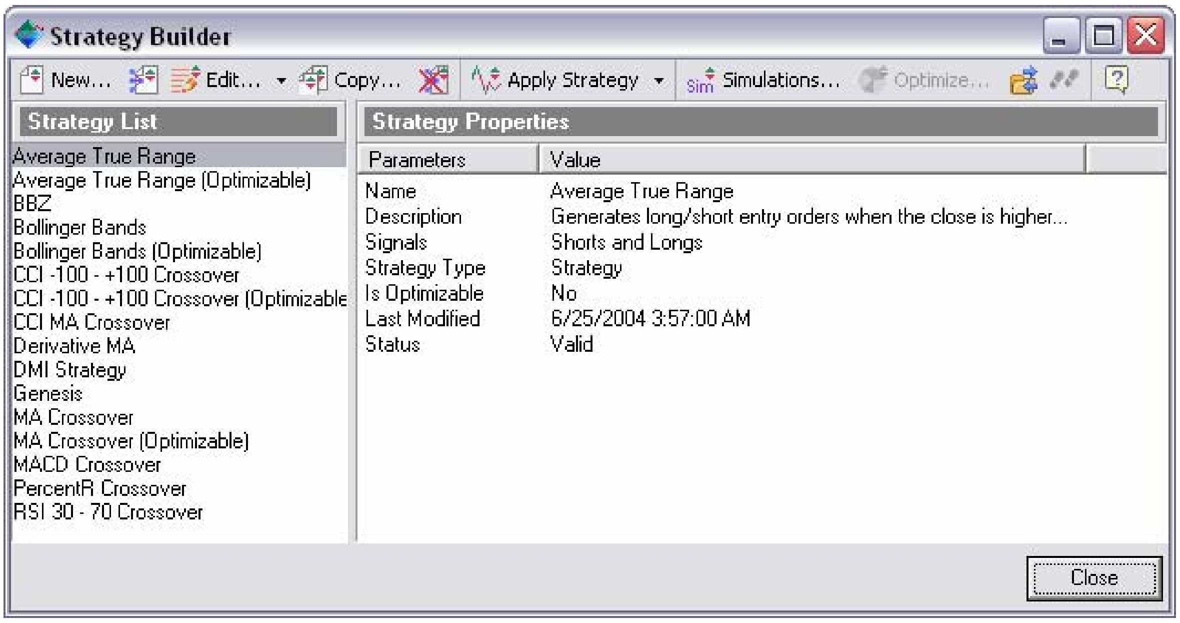


| Parameter | Value |
|---|---|
| Indicator Name | |
| Full Name | |
| Description | |
| Status | |

Expression:

## Trading Systems

Trading systems contain conditions (as well as instructions required to calculate the conditions), with the help of which the commands of input and output from a position can be performed. A trading system can be back-tested to identify its profitability, risk level and efficiency. **Strategy Builder** is a separate tool used to build and operate trading systems. **Simulation Manager** is used to analyze strategies.



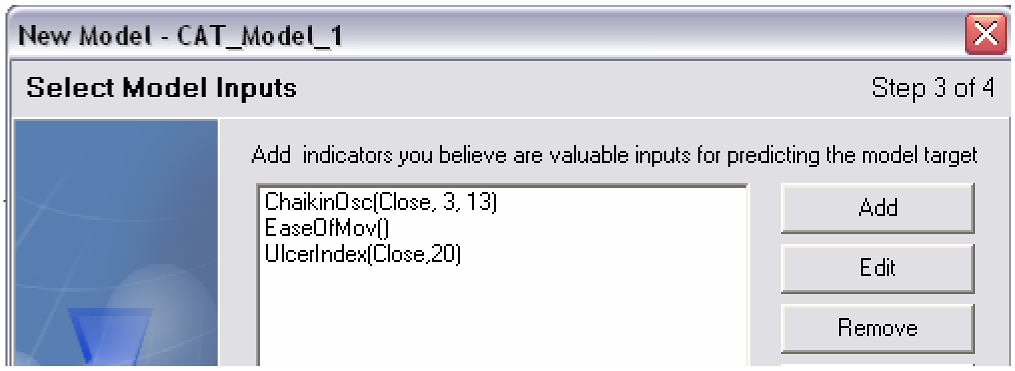
Average True Range
Average True Range (Optimizable)
BBZ
Bollinger Bands
Bollinger Bands (Optimizable)
CCI -100 - +100 Crossover
CCI -100 - +100 Crossover (Optimizable
CCI MA Crossover
Derivative MA
DMI Strategy
Genesis
MA Crossover
MA Crossover (Optimizable)
MACD Crossover
PercentR Crossover
RSI 30 - 70 Crossover

| Parameters | Value |
|---|---|
| Name | Average True Range |
| Description | Generates long/short entry orders when the close is higher... |
| Signals | Shorts and Longs |
| Strategy Type | Strategy |
| Is Optimizable | No |
| Last Modified | 6/25/2004 3:57:00 AM |
| Status | Valid |

## Inputs and Targets for Neural Models

Inputs and targets for neural models are expressions of any degree of difficulty, forming input and target parameters for neural networks.

The possibilities of Improvian enhance neural networks’ potential and enable traders to gain an advantage over those market-players who use other trading neural network-based software. **Model Builder** is a special tool designed to manage neural network models.


New Model - CAT_Model_1

Select Model Inputs

Step 3 of 4

Add indicators you believe are valuable inputs for predicting the model target

ChaikinOsc(Close, 3, 13)
EaseOfMov()
UlcerIndex(Close,20)

## Alerts

Alerts are a set of conditions which, when met, cause Tradecision to send a visual and/or audio signal. The Improvian-defined conditions for each alert are automatically checked at each bar for all the defined securities, helping the trader avoid missing an important trading opportunity. **Alert Builder** is a specific tool designed for building and managing alerts.

## Studies

Studies are similar to indicators in terms of computational methods. However, unlike indicators, they find bars that meet certain requirements and mark them on the chart. **Study Builder** is a specific tool designed to build and manage studies.

## Scans

Using market scans you can find stocks that meet some requirements defined with the help of Improvian expressions (or predefined conditions). By using Improvian you can examine hundreds of symbols in real time against your own custom-defined criteria. **NeatScan** is a separate application, designed for building and managing scans.

## Position Size Calculations

Position size calculation is a mathematical expression, defining the size of the next position. One of the most efficient strategies to increase the trading efficiency is the calculation of a position size based on the indicators of the trading system or the current risk/reward ratio. The Improvian language offers unlimited opportunities for performing this kind of calculations. Position size calculations can be edited in **Money Management Editor**.

## Function

Function is a set of frequently used or logically separate instructions. Building functions allows saving time and makes trading techniques more clear and compact. **Function Builder** is a specific tool designed for building and managing functions.

In addition, you can modify any of the built-in trading strategies, studies, indicators, and functions that are included in Tradecision.

Moreover, you can test your own money management rules defining them with Improvian. As traders know, money management is the most important part of successful trading. And Tradecision provides you with a powerful Money Management Editor. For your convenience the most popular money management rules are predefined and easily accessible.

Tradecision can store an unlimited number of technical analysis techniques.

For specific tutorials on each of the formula-based tools (Indicator Builder, Strategy Builder, and so on), refer to their specific chapters in the **Tradecision User Manual**.

# CHAPTER 2

# The Basics of Improvian

## Processing Principles

One of the key purposes of Improvian is to compare the price data from one bar and with that from other bars. In this connection, you need to understand how your Improvian techniques, such as technical indicator, trading strategy, and so on, examine the price data from a chart.

Typically, you receive data from a data provider in the **Data Manager**. Then you open a chart in Tradecision. A specified symbol's chart consists of multiple bars made up from price data. It can be minute, daily, weekly and other bars. Each bar simply sums up the prices for a trading interval and can contain such price values as open, high, low, and close prices for the period. Consequently, a chart is a visual representation of a period of trading history for a symbol, whereby individual bars represent trading periods.

The following is what happens when this simple trading strategy rule for Entry Long is written down using Improvian.

*Return Close > High\1\;*

Incidentally, you are instructing Improvian to compare the closing price of the current bar with the high price of the previous one, and generate an Entry Long order when the close exceeds the previous high.

This comparison is made for the closing price of every bar in the chart, the high price from the previous bar being compared every time. Although your trading technique is applied to a price chart filled with different bars, the procedure that is used to evaluate the data in the chart is always the same.

Each price bar includes price, volume, OI, date, and other information, received from a data provider. To evaluate a chart, Improvian starts reading the price data from the first bar in the chart as it came from the data provider. According to your Improvian procedure, it is now the current bar. The Improvian statements in your procedure are always evaluated relative to the current bar. With the first bar, there are no previous bars and, thus, the above comparison does not hold true.

When the process is completed as far as the bar evaluation is concerned, Improvian goes ahead in time to the next bar, making it the current bar for which the statements in your procedure are estimated.

Generally, an Improvian procedure consists of a number of statements which can result in a trading activity (for example, generating a buy or sell signal). Once all of the statements in the Improvian procedure for the current bar are completed, the price data from the next bar is read and the procedure is run another time using the new price data. The procedure is repeated, across the chart, from left to right, until all of the prices from all the bars on the chart have been examined. As a result, for a 500 bar on a chart, the Improvian procedure runs 500 times, once on every bar.

## Price Data Identifiers

Improvian gives you the very useful ability to evaluate price data. Moreover, Improvian provides a number of reserved words for accessing price data from each bar.

The following table represents the most frequently used price data values.

| Price Data | Abbreviation | Description |
|---|---|---|
| Open | O | First available price for the bar |
| High | H | Highest price within the bar |
| Low | L | Lowest price within the bar |
| Close | C | Last available price for the bar |
| Volume | V | Total volume of trades within the bar |
| Open Interest | OI | Total number of open contracts |

## Referencing Price Data

You can set the number of bars trading techniques can reference from the **Preferences** dialog box. For this:

1. From the **Tools** menu, click **Preferences**.
2. In the **Preferences** dialog box, select the *Improvian* tab.
3. In the **Calculations** area, select how many bars ago the trading techniques can reference by entering a value in the **Techniques can reference … bars back** box. The default value is 100 bars.

## Reserved Words

As any other language, Improvian has a set of main components that make up its core. These are words, symbols or methods used to build expressions. These components should be organized and ordered in a certain manner to build correct sentences and expressions. The basic Improvian dictionary contains a set of reserved words. Each of the reserved words has its meaning or purpose, such as a definition of program sections, branching identification, comparison or reference price values.

For example, in the following expression:

*if a>2 then a:=2;*

**if, then** are reserved words.

The following keywords are reserved to identify price fields:
· Open
· High
· Low
· Close
· Volume
· OI
· O
· C
· H
· L
· V

The following keywords are reserved to be used in Boolean expressions:
· True
· False
· AND
· OR
· NOT

The following is a list of the most frequently used Improvian Reserved Words.

| | | |
|---|---|---|
| var | if | byref |
| vars | then | byval |
| variables | else | true |
| variable | while | false |
| end_var | do | println |
| end_vars | for | print |
| end_variables | to | |
| end_variable | downto | |
| end_input | begin | |
| end_in | end | |
| end_ins | array | |
| end_inputs | function | |
| input | return | |
| in | boolean | |
| ins | numeric | |
| inputs | string | |

You cannot use a reserved word as a name for your indicator or any other trading technique.

From this point of view, some built-in functions can be considered as reserved words too. You can look it up in the **Improvian Function Reference**, whether a function can be used as a reserved word. For example, the following description for the Low function is provided:

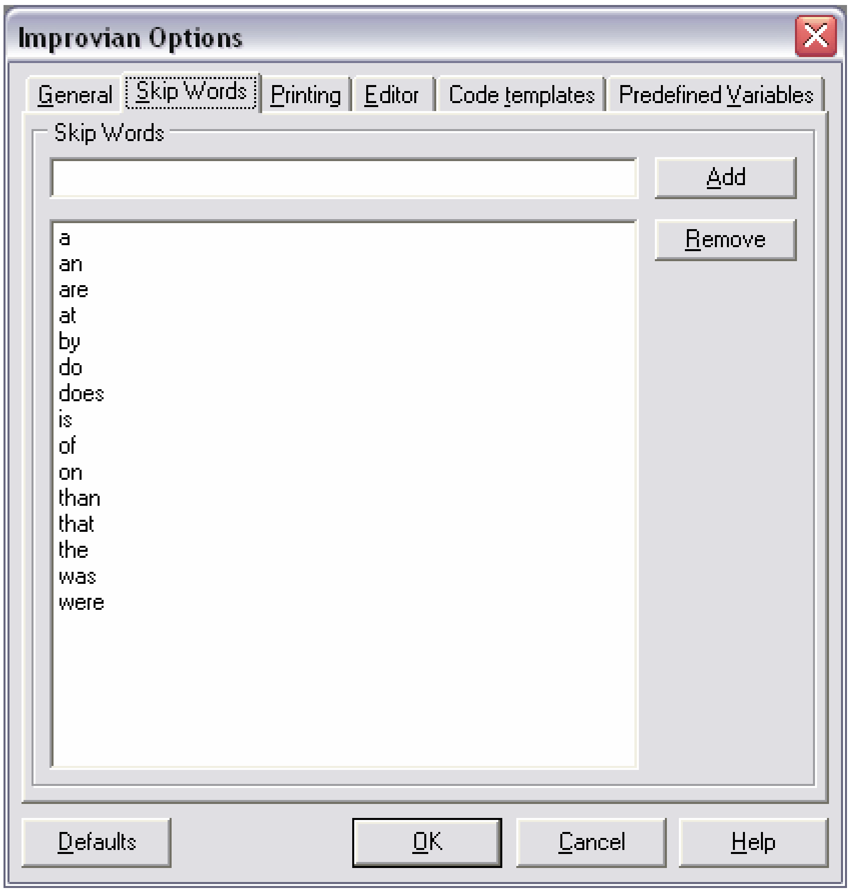
Name: Low
Full name: Low
Syntax: Low
Description: Returns Low value specified number of bars ago.
Returns: A numeric value
Example: Low
Reserved Word: **Yes**

## Skip Words

Skip words are another group of reserved words, used to make your techniques read more like English (or any other language). The Improvian Editor ignores these words when verifying your trading techniques.

Unlike in any other trading language that contains skip words, you can create your own dictionary of skip words to be used for creating trading techniques. For example, you can add words such as "shares", "later", "afterwards", "again", "British pounds", and so on, or replace the default skip words with their corresponding translation into your preferred language to make your rules more readable to you or your colleagues.

Skip words are managed from the *Improvian* tab. (Select **Tools** > **Improvian Options**> **Skip Words** tab).



a
an
are
at
by
do
does
is
of
on
than
that
the
was
were

## Expressions

An **expression** is any combination of reserved words, values, price data and operators that calculates some value. For example,

*(High-Low)/(High-Close) or ((Close > 50) AND (Close>Open)).*

Expression is part of a statement.

An expression can be:

- Numeric;
- Boolean (also called logical or true/false);
- String (text).

**Numeric expression** is a number or a reserved word that represents a numeric value or a combination of numbers, operands and reserved words that evaluates to number; for example:

*Open-10*

**Boolean or True/false expressions** can be either a *True or False* value*,* or an expression that evaluates to True or False. True/false expressions invariably involve a comparison, for example:

*Open > Close*

**A string (text)** is a set of symbols (no more than 1000) in parenthesis, for example:

*"This is a string"*

**String expression** is a string or a reserved word that represents a text value or a combination of strings, operands and reserved words that evaluates to string, for example:

*"This is a string " + "expression"*

## Statements and Punctuation

A **statement** is a combination of expressions and reserved words that represents a complete instruction and always ends into a semicolon, such as, for instance, the If-Then structure or variable declaration statements.

For example, **"return" statement** is an instruction of return. It stops a program's execution and returns a value.

Statements can be very simple, such as:

*Return Close>Open;*

or more complex multi-line expressions, such as:

```
if (Length >0) and (MyIndicator(Length)<14) then
  Average:=Sum / Length
else
  Average:=0;
```

Even though the second example includes several calculations and conditional expressions, both examples are valid statements that start with a statement reserved word (return and if correspondingly) and end with a semicolon (;).

In addition to the end-of-statement punctuation mark, there are several other punctuation symbols. You will use these often while writing in Improvian. The table below summarizes the usage of the punctuation symbols.

| Symbol | Name | Description |
|---|---|---|
| ; | Semicolon | Used to indicate the end of a statement |
| ( ) | Parentheses | Used to indicate operator precedence or group calculations |
| , | Comma | Separates items, such as parameters used with functions, on a list, |
| \ \ | Backslashes | References price data from the previous bars |
| [ ] | Square brackets | References array elements |
| { } | Curly brackets | Used to indicate comments |
| " " | Quotation marks | Used to indicate string (text) items |

## Operators

Operators enable you to manage reserved words and values to create complex expressions. Different operators are available for the different types of operands. The tables below summarize the operator usage and result type.

| **Operator** | **Operand Type** | **Semantics** |
|---|---|---|
| (unary) - | Numeric | Changes an operand value to the opposite. |
| + | Numeric | Sums up the values of two numbers. |
| + | String | Merges two rows. |
| - | Numeric | Subtracts the value of the second operand from the value of the first operand. |
| * | Numeric | Multiplies the value of the first operand by the value of the second operand. |
| / | Numeric | Divides the value of the first operand into the value of the second operand. |
| < | Numeric | Returns *True*, if the first operand is lesser than the second operand. |
| > | Numeric | Returns *True*, if the first operand is greater than the second operand. |
| <= | Numeric | Returns *True*, if the first operand is lesser or equal to the second operand. |
| >= | Numeric | Returns *True*, if the first operand is greater than or equal to the second operand. |
| = | Numeric | Returns *True*, if the first operand is equal to the second operand. |
| <> | Numeric | Returns *True*, if the first operand is not equal to the second operand. |
| = | String | Returns *True*, if the symbol string of the first operand is identical to the symbol string of the second operand. |
| <> | String | Returns *True*, if the symbol string of the first operand is not identical to the symbol string of the second operand. |
| (unary) not | Boolean | Returns *True*, if the operand's value is *False*, and returns *False* if the operand’s value is *True*. |
| and | Boolean | Returns *True*, if the value of the first and second operands are *True*, and returns *False* if otherwise. |
| or | Boolean | Returns True, if the value of at least one of the operands is True, otherwise False. |

| Operators | Operand Type | Result Value |
|---|---|---|
| +,-,*,/ | Numeric | Numeric |
| + | String | String |
| <, >, =,>=,<=,<> | Numeric | Boolean |
| =,<> | String | Boolean |
| not, and, or | Boolean | Boolean |

## Operator Precedence

When writing a program string that contains numerous arithmetic operations, you might want to more clearly indicate the operator precedence. For this purpose, round brackets should be used.

The following examples illustrate the operator precedence. This precedence can be changed with the help of brackets. The operations in brackets are the first to be executed.

For instance,

*Close – (High – Low) is understood as: High – Low = X, then Close – X = Y*

When there are numerous brackets, the calculations start with the innermost brackets.
Expression 4 * (2 * (3+4))...is calculated as 3 + 4 = 7, then 2 * 7 = 14, then 4 * 14 = 56.

If an expression does not contain any brackets, all the operations should be executed according to the rules of operator precedence. Equal precedence operations have to be made from left to right. For example:

*2+3+4 = (2+3) + 4.*

The table below shows the precedence order assigned to operators in the Improvian. The operators in this table are listed in the precedence order: the higher the operator is in the table, the higher is its assigned precedence.

Here is the **Operator Precedence** table:

| |
|---|
| Not, (unary) - |
| /, * |
| +, - |
| <=, >=, >, <, =, <> |
| And |
| Or |

# CHAPTER 3

# Creating the First Custom Indicator

## Using Indicator Builder

Indicator Builder is a special tool used to create and manage custom indicators. To launch Indicator Builder, click **Tools** > **Indicator Builder** menu item or press F8. Indicator Builder provides easy access to the tasks you may need to complete while managing or using your custom indicators.

You can quickly select an indicator from the Custom Indicators List. The **Indicator Properties** pane displays the name, description and status of the selected indicator along with the Improvian code that instructs how to calculate the indicator. You can use the Indicator Builder toolbar to create new indicators, edit a selected indicator, create a copy of an indicator, delete an indicator or insert it into the current chart.

Tradecision contains several pre-defined custom indicators you can use at the beginning. Below you can see how to create new custom indicators using the formula for the well-known Stochastic Fast %K indicator or rather part of a well-known Stochastic indicator.

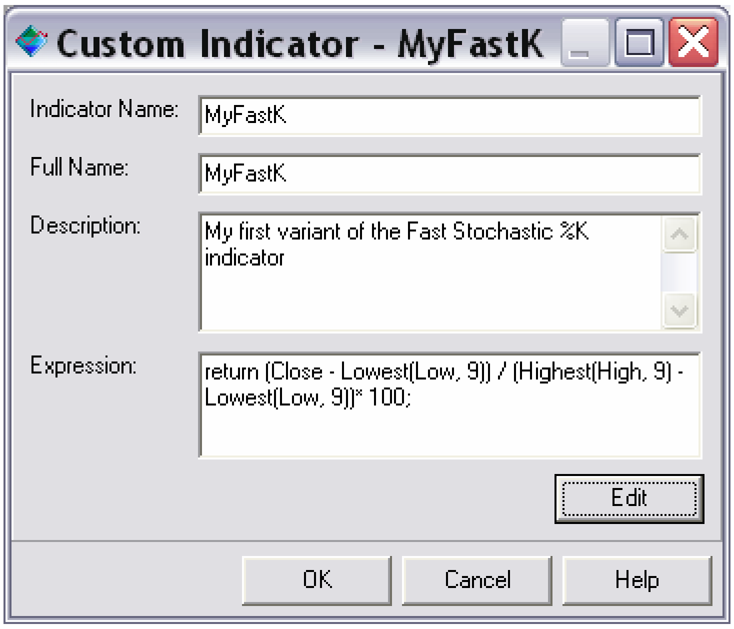
To create the Stochastic Fast %K indicator:

1. On the Indicator Builder toolbar, click **New**.
2. In the *Custom indicator* dialog box, in the **Indicator Name** and **Full Name** boxes, type "*MyFastK*".
3. In the **Description** box, type "*My first variant of the Fast Stochastic %K indicator*".
4. Click the **Edit** button.

The Improvian Editor will be displayed.

5. In the *Improvian Editor* dialog box, in the Expression box enter

*return (Close - Lowest(Low, 9)) / (Highest(High, 9) – Lowest(Low, 9))* 100;*



Indicator Name: MyFastK

Full Name: MyFastK

Description: My first variant of the Fast Stochastic %K indicator

Expression:

```
return (Close - Lowest(Low, 9)) / (Highest(High, 9) -
Lowest(Low, 9))* 100;
```

6. Click **OK**.

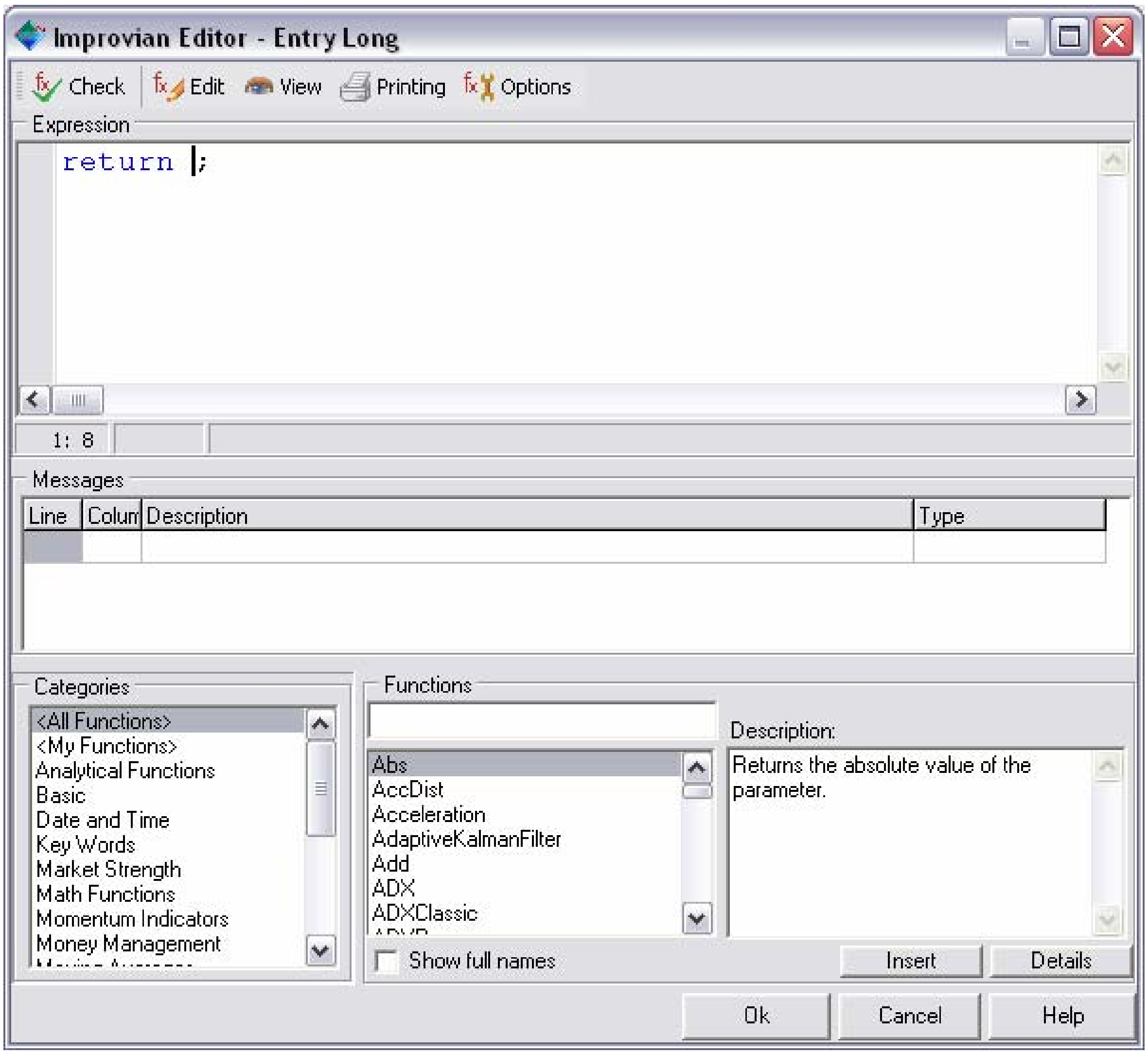
7. In the *Custom indicator* dialog box, click **OK** to save the new indicator and add it to the list.

   ***Note:*** *This is not the final version of the indicator, since division into zero is possible here. The correct indicator code is provided in one of the following chapters.*

In the next chapter, you will learn how to use the Improvian Editor.

## Using the Improvian Editor

The Improvian Editor is a special word processor that allows you to write your own formulas in the Improvian language. The Improvian Editor is automatically displayed in a separate dialog box when you create or edit an indicator, strategy, study, alert, scan or model input.



```
return ;
```

1: 8

<All Functions>
<My Functions>
Analytical Functions
Basic
Date and Time
Key Words
Market Strength
Math Functions
Momentum Indicators
Money Management

Functions: Abs, AccDist, Acceleration, AdaptiveKalmanFilter, Add, ADX, ADXClassic

Description: Returns the absolute value of the parameter.

The **Expression** pane is used to write an indicator or another technique code. The case used when entering techniques does not matter. For example, “RSI" and "rsi" are treated the same way. You can use the standard clipboard commands while editing expressions. To copy the highlighted text, press CTRL+C; to cut, press CTRL+X; to paste, press CTRL+V.

The **Messages** pane contains error messages if your code is invalid.

The lower pane contains the Improvian **Dictionary**, where you can select built-in functions to view their details or insert the functions into the Expression box under the cursor with a double click.

To use a built-in function from the Dictionary:

1. In the **Categories** group box, select the appropriate category;

2. In the **Functions** area, select the function that you need or simply type the function in the box under **Functions**. You can read the description of the selected function in the **Description** box.

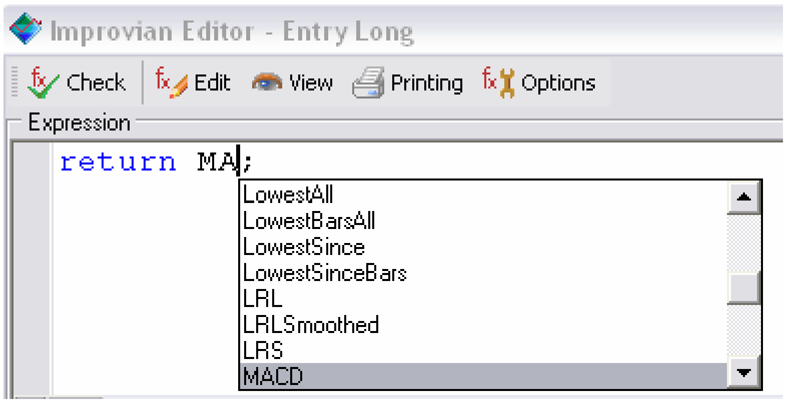
3. Select the highlighted function, and double-click it, or click **Insert.**

The selected function will be displayed in the **Expression** box.

## Using AutoComplete

There is another way of using the Improvian Dictionary, which is **AutoComplete** feature. In the **Expression** box, enter the first letter of the built-in function and press Ctrl-Space or right-click to select **AutoComplete** from the list.

The capability allows automatic expansion of a word from the Dictionary when only several of the letters are entered.



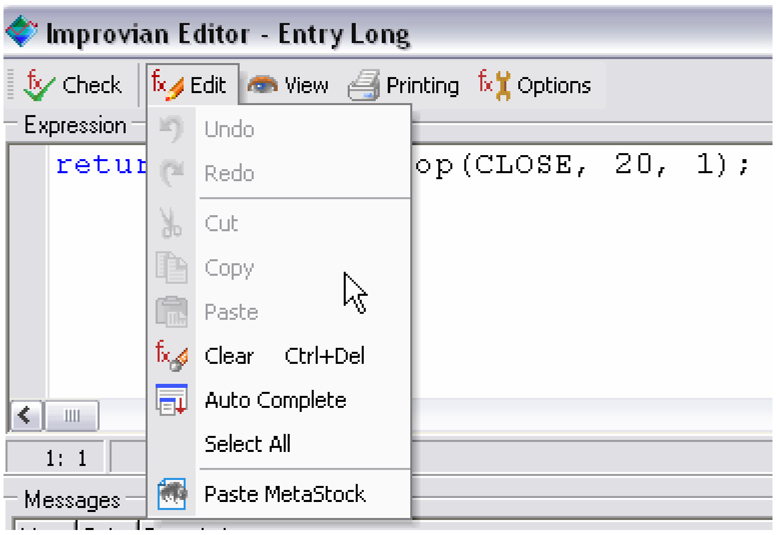
```
return MA;
```

## Improvian Editor Toolbar

You can use the Improvian Editor Toolbar for the following tasks:

**Check** - checking indicator (or another technique) validity. If an error is found, a message explaining the error will appear in the **Messages** box.

**Edit** - accessing Undo & Redo features, work with the Clipboard, clear the **Expression** box, use Auto Complete function, or paste techniques from MetaStock©.



```
retur        op(CLOSE, 20, 1);
```

**View** – selecting whether to display Messages pane and Dictionary pane or not.

**Printing** – printing your technique.

**Options** – changing editor preferences, modifying fonts and colors for Improvian lexical and syntax constructions and creating/editing code templates.

## Using Variables

Variables are simply named blocks of memory or placeholders. As long as a trader (i.e. you) provides names for them, he can assign any names to the variables. To make the text of your trading technique clear and easy to read, try to make these names meaningful.

In your technique you will be able to save the values in memory blocks. After that you will be able to refer to these values using their names, or recalculate them. For instance, to save an entry containing the total volume of trades for the previous bars, you can create a variable with the name "TotalVolume". With every new bar, you will be able to place a new value of the total volume of trades into the variable called "TotalVolume." By creating variables you reserve some part of the memory for storing data that your technique can later use.

When inventing names for your variables, abide by the following rules:

Firstly, the name of the variable cannot exceed 255 symbols. When selecting the length and name of the variable, try to find the right balance between descriptiveness and conciseness. Very short names, such as i, k, st, do not provide any information about the value of the variable. Very long names take too long to write and read.

Secondly, the name of the variable should start with a letter. The following symbols can be letters, numbers and underline symbols. *You cannot use spaces and most of the punctuation marks in the name of a variable*. Since Improvian is not case-sensitive, you can use caps and lower case characters in the names of the variables in any combination. For instance, ParabolicSAR, parabolicsar and ParabolicSar in Improvian will be considered as one and the same name.

Finally, you cannot use any of Improvian's reserved words as variable names.

The following are examples of valid variable names:

*BuyLevel*
*My_SMA*
*PriceLevel*
*Value23*
*AMOUNT@ParabolicSAR*

The following are examples of invalid variable names:

*79ParabolicSAR*
*Price Level*
*Cur.volume*
*Close*
*14*

## Declaring Variables

In Improvian, the variables can be of three types: Numeric, Boolean and String. Strictly speaking, only a string variable can contain text and only a numeric variable can contain a numeric value. To avoid pitfalls and inaccuracies, we suggest that you declare the variables before you start using them.

When you declare a variable, you not only impart the name of the variable to Improvian, but you also indicate how this variable should be used. For instance, to improve your indicator MyFastK, you can add the following strings at the beginning of the code:

```
Var
        LowestLow := 0;
        Difference := 0;
End_var
```

The first string («*Var*») informs the Improvian that the following is the variable declaration block.

The second string («*LowestLow := 0;*») is the declaration of the variable. It informs the Improvian that you are going to use the variable with the name LowestLow in the program and that this variable will be used as a number. Similarly, the third string informs that you are going to use the numeric variable Difference.

And, finally, the last string says that this block of the declaration of variables is complete and provides instructions on how to calculate the indicator.

The first variable is used to avoid double-calling the LowestLow function in the indicator body, thus increasing the speed of the calculation.

The second variable is used to improve the readability of the code and shorten the long formula. As a result, the source code of your indicator MyFastK can be written as follows:

```
Var
        LowestLow := 0;
        Difference := 0;
End_var

LowestLow := Lowest(Low, 9);
Difference := Highest(High, 9) – LowestLow;

return (Close - LowestLow) / Difference * 100;
```

As you can see, both variables store the calculation results in order to refer to the results of this operation later without having to repeat the formula or expression.

This is not the final version of the indicator, since division into zero is possible here. The correct indicator code is provided in one of the following chapters.

> ***Note:*** *When creating your trading techniques in Improvian, be careful not to make any spelling mistakes in the variable names. Mistakes in variable names cause errors in programs. Such errors are difficult to*

*find. For example, let us assume that there is a variable called Difference in your technique and you have misspelled this variable as Diference in the technique. For Improvian Difference and Diference are absolutely different variables and each of them can have its own value.*

*Every symbol of the program is important. Be careful not to miss a single symbol and not to make a single mistake: this will result in Improvian displaying an error message or in an unpredictable erroneous behavior of the trading technique.*

# Understanding Technique Branching

## Program Flow and Branching

Technique Branching is the order according to which a technique executes its code. Up until now your indicator had a successive program stream and the actions have been performed starting from the first string, and then, consecutively, string after string, till the very end of the program.  However, most of the indicators will sooner or later reach a point where a decision has to be made regarding some specific part of the data. Here the technique should analyze the data, decide what to do with it and move on to the corresponding part of the code.

The situation when the consecutive flow of your technique is interrupted and the technique jumps to a new section of the program code as result of a comparison or decision made based on certain parameters is called branching or Technique Branching.

You need to understand how you can manage your technique sequence, i.e. the sequence in which the operators are executed, in order to be able to perform various operations depending on data that your technique obtains.

## Understanding Operator If…Then

The operator If…Then is most often used to organize the branching of a program. This operator compares the data and makes a decision on what needs to be done based on the results of the comparison.

If this condition is met, the technique is executed by the operator following the keyword "Then." Otherwise, the program performs no action and simply proceeds to another string. Thus, each comparison has two possible outcomes.

> ***Note:*** *Relational operators, such as the sign of equality, enable you to compare two portions of data. For instance, comparing literal variables you can check the variables for certain values. The most frequent is the relational operator represented by the sign of equality (=) which checks whether the two expressions are equal. However, there are also relational operators for such relations as less (<), more (>), and not equal (< >). When you use relational operators to compare two values, you write a Boolean (or logical) expression otherwise called as a true or false condition.*

## Simple operator If…Then

The simple operator "If…Then" consists of the key word "If" followed by a Boolean expression (condition). You should end the Boolean expression into the key word "Then" and an operator or a group of operators that you want to accomplish if the Boolean expression is true.

A Boolean expression is the expression which takes one of the two meanings: true or false. For instance, expression 3 + 4 = 7 is true, while expression 6 + 1 = 9 is false. Usually, a Boolean expression compares a variable with a constant or with another variable, such as MovingAverage + 0.1 > LatestClose OR MovingAverage - 10 = CurrentLow .

To avoid the division by zero mistake in the indicator MyFastK, we can change the expression as follows:

```
Inputs
        Length:"Length",9,0,1000,1;
End_inputs

Var
        LowestLow := 0;
        Difference := 0;
End_var

LowestLow := Lowest(Low, Length);
Difference := Highest(High, Length) - LowestLow;

if Difference = 0 then return 0;

return (Close - LowestLow) / Difference * 100;
```

***Note:** Some indicator strings are printed with an indent. By combining the indented indicator strings that are part of the "If" block you can better see the structure of your source code. Empty strings are used in the body of the indicator in order to separate the code blocks from one another. Tradecision overlooks all these empty strings and indents. However, this makes your indicators easier to read and understand for you and any other programmer.*

*What happens inside this version of MyFastK indicator? Let us assume that for some of the bars the value HighestHigh(High, Length) will be equal to the value LowestLow. When Tradecision reaches the operator If…Then, it compares the value of the variable Difference with the number "0." As most of the bars these values do not correspond to each other, or, in other words, the statement is false, Tradecision skips the code string following the keyword "Then" and proceeds by executing the keyword "else."  In those cases when Difference is equal to zero, Tradecision will avoid making a mistake of dividing into zero by assigning the zero value to the indicator.*

## Assigning Values to Variables

While reviewing code of a new version of the indicator, you should remember that the special operator (:=) is used to assign values to variables. The variable is located to the left of the operator. It is followed by the operator ":=", and then which, in turn, is followed by a description of the action performed by the value.

The expression to the right can be a number, a mathematical expression, a built-in Improvian function, generating the value, or any other variable. In the latter case, the variable value to the left of the operator «:=» will receive the same value as the variable to the right. In this case, any kind of character will be acceptable to the right of the sign of equality, which is produced by the value corresponding to the type of the variable. The variable type (numeric, Boolean, string) is defined by the first assignment in the block of the declaration of variables Var.

> ***Note****: In all the above cases, the mathematical operation is located to the right of the ":=" sign. The following entry is unacceptable: Variable1 + Variable2 := Variable3.*

## Using Inputs

The current version of the calculation Highest and LowestLow functions maintained for the last 9 bars.

Obviously, there is a need to increase or reduce this value in accordance with each specific security.

In order to eliminate the need to change the source code every time or build several very similar indicators that vary by one number in the calculation, such input values are used that can be set each time the indicator, or any other technique, is called. In other words, the inputs are indicator parameters that can be replaced and are used to calculate the indicator value.

To set inputs, a special block is used, which begins with the keyword Inputs (or Ins) and ends into the keyword End_inputs (or End_ins). Every input should have a name and initial value. To ensure automatic control of the parameter entry accuracy, you can set the minimum and maximum possible parameter values and incremental step.

Within the indicator you can use input values in the same manner as you use numbers. However, you cannot assign any new values to an input inside a technique, as to do so you will need a special variable. In other words, the input is used as a constant specified before using the technique.

> ***Note:*** *If you need to use some constant value within your technique, you can add a separate parameter in the Inputs block with the constant value. Tradecision will ensures that the parameter will not be changed and will be used as a constant.*

To make the MyFastK indicator universal you will need to add at least one input: the number of bars for the calculation of HighestHigh and LowestLow (9 by default).

To add an input, open MyFastK in Improvian Editor and enter the following strings at the very beginning of the indicator body:

```
Inputs
	Length: "Length", 9;
End_inputs
```

Now replace 9 with the Length input in the indicator code.
You will also be able to set the minimum (0) and maximum (500) input values and increment step (1). As a result, the complete indicator body will look as follows:

```
Inputs
	Length: ”Length”, 9, 0, 500, 1;
End_inputs

Var
	LowestLow := 0;
	Difference := 0;
End_var

LowestLow := Lowest(Low, Length);
```

```
Difference := Highest(High, Length) – LowestLow;

return (Close - LowestLow) / Difference * 100;
```

# CHAPTER 4

# Creating the First Custom Study

## Understanding Nested If...Then

The simple construction "if…then…else" helps make a decision depending on whether a certain condition is true or false, i.e. you have to make a decision on whether "to execute the first or the second set of instructions depending on the condition." If you need to make a decision on whether "to execute the first, second or third set of instructions" (or more complex) nested operators, "if…then…else" are used.

By using the nested cascade operators "if…then" you can reduce the time spent on calculations. You can select to use these operators only when required, i.e. when a certain condition is met.

When Tradecision attends to the first operator "If...Then," the application checks the value of the variable "choice" as a result of each bar's analysis using the abovementioned study. If both conditions are true (which is the case with hammer bars. See page 50), Tradecision proceeds to the next operator "If...Then." If one of the conditions is false (for more information on operators and logical operators refer to the following chapters), Tradecision will ignore the nested operator "if…then" and proceed to the operator "else", assigning the value "false" to the study, which will mean that this bar cannot be a hammer bar.

Using the combination of the keywords "if…then…else" we can write a program that can make alternative decisions. In itself, the "if…then" combination can help make a decision and execute special commands when the condition is met. However, if the conditional operator contains the "else" phrase, the program will select one of the two sets of operators depending on the result of the comparison operation written in the inequality.

If the inequality is true, those conditional operator instructions will be executed that belong to the "if…then" construction. If the inequality is false, those instructions that belong to the "else" construction are executed. The execution of the program follows one of the branches. After the construction "if…then…else" is executed, the instruction of the "else" construction following the last semicolon is executed.

## Using Comparison Operators

In your first indicator MyFastK only an equality operator was used for comparison. Often, you will need to compare values using other methods. Sometimes, as in the hammer example on page 50, one needs to find out whether a value is smaller or larger than another value. Improvian demonstrates the full range of relational operators that you can use in "If…Then" operators and other operators that use comparisons. The following table lists relational operators and simple examples of their use.

## Relational Operators

| Operator | Value | Examples |
|---|---|---|
| = | Equal | 5 = (6 - 1), Open = Close or «IBM» = «IBM» |
| <> | Not equal | 5 **<>** (4 + 4), Open <> Close or «IBM» <> «MSFT» |
| < | Less than | 5 < 44, Open < Close or «AAA» < «BBB» |
| > | More than | 51 > 29, Open > Close or «C» > «F» |
| <= | Less or equal | 6 < = 7, Open <=Close or «C» < = «C» |
| >= | More or equal | 14 >= 14, Open >=Close or «O» > = «C» |

## Using Logical Operators

The term "Logical Operators" is used to mean Boolean operators, i.e. operators dealing with True/False values. Those are operators such as & (AND), |NOT, and || (OR), enabling you to create logical expressions that yield true or false results.

As it is clear from the hammer study example on page 50, a simple comparison using the "If…Then" operator is often not enough to identify whether a bar complies with your criterion. How can you be sure that, for instance, a price falls within a certain range? The best way to guarantee that data is within a proper range is to use logical operators in the "If…Then" expression.

Let us assume that you need to make sure that "Close" is within the range between 10 and 50 inclusive. To do so, you need to make sure that "Close" is more or equal to 10. However, you will also need to make sure that "Close" is less or equal to 50. To help you handle this situation, Improvian offers you three logical operators: **And, Or,** and **Not**, which can be used in any kind of combination in the "If" conditions.

The **And** operator requires that all expressions be true for the complete expression to be true. For instance, in order to be sure that "Close" is within the range from 10 to 50 inclusive, you need to set the following condition:

*Close >= 10 And Close <= 50*

If the value of "Close" for any of the bars is equal to 51, the general condition will be false because the left part of the expression is false. This should be kept in mind when combining expressions using And: if any of the expressions is false, the overall expression is false.

In case of the hammer study, it would be expedient to carry out further checks only if for the current bar the hammerhead is located above the middle of the bar (Min > MedianPrice) and if this hammerhead exists at all (Open <> Close).

*if Min > MedianPrice and Open <> Close*

The **Or** operator requires at least one expression to be true for the overall expression to be true. For instance, both the following expressions are true as at least one of the compared expressions is true;

(2 + 3 = 1) Or (2 + 2 = 4)

Close >= Open Or Open > Close

The **Not** operator inverts (or negates) the value of the logical expression. For instance, the following expression is not true:

(High < Low)

However, if you add the Not operator before this expression, it will turn the false expression (High < Low) into a true expression: Not(High<Low)

And, Or, Not are called logical operators because they use computer logic to combine two or more Boolean expressions into one big Boolean expression {keep it in mind that a Boolean expression takes one of the two expressions: true or false).

Look at this expression:

*(3 + 4 = 7) And Not (4 + 1 = 4)*

Is this expression true or false? If you conclude that it is true it means that you have understood how the logical operators function. The expressions on both sides of "And" are true. Therefore, the entire expression is true. The table below summarizes your knowledge of logical operators.

## Logical Operators

| Operator | Value |
|---|---|
| And | True, if both sides of the expression are true. |
| Or | True, if one or two sides of the expression are true. |
| Not | Turns true to false and vise versa. |

Of course, you do not need to write such expressions in your program:
(3 + 4 = 7) And Not ( 4 + 1 = 4)

They do not serve any other purpose than to educate. However, when you use variables, you can never know in advance whether your expression will be true or false. For instance, is the expression below true or false?

*(Profit >= MinProfit) And (Risk <= MaxRisk)*

Until you know the true value of the variables Profit, MinProfit, Risk and MaxRisk you will not be able to say anything about these expressions. Nevertheless, using logical operators in "If…Then," assertions your program will be able to receive data and take the appropriate action based on whether the expression turns out to be true or false.

## Referencing Previous Values

If you need to reference the value of a price series, variable or expression to any of the previous bars, you need to use a special qualifier after the function name variable.

To reference the previous values just add the number of bars back (you want to reference) enclosed between two backslashes.

For example, to reference Close one bar ago, you need to write *"Close\1\".*

Previous values are referenced relative to the current bar. When your study is verified against the 9th bar of a chart, Close\1\ refers to the Close of the 8th bar.

For example, in the hammer study (on page 50) above the condition "*LRL(Close, 9, 0) < LRL(Close, 9, 0)\1\"* compares Linear Regression Line for the current bar with the Linear Regression Line value one bar ago.

## Using Study Builder

Study Builder is a special tool used to create, manage and load your custom studies into a chart. Unlike the indicator which returns a numeric value for each bar, the study returns a Boolean (true/false) value to identify the bars that meet your criteria. To launch Study Builder, click **Tools** > **Study Builder.**

With Study Builder you will be able to mark a particular market activity and/or specific chart pattern with color markers plotted over a chart**.** This enables you to follow the recurrence and consequences of those market conditions that you want to note.

As soon as it is added to the Studies List, a study becomes part of your study collection, and you can easily edit it and apply it to any chart. The number of diverse studies you can build is virtually unlimited, so you can build as many as you can come up with.

You can quickly select a study from the Studies List. The **Study Properties** pane displays the name, description and status of the selected study, along with the Improvian code that provides instructions on how to evaluate each bar using the study. You can use the Study Builder toolbar to create a new study, edit a selected study, create a copy of it, delete the study or insert it into the current chart.

Tradecision contains several predefined studies (i.e. InsideBar, Islands, Pivots) you can use for a start. Below you can find out how to create a new study that identifies bars with well-known pattern – Wide-ranging day.

To create the Hammer study:

1. On the Study Builder toolbar, click **New**.

The *Custom Study* dialog box will be displayed.

2. In the *Custom Study* dialog box, type "Hammer" in the **Study Name** and **Full Name** boxes.

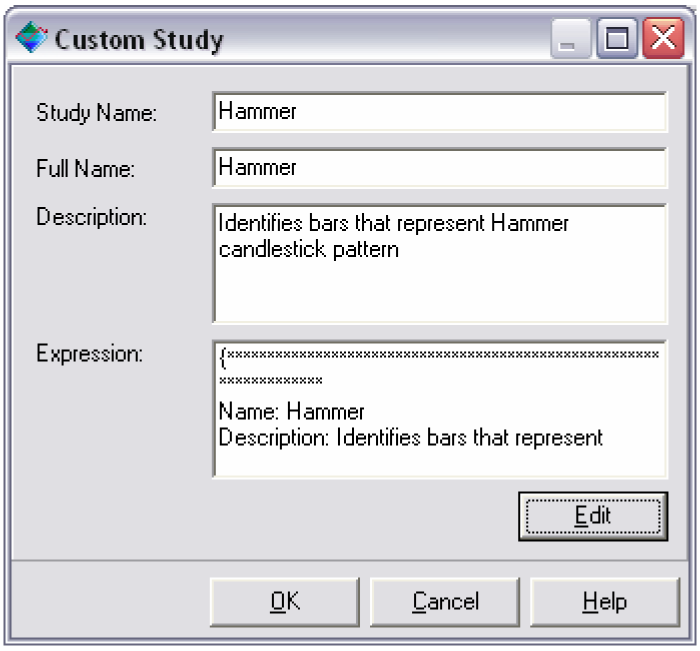
3. In the **Description** box, type "*Identifies bars that represent Hammer candlestick pattern*".

4. Under the **Expression** box, click **Edit**

Improvian Editor will be displayed.

5. In the *Improvian Editor* window, in the **Expression** text box type (or cut&paste through the Clipboard) the study code provided on the next page.

6. Click **OK**.



Study Name: Hammer

Full Name: Hammer

Description: Identifies bars that represent Hammer candlestick pattern

Expression:
```
{***********************************************************
**************
Name: Hammer
Description: Identifies bars that represent
```

7. In the *Custom Study* dialog box, click **OK** to save the new study and add it to the Studies List.

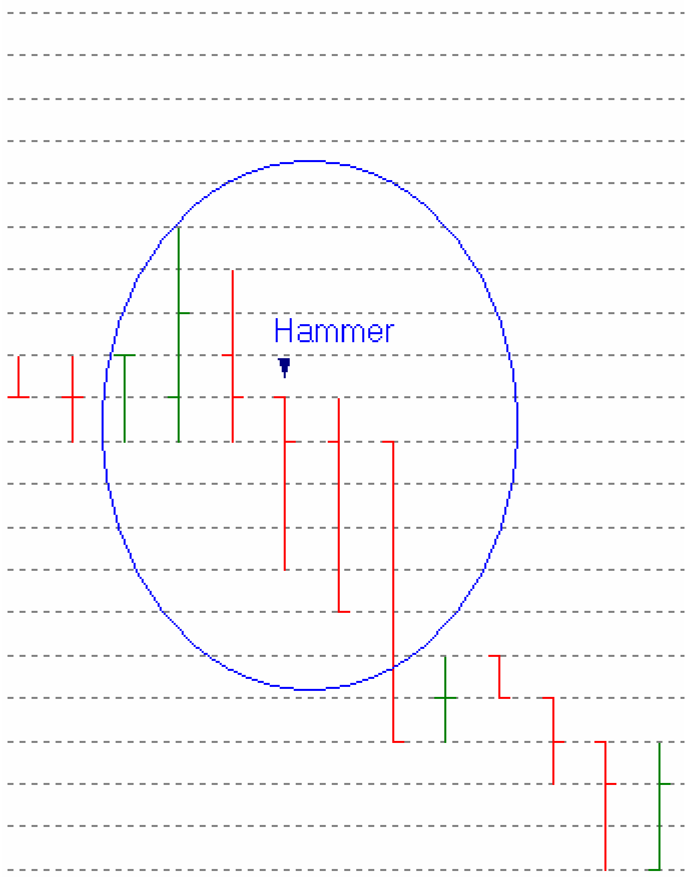
```
{*******************************************************************
Name: Hammer
Description: Identifies bars that represent Hammer candlestick pattern
Last Edited:  [ date ]
*******************************************************************}
Var
        Min:=0;
        Max:=0;
End_var

Min:=Min(Close, Open);
Max:=Max(Close, Open);

if Min > MedianPrice and Open <> Close then
if Min - Low > (Max - Min) * 2 and
High - Max < Max - Min and
LRL(Close, 9, 0) < LRL(Close, 9, 0)\1\ then
This:=true;
else This := false;
else This := false;

return This;
```

The Hammer Study inserted into a minute chart:



In the following chapters, we will provide a detailed overview of the new Improvian elements in this code: nested if...then, comparison operators, logical operators and previous values referencing.

# CHAPTER 5

# Your First Trading Strategy

## Using Strategy Builder

With Strategy Builder you can design, historically test and fine-tune your trading strategies.

To create a trading strategy, you need to formalize your own trading ideas or modify/improve the existing ones. To do so, you need to define the conditions for your buy/sell rules using Improvian Editor.

To launch Strategy Builder, in the **Tools** menu, click **Strategy Builder**.

The *Strategy Builder* dialog box contains a toolbar, a list of your strategies and the parameter pane that displays the parameters for the selected strategy. Using the Strategy Builder toolbar you can create, edit, copy, delete and apply the selected strategy.

Tradecision contains several predefined strategies based on well-known trading ideas. You can review them to get an idea about the strategies' structure and the instructions used in buy/sell rules. Below you can find information on how to create a new strategy based on a wide-ranging day pattern.

Wide-ranging days have a true range that is far larger than the true range for the previous days and are especially meaningful after a strong trend. The simplest formal definition of wide-ranging days is as follows: a wide-ranging day is a day on which the true range for the current bar is twice (or more) bigger than the true range of the preceding N-day-long period. The simplest strategy based on the wide-ranging day pattern is entering into a position if the close of the bar following the wide-ranging bar is more (or less) than the true high (or true low) of the wide-ranging day.

> ***Note**: This rule is too simple to be used as is, so you should be prepared to learn to improve this strategy in the following chapters.*

To create a new strategy:

1. On the Strategy Builder toolbar, click **New**.

The *Strategy* dialog box will be displayed.

2. In the **Strategy Name** box of the *Strategy* dialog box,type "*Wide-ranging days.*"

3. In the **Description** box, type "*Enters position based on wide-ranging days pattern.*"

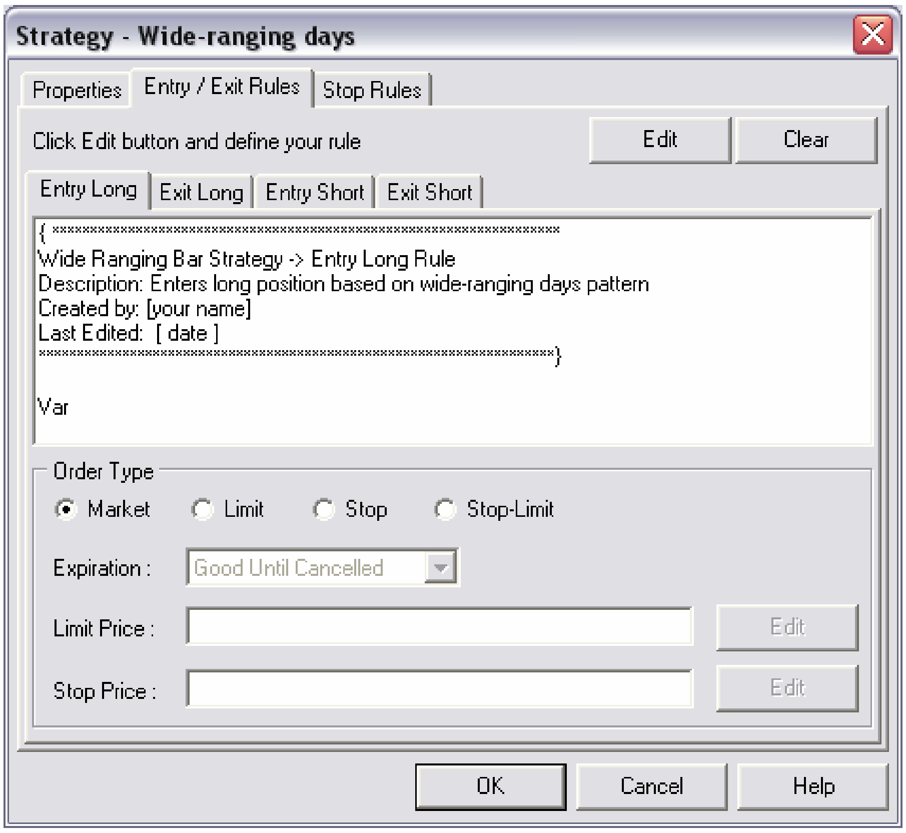
4. In the **Signals** area, select **Only Longs**.

5. Click the **Entry / Exit Rules** tab, select the **Entry Long** tab and then click **Edit**. The **Improvian Editor** will be displayed.

6. In the **Expression** of the *Improvian Editor* window, type (or cut & paste through the Clipboard) the long entry rule code provided below the list.

7. Click **OK**.

8. Select the *Exit Long* tab and click **Edit**. In the Improvian Editor dialog box, in the **Expression** text box type (or cut&paste through the Clipboard) the entry long rule code provided below the list.



```
{ ************************************************************************
Wide Ranging Bar Strategy -> Entry Long Rule
Description: Enters long position based on wide-ranging days pattern
Created by: [your name]
Last Edited:  [ date ]
************************************************************************}

Var
```

Order Type: (● Market)

Expiration : Good Until Cancelled

Limit Price :

Stop Price :

9. Click **OK**.

10. In the *Strategy* dialog box, click **OK** to add the new strategy to the strategies list.

```
{ *****************************************************************
Wide Ranging Bar Strategy -> Entry Long Rule
Description: Enters long position based on wide-ranging days pattern
Created by: [your name]
Last Edited:  [ date ]
*****************************************************************}

Var
        Length:=9;
        VolatilityRatioLimit := 2;

        Sum:=0;
        i:=0;
        BarTrueLow:=0;
        BarTrueHigh:=0;
        PeriodAverageTrueRange:=0;
        VolatilityRatio:=0;
        PrevBarIsWideRanging:=false;

        Array: BarTrueRange[50]:=0;
End_var

{ calculate average true range for the previous Length bars }

For i := 1 To Length do
begin
    { find true low for the i-th bar ago }
    If Close\i+1\ < Low\i\ Then
       BarTrueLow := Close\i+1\;
    Else
        BarTrueLow := Low\i\;

    { find true high for the i-th bar ago }
    If Close\i+1\ > High\i\ Then
       BarTrueHigh := Close\i+1\;
    Else
        BarTrueHigh := High\i\;

    { for each bar save its true range in array }
    BarTrueRange[i] := BarTrueHigh - BarTrueLow;

    Sum := Sum + BarTrueRange[i];
end;

PeriodAverageTrueRange := Sum / Length;
VolatilityRatio := BarTrueRange[1] / PeriodAverageTrueRange;
PrevBarIsWideRanging := VolatilityRatio > VolatilityRatioLimit;

Return PrevBarIsWideRanging and Close > High\1\;
```

***Note:*** *You can make this strategy shorter by using built-in functions. However, the main purpose of this chapter is to show how to create pattern-based strategies using only price data (Close, High, and so on) and Improvian's capabilities.*

## Understanding Loops

A cycle is a recurring process of executing a block of commands. Operators are executed from the start of the block and until the program reaches the end of the block. Having reached the final point, the program returns to the start of the block and the process is repeated. The process can be repeated infinitely.

Cycles are useful if you need to calculate a value based on the number of the preceding bars or to compare one of the parameters of the current bar with the largest /smallest/average value of one of the previous N bars. For instance, you can search for the biggest true range for the previous N bars with the help of the cycle.

## Understanding For...Do Loops

The "For" cycle is, probably, the most frequently used cycle in Improvian. It instructs the program to execute a block of code a specified number of times.

Look at string "buy rule" that begins with the keyword "For." The cycle starts with this string. The word "For" informs Improvian that you are initiating the "For" cycle. After "For" you will see "i" (short for "index"), which is the value that rules the cycle. The value can have any name that is valid for a variable. The variable that rules the cycle is a meter of cycle loops. Improvian takes the number following the assignment operator to start the count. The number that follows the keyword "To" is the last "i" value in the cycle. This means that during the execution of the cycle, "i" will assume values from "1" to "Length" In succession. As long as the value "Length" is specified in the ninth block of the declaration of variables, the cycle will repeat nine times, i.e. all the instructions within the cycle will be executed nine times (for each of the nine previous bars).

In the "long entry rule," at the point where the "For" cycle starts, Improvian places number 1 in the variable "I" and remembers the limit that defines the condition for the stop of the cycle (incidentally, this is the value of the variable "Length"). Next, Improvian proceeds to the next string that compares the Close of the previous bar with the Low of the current bar ("current" in this case means the bar being processed by the cycle, i.e. the bar i bars back) to define the true low based on the results of the comparison. Following this, Improvian consecutively executes all the instructions up to the keyword "end." The keyword "end" defines the end of the body of the cycle and instructs Improvian to increment (or increase) the value of the control variable and start executing the operations over again starting from the top of the cycle. Thus, i becomes equal to 2 and the program returns to the string "For." Then the program compares the value of i with the number following the keyword "To." If the meter of the cycle (in i) is less or equal to the number following "To," the program will execute the cycle again. Incidentally, the process continues until i is greater than Length (i.e. 9).

If you want the control variable of the "for" cycle to decrease and not to increase, replace the keyword "To" with "Downto."

For instance, the above cycle can be written as follows:

*For i:=Length downto 1 do*

In this case, the indexing inside the cycle needs to be changed accordingly as it will be described later.

## Using Variables in Loops

As with most numeric values in the program, you can replace numbers with variables and use them in your "For…Do" cycles the way the variable "Length" was used instead of 9 in the above example. Practically, you will use variables in your cycle parameters as often as numbers, or even more often.

Using variables in the cycle "For...Do" is an effective programming tool, making your programs more flexible.

The control variable of the "For...Do" cycle (for instance, i in For i = 1 to 9 do) is an average variable. The only difference between this variable and the other variables is that Improvian knows how to use this cycle variable for its own purposes, such as maintaining the cycle for as long as required. And as the cycle variable does not differ from the other variables, its value can be set even within the cycle.

To ensure uniformity of your techniques, we suggest naming cycle control variables identically in all of them. Most often cycle control variables are called *index, i, enumerator, e, control, counter, c*. Choose the most suitable version and always use it. This will increase your techniques' readability and reduce the time one will have to spend on understanding the code at a later date.

## Understanding Arrays

In your techniques you will face the task of storing and processing multiple interlinked values.

For instance, to use the cycle for processing nine various variables, you will have to employ an array. An array is a variable which can have more than one value.
An array is designed to provide access to a series of variables under one name. It will be convenient for you to use arrays if you need to store similar information about the number of previous bars.

To use an array, declare it in the variable block with the help of the keyword "array." For example, the following string:

*Array: BarTrueRange[50]:=0;*

declares the array "BarTrueRange" composed of 50 elements and assigns the value "0" to all the array elements during the declaration. In this manner an array can be used to store numeric variables.

If you need to use an array to store Boolean variables, you need to assign the values "true" or "false" to the array elements during the declaration, for example:

*Array: BullBars[14]:= false;*

If you need to use an array for storing string variables, you need to assign an empty string to the array elements or any symbol string, for example:

*Array: DJI_Tickers[30]:= " ";*

You cannot use number arrays to store Boolean or string variables and visa versa. One array can store only elements of one type.

You cannot use any elements or inputs in the declaration of the array. For example, you cannot declare an array in the following manner:

*Array: BarTrueRange[Length]:=0;*

*Array: BarTrueRange[50]:=InputValue1;*

The memory blocks that make up an array are called array elements. For instance, in the "amount" array, amount[1] is the first element in the array, amount[2] is the second element of the array, and so on. An array can contain from 1 and up to 65 535 elements. If you try to write a value that is less than 1 or larger than the array size set during declaration, you will receive the error message "Array dimension must be in the range from 1 to 65535" during the technique's execution.

Indexes are used to derive an individual value for each array element. Index is the number that identifies a block in which one of the array values is stored. For instance, to refer to the first result in our array "BarTrueRange," you will have to write BarTrueRange[1]. Here the index is the number in the square brackets. In this case, we are referring to the true range for the first bar in the array. To refer to the second average result, write BarTrueRange[2]. To refer to the third and fourth average results, write BarTrueRange[3] and BarTrueRange[4] respectively, and so on.

To assign values to the array elements, indicate the element index in square brackets. For instance, the following instruction:

*Ranges[1] := High – Low;*

will assign the value "High – Low" to the first element of the array "Ranges."

## Using a Variable as Array Index

As you already know, if necessary, most numeric constants in Improvian can be replaced with numeric values. A natural and very efficient way of using this option is to simultaneously use one variable as a meter of the cycle "For…Do" and as an array index. For instance, in our example, the variable "i" is used as the cycle control variable and index of the array BarTrueRange.

> ***Note:*** *The variable "i" is also used to compare the values High and Close or Low and Close for adjacent bars, to calculate their true high or true low.*

## Using Comments

### Understanding Comments

To remember why you have written parts of your code this way or another, to remember your ideas and what you are making your technique do in a command, you should insert comments into your source text.

In smaller techniques, featured in this manual, comments are not necessary. However, they will become essential when you start writing larger techniques in which you can easily lose track of your ideas. Comments will help you look through the source text quickly, showing you what exactly is encoded in the technique.

A comment has a start and an end. Everything in between is ignored by Improvian and you can insert any kind of text into the comment without affecting the technique.

*{ This is Improvian  comment }*

The above string is an example of a comment. Every comment starts with "{" and ends with "}".

***Note:*** *In Improvian Editor, comments are highlighted with color.*

## Why do you need comments?

Comments are not necessary for Improvian and the language ignores them. Comments are designed for the trader. They contain advice, assumptions on what you are trying to achieve or a description of how well this technique works. In a comment you can write absolutely anything you want, although the more descriptive a comment is, the more useful it will be for you later.

Most comments written by experienced traders start with several lines providing basic information on the technique. For example, for the MyFastK indicator, we recommend using the following opening comment:

```
{
Name: MyFastK
Description: Calculates Fast Stochastic
Created by: [your name]
Last Edited:  [ current date/time ]
}
```

These several strings inform you about the purpose for which the technique was written, by whom and when. In the source text, you can write comments in the form of notes, for instance:

*{ find out tomorrow why this part don't work as expected }*

or:

*{ enter gap identification code here later }*

Comments can be:

- Multi-string, as in the first example above;
- One-string as the second and third strings above (as long as it is a comment and not an instruction, you do not need a semicolon at the end of the string).
- Closing a string. These comments are located in the same string with the Improvian instruction, most often at the end of the string.

## Advanced Comments Usage

### Comments as a Means of Code Deactivation

Since comments are ignored by the compiler, they can be used to deactivate certain parts of the technique. If something works incorrectly, make a comment on it. Besides, you can insert a note to clarify why this section is commented upon.

Sometimes, just it happens so that something that is supposed to work doesn’t. The reason for that may be that you accidentally commented on this part of the technique. If a very large part of the source code is marked with the commentary color, you have most probably missed a closing parenthesis.

### The Danger of Nested Comments

The most grievous error that may occur while making comments is the so-called nested comments. A nested comment is a comment that is located inside another comment. Let us look at the following part of a technique:

```
If Close > Open then
begin
   BullBarsNo := BullBarsNo + 1;
   Bull_Trigger := BullBarsNo > Limit;  { Limit is entered as technique input }
end;
Else
   BearBarsNo := BearBarsNo + 1;

Sum := Sum + 1;
```

This technique already contains a comment. Let us assume that you want to change this part of the technique to ensure that the operator "if" is not executed at all, and only the increment of the Sum variable is executed. To achieve this, make a comment on all but the last string that you want executed. Intuitively, traders do the following:

```
{
If Close > Open then
begin
   BullBarsNo := BullBarsNo + 1;
   Bull_Trigger := BullBarsNo > Limit;    { Limit is entered as technique input }
end;
Else
   BearBarsNo := BearBarsNo + 1;
}
Sum := Sum + 1;
```

However, for Improvian a comment begins with the opening bracket before "if" and ends into the closing bracket before "end," i.e. the first curly closing bracket found. Therefore, nested comments should not be used.

To avoid the pitfalls of nested comments, be careful when deactivating part of a technique. Incidentally, the solution is to remove the comment { Limit is entered as technique input }. Alternatively, you can make a comment on each separate string, in which case our string would look as follows:

*{Bull_Trigger := BullBarsNo > Limit;      { Limit is entered as technique input }*

This method is efficient because the comment still ends into the } symbols. The additional symbols { in the comment are ignored.

## Special comments

Comments can be used for the visual separation of different parts of code. For instance:

```
{----------------------------------------------}

{!!!!}
```

Comments can be decorated with special symbols. For instance:

```
{******************************
Name: MyFastK
Description: Calculates Fast Stochastic
Created by: [your name]
Last Edited:  [current date/time]
******************************}

{--------------------------------------------
*  Name: MyFastK                            *
*  Description: Calculates Fast Stochastic *
*  Created by: [your name]                  *
*  Last Edited:  [current date/time ]       *
---------------------------------------------}
```

# CHAPTER 6

# Improving Your Strategy

## Understanding Functions

Lengthy techniques are hard to organize and read. A full-size technique can contain five and more pages of code, and you will have to spend a lot of time on locating the part of the code you need. To solve this problem, you can use modular methods of writing techniques. Thus you will be able to subdivide a long technique into sub-techniques whereby each part performs its own individual task.

To make the code of your technique easier to understand, divide the technique into sub-techniques. As a result, each sub-technique in your technique has only one main task while being short and specific. When you overload your sub-technique with too many tasks, you strip it of its individuality.

In order to subdivide your techniques into sub-techniques and to use one and the same code in several techniques, Improvian offers functions (in other words, sub-techniques). Functions can be built using Function Builder.

Functions are designed to divide your techniques into smaller sub-techniques. Functions return the value to the technique that activates them. The activation of a function should envision the return of the function to the variable or use of the return value in an expression.

> ***Note:*** *Tradecision provides separate entry windows for separate entry and exit rules for long and short orders to make your code clear and avoid lengthy code in one place. For the same reason, Strategy Builder provides different windows for entering stop loss and profit take rules for long and short trades.*

## Using Function Builder – Your First Custom Function

Function Builder is a special tool used to create, edit and manage your custom functions. Unlike indicators or studies, a function can be used not only to evaluate something for each bar, but also to perform some general operations, such as finding the minimum of two values or converting a numeric value into a Boolean value. To launch Function Builder, click **Tools>Function Builder**.

Just like with the rest of the building functionality, you can quickly select a function from the Functions List. The **Function Properties** pane displays the name, description and status of the selected function, along with the Improvian code for it. You can use the Function Builder toolbar to create a new function, edit a selected function, create a copy of it or delete it.

After a valid function is added to the Custom Functions List, it can be used in any of your techniques just like any other built-in function.

Let us improve our entry long rule from the previous chapter using the BarTrueRange function.

To create the BarTrueRange function:

1. On the Function Builder toolbar, click **New**.

The *Custom Function* dialog box is displayed.

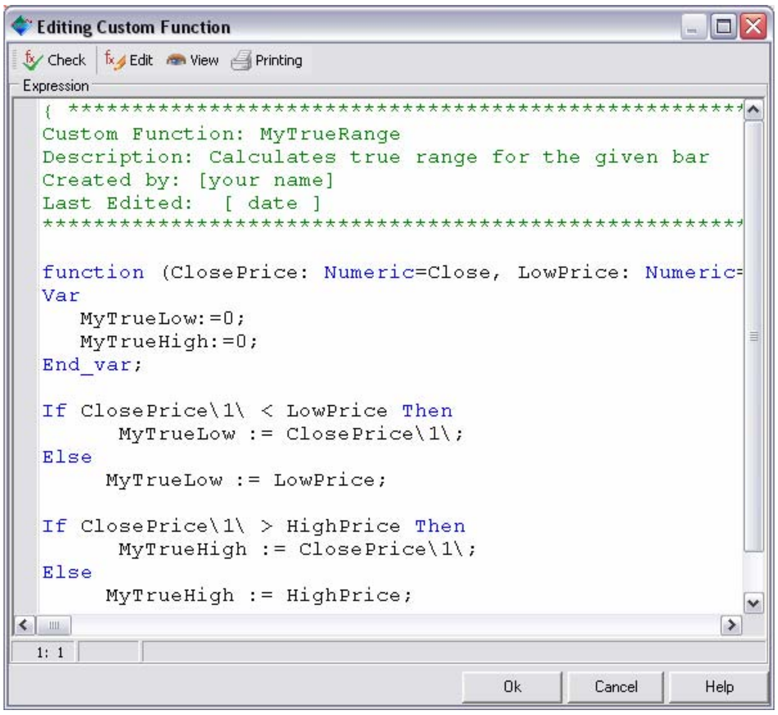
2. In the **Function Name** and **Full Name** boxes of the *Custom Function* dialog box, type "MyTrueRange."

3. In the **Description** box type *"Calculates true range for the given bar."*

4. Under the **Expression** box, click **Edit**.

Improvian Editor will be displayed.

5. In the **Expression** box of the Improvian Editor window, type (or cut and paste through the Clipboard) the function code provided under the list.



```
{ *******************************************************
Custom Function: MyTrueRange
Description: Calculates true range for the given bar
Created by: [your name]
Last Edited:  [ date ]
*******************************************************

function (ClosePrice: Numeric=Close, LowPrice: Numeric=
Var
   MyTrueLow:=0;
   MyTrueHigh:=0;
End_var;

If ClosePrice\1\ < LowPrice Then
      MyTrueLow := ClosePrice\1\;
Else
     MyTrueLow := LowPrice;

If ClosePrice\1\ > HighPrice Then
      MyTrueHigh := ClosePrice\1\;
Else
     MyTrueHigh := HighPrice;
```

6. Click **OK**.

7. In the *Custom Function* dialog box, click **OK** to save the new function.

```
{ *****************************************************************
Custom Function: MyTrueRange
Description: Calculates true range for the given bar
Created by: [your name]
Last Edited:  [ date ]
*****************************************************************}

function (ClosePrice: Numeric=Close, LowPrice: Numeric=Low, HighPrice:
Numeric=High): Numeric;
Var
   MyTrueLow:=0;
   MyTrueHigh:=0;
End_var;

If ClosePrice\1\ < LowPrice Then
      MyTrueLow := ClosePrice\1\;
Else
     MyTrueLow := LowPrice;

If ClosePrice\1\ > HighPrice Then
      MyTrueHigh := ClosePrice\1\;
Else
     MyTrueHigh := HighPrice;

Return MyTrueHigh-MyTrueLow;
```

## Understanding Functions

The first line of a function declares a function, function parameters and the type of the return value. In our example

*function (ClosePrice: Numeric=Close, LowPrice: Numeric=Low, HighPrice: Numeric=High): Numeric;*

the keyword "function" informs Improvian that you are declaring a function. The keyword is followed by a set of parameters in parenthesis. *ClosePrice* is the first parameter of this function. It has the Numeric type and its default value is Close. Similarly, the second parameter *LowPrice* has the Numeric type and its default value is equal to the Low of the current bar, and so forth. If you use more than one parameter in the function, you need to separate the parameters from one another with commas while declaring the function and activating it in the body of the technique.

The parameters are used in the function body in the same manner as the variables are used in a technique's body.

> ***Note**: The naming conventions for functions and parameters are the same as the naming conventions for variables.*

The part of the string after the parenthesis "*: Numeric*" informs Improvian that this function returns to the activating technique a value of the Numeric type. This also means that in the function body the value of the expression after the keyword "Return" should be numeric, for, otherwise, a type mismatch error will occur.

The following string of the function is the string that informs Improvian which value will be returned from the function.

> ***Note**: To return a value from a function, the keyword "Return" is used, which, among all, denotes the end of the function.*

Now, in order to be able to use the function in the body of any technique, you need to activate it in the following manner:

*MyTrueRange(Close, Low, High);*

A function can have as many parameters, as required. However, you need to make sure that the parameters defined by you during the procedure activation match the type and sequence of the parameters that you defined in the string of the function declaration "*Function.*"

During the function declaration, you can use different variable names. However, they should be of the same type. In other words, in the example *MyTrueRange* we can use variables or functions rather than the variables *ClosePrice, LowPrice* and *HighPrice*, which were specified during the declaration. It is important that the types coincide, and not the parameter names.

Functions can have no parameters at all. In order to activate a function that has no parameters, you do not have to write anything in parenthesis. For instance: *RandomValue().*

One should bear in mind, that a function receives parameters in the same order in which they are listed during the function activation. In the string of the function declaration "Function," the parameters should be listed in the same order, as during the function activation. For instance, *MyTrueRange(Low, Close, High)* will return the wrong result.

## Using Functions in Strategy Builder

Next, to simplify our entry long rule with the help of the function MyTrueRange, do the following:

1. To launch Strategy Builder, from the **Tools** menu, select **Strategy Builder**.
2. From the **Strategy List**, select **Wide-ranging days** and select **Edit>Entry Long**.
3. In the **Expression** box in the *Improvian Editor* window, delete (or comment on) the following lines of the code:

```
BarTrueLow:=0;
BarTrueHigh:=0;
    …
    If Close\i+1\ < Low\i\ Then
       BarTrueLow := Close\i+1\;
    Else
        BarTrueLow := Low\i\;

    If Close\i+1\ > High\i\ Then
       BarTrueHigh := Close\i+1\;
    Else
        BarTrueHigh := High\i\;
```

And instead of

```
BarTrueRange[i] := BarTrueHigh – BarTrueLow;
```

write

```
BarTrueRange[i] := MyTrueRange(Close\i+1\, Low\i\, High\i\);
```

4. Click **OK**.

Now your strategy rule has the following code:

```
{*************************************************************
**
Wide Ranging Bar Strategy -> Entry Long Rule
Description: Enters long position based on wide-ranging days pattern
Created by: [your name]
Last Edited:  [ date ]
*************************************************************
**}

Var
        Length:=9;
        VolatilityRatioLimit := 2;

        Sum:=0;
        i:=0;

        PeriodAverageTrueRange:=0;
        VolatilityRatio:=0;
        PrevBarIsWideRanging:=false;

        Array: BarTrueRange[50]:=0;
End_var

{ calculate average true range for the previous Length bars }
For i := 1 To Length do
begin
    BarTrueRange[i] := MyTrueRange(Close\i+1\, Low\i\, High\i\);
    Sum := Sum + BarTrueRange[i];
end;

PeriodAverageTrueRange := Sum / Length;
VolatilityRatio := BarTrueRange[1] / PeriodAverageTrueRange;
PrevBarIsWideRanging := VolatilityRatio > VolatilityRatioLimit;

Return PrevBarIsWideRanging and Close > High\1\;
```

By using your new MyTrueRange function you can avoid using BarTrueRange array, thus simplifying your technique even further. To know how this can be done, analyze the code below:

```
{
*************************************************************
****
Wide Ranging Bar Strategy -> Entry Long Rule
Description: Enters long position based on wide-ranging days pattern
Created by: [your name]
Last Edited:  [ date ]
*************************************************************
*****}

Var
        Length:=9;
        VolatilityRatioLimit := 2;

        Sum:=0;
        i:=0;

        PeriodAverageTrueRange:=0;
        VolatilityRatio:=0;
        PrevBarIsWideRanging:=false;
End_var

{ calculate average true range for the previous Length bars }
For i := 1 To Length do
    Sum := Sum + MyTrueRange(Close\i+1\, Low\i\, High\i\);

PeriodAverageTrueRange := Sum / Length;
VolatilityRatio := MyTrueRange(Close\2\, Low\1\, High\1\) /
PeriodAverageTrueRange;
PrevBarIsWideRanging := VolatilityRatio > VolatilityRatioLimit;

Return PrevBarIsWideRanging and Close > High\1\;
```

## Scope of Variables

Now that you know about functions, you can learn how Improvian manipulates the variables between functions and techniques. The question arises, why we pass "Close" as a parameter when we activate the function MyTrueRange? Why could we not directly use the "Close" value in the function? Are not both "Close" and function MyTrueRange part of the same technique?

Parameters have to be used in functions due to the so-called visibility range, which defines whether the functions in a technique can see a certain variable or value or not.

For example, if you have changed a technique in such a way that it cannot pass the values Close, Low and High as parameters of MyTrueRange, your technique will try to execute MyTrueRange, causing an error message.

The reason for that is the following: the price values are accessible only in those techniques that deal directly with price data, as functions can just as well be used in those places in which Improvian cannot guarantee the availability of the price data. Additionally, a variable declared in a technique is inaccessible within a function because the technique does not have to activate exactly this function, and, therefore, Improvian will not be able to guarantee that the rest of the techniques will contain the same variables.

However, you should remember that the reverse is also true. In other words, a variable declared in one function is inaccessible in another function or technique. The variables MyTrueLow and MyTrueHigh declared in MyTrueRange are accessible only within MyTrueRange and will not be accessible in the activating technique. Thus, if you do not expressly specify the value of Close (or any other variable specified within a technique) as a parameter, the function MyTrueRange will not be able to access it. If a function still tries to access the variable, Improvian cannot recognize this and displays an error message.

In the case of the MyTrueRange procedure, the variable MyTrueLow is local with respect to MyTrueRange. Limitations such as this one are fair for all trading languages and were introduced due to the reasons described below.

As long as the variable cannot be seen anywhere, with the exception of the function where it is declared, you will never have to worry about some other function changing its value without you knowing about that. In addition, local variables make functions all-sufficient as if everything that a function needs were combined in one block. If the value of a variable used in the function causes the technique to crash, you will know exactly which of the modules needs to be checked. You will not have to search through the entire technique to find the cause.

## Improving Strategy Ideas

A strategy that is based on wide-ranging days can be improved by introducing a term of signal range. The upper boundary of the signal range is calculated as the highest maximum of the time period from N1 day and up to the wide-ranging day.  The boundaries of the signal range are defined by the highest and lowest of the true minimums and maximums on the N1-day interval up to the last day with a wide range plus N2 after it.

The entry into the long position (and exit from the short position) occurs when the closure exceeds the upper signal range limit. In a similar manner, the entry into the short position (and exit from the long position) occurs when the closure is below the minimum signal range.

## Understanding While… Loops

Unlike the "For" cycle, whereby the cycle index is defined by the cycle borders, the "While" cycle continues being executed until the condition of its execution is met. The condition is a Boolean expression similar to those that you used with the "If" operator. In other words, any expression that can be true or false can be used as a condition in the "While" cycle.

For example,

```
{To find the nearest bar which is greater than the target}
while C\i\<=target do i:=i+1;
```

A technique positions the value of the control variable before the start of the "While" cycle. This is important because it guarantees that the initial control value is a non-zero one. If it happens so that initially the control variable was equal to 0, the technique will never reach as far as the operator block in the "While" cycle, because the condition of the cycle execution will immediately be considered false. Mistakes like this one, despite their seeming plainness, sometimes take up a lot of traders' time.

> ***Note:*** *There is a variety of cycling methods designed for various programming situations. Although experience is always the best teacher, certain things still should be borne in mind during the selection of the cycle construction. When you want a cycle to be repeated a specified number of times, the best choice will be the "For...Do" cycle. If you want the cycle to be executed until it encounters a new condition, use the "While" cycle.*
>
> *Pay attention not to write cycles that never end! For instance, if you are writing a "While…Do" cycle with a condition that never receives a true value, the cycle will repeat eternally. When this happens, it seems that your technique has locked up your computer. In fact, your technique is simply executing your eternal cycle without ever going to abandon it. Oftentimes, the only way out would be pressing Ctrl+Alt+Del to force the technique to stop.*

# CHAPTER 7

# Debugging Trading Techniques

There are two types of programming errors that can unexpectedly occur in your techniques.

The first type is the execution error, which usually crashes the technique and causes an error message.

The second type is the logical error, which is far more insidious and hard to detect. Logical errors occur when you write a string of the source code that does not perform as expected. It is quite possible that you have been adding the wrong variables, using incorrect logical operators or forgetting to add brackets to the expressions to define the operator precedence.

This chapter is dedicated to those technique execution errors that, being incorrectly processed, result in error messages displayed on the screen during a technique's execution, or in aborting the technique. The most popular execution error is the attempt of your technique to divide a certain number by zero.

## Detecting Execution Errors

According to computer experts, no faultless software programs exist. As you start writing full-size techniques, you will see that for yourself! As your techniques become longer, you will make sure that the error search and recognition can be a hard and complicated task.

When your technique consists of a few strings only, you can check each string of code and detect the inaccuracy. However, when your technique grows to include dozens and even hundreds of code strings, it is unlikely that you will enjoy the idea of reading through every string of code. Instead, you will most probably want to locate the part of the technique where the error has occurred and thus speed up the search.

For this purpose, the Tradecision application provides a simple-to-use debugging tool called Trace Watcher, which will help you easily find errors in the programming code.

## Errors in Your Techniques

When writing complex techniques, be ready to spend a lot of time on error recovery, from simple spelling mistakes to logical errors.

Errors are inevitable. Even the developers of Improvian receive error messages every day. You should not be embarrassed by errors. Consider them training tools or polite reminders. Improvian tries to identify the cause of an error and locate it as precisely as possible, displaying a corresponding message on the Messages panel.

There are four types of errors that can occur in your techniques: syntax, semantic, run-time and logical errors. The following chapters describe how to avoid and fix them.

## Error messages

To understand why syntax errors occur, replace the following first three strings of the Var block (right after the comments) in our entry long rule and press F5:

```
Var
  WideRangingDayLocated := false
  Control := true;
```

After the verification, you will see a common error message in the first string of the **Messages** panel:

*3 | 4 | At "Control" missing ";" | Sintax error*

An error message contains the following information:
<3> - a string in which the error occurred. In this case it is string 10, although errors may have occurred in other strings as well.
<4> - the number of the string symbol where the error occurred according to Improvian.
<At "Control" missing ";"> - a description of the error.
<Syntax error> - the type of the error. In this case, it is an error that occurred during the syntax analysis (i.e. a syntax error), or a similar error.

Despite all the information received, it may still be unclear what exactly is wrong. However, Improvian will try to provide as much information as possible on the reason for it being unable to continue building the technique.

In fact, the error is in string 2 (the number of the string is indicated talking the comments into consideration). However, Improvian is flexible and it does not detect that something is missing until it reaches string 3.

> ***Note**: You can use this flexibility and make the change in string 3. However, this will reduce the readability of the code).*

## Syntax Errors

The type of an error is also important. Syntax analysis, or parsing, allows detecting a syntax error, i.e. an error resulting from the omission of an Improvian punctuation symbol. Syntax defines the method to combine language elements. In other words, syntax defines the sequence of combining language elements. Therefore, a syntax error means that two language elements that, according to the Improvian syntax cannot follow each other, Have been placed next to each other. Incidentally, we can see the omission of a "semicolon" symbol at the end of string 2.

To correct this error, we need to edit the source text and fix the invalid part. In this case, we need to edit the rule all over again and add a semicolon at the end of string 2. Even if you take a very close look at string 3, you will not be able to see anything incorrect there. However, if you look a little way back, you will see the error and fix it. If you have been unable to locate an error in the string indicated by the compiler, remember the syntax errors. By the way, the omission of a semicolon is one of the most common errors in techniques.

To facilitate your search for errors, the numbers of the current string and column (under the cursor) are displayed to the left in the status bar (at the bottom of the edit panel).

The number of the string can be indicated inaccurately. If a semicolon is missing, the error can be detected only in the next string. This assertion is also true for the other types of errors. In any case, the cause of the error is found somewhere very close to the indicated coordinates.

As you can see, the **Messages** panel contains errors that occurred because Improvian could not correctly identify the string of the first variable’s definition due to the absence of the semicolon. The best thing to do would be to correct the first error and then check the technique again by pressing F5 or double-clicking the toolbar button **Check**. As in this case, it often happens that the first error is the only real error, and Improvian also indicates other errors that sequent from it.

To correct the technique, you need to fix the errors. In our example, this means adding a semicolon. Edit string 2 by adding a semicolon at the end of the string.

Understandably, not all the errors can be fixed so easily. When you receive an error message, try to understand it to see where in the source text the mistake occurred.

## Semantic Errors

Another example of simple errors is spelling mistakes. Change a string in our rule to:

*Control := not(WideRangingDayLocatd) and (N2 <= BarsBackLimit);*

and press F5 to check the technique. In this case, the unintentional spelling mistake in the variable name is the omission of the last letter "e."

Improvian searches for the variable WideRangingDayLocatd, but it cannot find it and displays the corresponding error message: "The variable WideRangingDayLocatd is not declared." Sometimes, it is difficult to spot the misprint in a long variable name at first glance and it seems that Improvian is wrong. In this case, we are dealing with a semantic error. Semantic errors occur when the code is in compliance with the rules of the language (the code syntax is correct), but the semantics (meaning) cannot be defined unambiguously or does not comply with the previous code (the task).

To fix this error, edit the source text again and replace the word WideRangingDayLocatd with the one that would make the program correct. The most common semantic errors are misprints in function parameters or function types, for instance, the activation of CrossAbove(Highest(High, 13)) where the second (or first) parameter is missing.

## Run-Time and Logical Errors

Let us assume that you have decided to use the simplest variation of the FastK indicator, provided in Chapter 2:

*return (Close - Lowest(Low, 9)) / (Highest(High, 9) – Lowest(Low, 9))* 100;*

Sooner or later, for a certain security and period, the indicator will encounter a situation whereby Highest (High, 9) will be equal to Lowest(Low, 9), and the difference will be equal to zero. In this case, when you try to insert the indicator into a chart, you will receive an error message. This may confuse you, as your indicator has been functioning without any errors for many other securities or even functioned well for the same security in the past. In such cases you have to deal with run-time errors.

## Run-Time Error Messages in Tradecision Techniques

1) "The index is outside of array bounds." this means that an array element outside the array was called. For example, there is a declared array consisting of 100 elements and there is a call to the array element in the program
Var array:ar[100]:=0; end_var
ar[101]:=10;

2) "Division by zero."  This error message means that a division into zero has taken place. For instance, the indicator with the text return (C-O)/(H-H\1\); will issue this message during the insertion if the expression H-H\1\ will produce value equal to 0. To avoid this, use constructions of the IFF type, so that you can convert the initial expression retaining its meaning into the following one: return IFF(H-H\1\=0,0,(C-O)/(H-H\1\)). If the division-into-zero situation arises, the expression will return 0.

3) "Exception thrown by function". This message means that in the attached DLL function an exception was generated during its activation.

4) "Floating-Point Operation Error". This message means that a floating-point operation error has occurred. This message is issued if the result of an arithmetic operation for the module exceeds the admissible value of 1.7 E 308. This usually happens in case of accumulated summation or multiplication, for instance, when calculating an indicator that uses the result of its former calculations in its expressions. In this case the sum will increase infinitely until it exceeds the bounds. Check the expressions carefully to see where in the expression the required devisor could have been omitted: return (h+l+c)* this\1\ / this\2\.

In all the above cases, Improvian’s debugging options to establish the cause of the error.

## Detecting Run-Time and Logical Errors

Tradecision provides a simple debugging tool for detecting run-time and logical errors – Trace Watcher.

To locate run-time and logic errors using Trace Watcher, change the code of the last operator If in our technique to the following:

```
if WideRangingDayLocated then
   This := CrossAbove(Close, Highest(High, N2+N1+1));
else
   This := false;
```

Some traders could easily write this code and check whether Close exceeds the value of the highest High for the previous N2+N1+1 bars. At first glance, the formula seems correct and its verification does not expose any errors. However, after inserting this technique into a chart you will not detect a single signal. You can go through the code over and over again checking the logic and you will still be unable to find the cause for this system behavior. In such cases, use Trace Watcher to find out whether it is true that the input condition is met for neither of the bars.

Trace Watcher enables viewing the variable values during the execution of a technique. To display the variable values in one of the windows, Trace Watcher employs a special Improvian function, "print" or "println."

## Function print()

The function print() displays the text in between double quotes or variable values. The text is displayed in one of the Trace Watcher windows.
To display certain special symbols, use ex-cape-sequences in print(), – sequences of symbols that start with a left slash.

The following are three examples of how the print() function is used.

*Print(0, "Entered short position using secondary condition.");*

*Print(0, BarNumber());*

*Print(0, "Current bar number is: " + NumToString(BarNumber(), 0));*

Here the first parameter is the number of Trace Watcher window, which can change from 0 to 9.

The second parameter is the text that you want displayed on the Trace Watcher screen. If the text is within double quotes, it will be displayed on the compiler screen without change. If the text is the name of a variable or function, the compiler screen will display their value. The third output option employs the possibilities of Improvian for working with strings while formatting the output. The functions for performing operations with strings will be described later.

The print() function can display several strings of text with the help of escape-sequence \n (\n — new string). The table below lists escape-sequences that can be used in the print() function.

## Escape-Sequences in the Print() Function

| Sequence | Value |
|---|---|
| \n | Enter (new line) |
| \t | Tabulation (indent) |
| \” | Quotation mark symbol - “ |
| \\ | Backslash - \ |

For instance, the following activation of the function print() will result in two strings being displayed in the compiler window.

*Print(0, “\n The last entered position is: \n \t \” BUY \” “);*

## Function Println()

The function println() is similar to the function print(), but it displays its parameters from a new line. The following two activations are absolutely similar:

*Print(0, "\n New string");*

*Println(0, "New string");*

## Working with Trace Watcher

In order to see the cause of an error, extend the code using the following commands:

```
if WideRangingDayLocated then
begin
      This := CrossAbove(Close, Highest(High, N_after+N_before+1));
      Println(0, "Bar #: " + NumToString(BarNumber(), 0));
      Println(0, "Close: " + NumToString(Close, 2) + "\t HH: " +
NumToString(Highest(High, N_after+N_before+1), 2));
End;
else
   This := false;
```

Now save the code and insert the strategy into the chart. Then open Trace Watcher by selecting **Trace Watcher** from the Tools menu. In the zero tab, you will be able to see the Close and Highest High prices for those bars that meet the specified criteria.

While scrolling down to see the values of the bars, you can see that your Close is always a bit smaller or at least equal to the Highest High. Probably, the prices for this security always go up after the wide-ranging days. You can try to insert the strategy into a couple of other charts to verify this assumption. However, the picture you will see will always be the same.

You can examine some of the bars in your chart using the bar numbers displayed in the Trace Watcher window to see what is going on in the market. After verifying the Close and highest High values for several bars, the reason for no signal having been generated is clear: you are trying to compare the Close with the High, including the current bar, which makes your condition impossible to execute. The solution to this problem is obvious: you have to replace the Highest (High, N2+N1+1) with the Highest (High\1\, N2+N1+1) in order for the current bar not to be included in the calculation of borders of the signal strategy range.

Now you can easily delete the strings with println and "begin…end;"from your strategy.

> ***Note**: To reduce the size of the code (the number of the strings) we performed a string combination operation. More information on operations with strings will be provided in the next chapter.*

## Working with Strings

### Merging Strings

Sometimes, you will find yourself in a situation when you have to combine two or more strings into one string. For example, you may have to combine the name and value of your string variable that you are trying to display in the debugger window. To derive a single standalone string, you will have to combine these strings by attaching one string to the end of the next string. The new string must fully incorporate the content of the two initial strings. To solve this task, use the Improvian concatenation operator, represented by the **+** (plus) symbol. This symbol is present on your keyboard. For instance, to combine two strings, type the following:

```
"StringVar value is: " + StringVar;
```

A simple string combination is not, however, a complete expression. You should inform Improvian where you want the new string saved. To do so, use the Improvian assignment operator represented by the colon and sign of equality (**:=**). For strings, the assignment operator is used in the same manner as for numeric variables.

For the purpose of our example, use the following command:

```
FullString := "StringVar value is: " + StringVar;
```

To see how it all works, let us consider the following strings of the programming code:

```
Var
    Definition:="String value is: ";
    FullString:="";
    StringVar:="";
End_var;
FullString := Definition + StringVar;
println(0, FullString);
```

In this fragment of the technique, three variables are declared first, then values are assigned to two of them: StringVar and Definition. In the sixth string, these strings are merged together and the result is assigned to the third variable FullString.

## Substring Extraction

In the same manner as you combine several strings to create a long string, you can divide strings into shorter strings. These shorter strings are called substrings. In order to clip out any part of a string, Improvian suggests using the Substring method.

The Substring function returns the specified number of symbols from the beginning of the string along with the specified symbol to the end of the string. It cuts the specified symbols out of the string. For instance, if we add the following expression to the end of the previous example:

*CurrentTicker := Substring(TickerList, 5, 4); {CurrentTicker value is GOOG}*

## Locating a Substring

Now that you know how to extract a substring from a larger string, you need to learn to locate the string you need. Let us assume that your string contains a ticker list and you want to find the ticker GOOG in it. To solve this task of locating a substring in a string, use the function InString. For instance,

*TickerPosition := InString(TickerList, "GOOG"); {TickerPosition value is 5}*

## Converting String to a Number

As we already know, there is a big difference between numeric values and text strings even if a text string contains numbers. Numeric values can be used in mathematical operations and strings cannot. Fortunately, Improvian offers a convenient function called StringToNum, which enables converting strings with numbers into numeric values that can be directly used in mathematical operations. You can also convert numeric values into strings with the help of reverse conversion using the NumToString function, thus expanding your capabilities of processing the calculation results.

To convert a string into a number, use the StringToNum function, as it is shown in the following fragment of technique code:

*Number := StringToNum(StringVar);*

# 8 Tips that Will Help Make Debugging Easier

If the height of the source text of your technique is greater than the height of Improvian Editor screen, and you are writing a technique for the first time, you will most probably be unable to just sit down and encode the entire technique from scratch, making it work perfectly.

The human nature is such that errors are common during the writing of techniques. Improvian does notice spelling mistakes, but there can occur other types of errors (or defects) in techniques. No errors may be found in a technique during the verification, but when the technique is run, unexpected behavior occurs despite the fact that the technique simply executes the trader’s commands. It often happens so that the trader forgets about some of the essential things while using Improvian. When that happens, it would be reasonable to revert to this chapter and read once again the overview of those methods that can be used to solve your techniques-related problems. You should do so before you start making phone calls, sending e-mails, or discussing your problem in any other way. Thus you will save your time and acquire the necessary skills faster.

## Use the Built-in Improvian Functions in Full

Improvian offers a large number of built-in functions to manage the progress of a technique and to define the necessary conditions. These functions operate properly and often simplify the code. For instance, when writing your own "While" cycle, you can use the functions BarsWhile(), BarsSince, ValueWhen(), and instead of writing the "For" cycle you can use the functions CountIf(), IsIncreasing() or IsHighest();

## Do Not Try to Fix All the Bugs at Once

Fix your bugs one at a time. If there are many, try to avoid the temptation of fixing all the errors in one editing session. Instead, fix one error, verify the technique, and execute it to see how it works. Often, the solution to one problem automatically solves all the other problems too. In any case, start with the problems that result in infinite loops or logical inability of the technique to reach the keyword "return."

Remember that trying to fix many bugs at a time you can breed new errors, whereas it would be much easier to control the situation if you remember that you have changed, for example, strings 12 and 17 of your source text only.

## Break Long Code into Functions

If your source text tends to become quite long, try to break it into parts, or rather, into functions. As shown in the previous strategy-related example, breaking the code into functions makes the code simpler and easier to understand. Besides, the breakdown into functions allows simplifying the bug-tracking and fixing process.

## Format Your Code an Appropriate Way

Use comments to write short descriptions of subtasks and break the code down into separate sections.

Avoid the temptation of writing several commands in one string if you are not sure that this part of code functions well.  Your code will be longer, but if you receive a message about an error occurring in one of these strings, you will know where to look for the error. Finally, when you are sure that your code functions as expected, you will be able to fit the whole of it into one string, should you want to do so.

Try to adhere to the formatting style used in the built-in functions or create your own style that would help you visually check the compliance of the "begin…end" and "if…else" blocks in the in-build "If" operators. At first the indents may seem excessively long. However, they are irreplaceable for fast definition of code structure and search of a missing "end," for example.

There are many reasons why you should follow the above tips, and the most important of them is that this way you will improve the readability of your code. The easier it is to read your code, the easier it is to find the error.

## To Locate Logical Errors Fast, Discuss Your Technique with a Colleague

One of the best ways to find an error at once is to show your technique to another trader. The other trader does not know what you have been trying to encode and you will have to explain it to him, discussing with him the logic of your code. As you do this, you will understand which part of your code does not meet your intentions.

## Sometimes It Is Useful to Disable Part of Your Code Using Comments to Localize an Error

Let us assume that there is an error in the function GetMySignal(), but you do not know where exactly: at the beginning of the code, somewhere before the numerous activations of the mathematical functions, or somewhere at the end of the code.

One of the ways to know this is to comment on, for instance, all the activations of the mathematical functions. This way you will narrow down the search space to be analyzed. If the error persists, you will know that it is located outside the commented code.

## Use "Return" to Reduce Your Technique

Another way to identify the location of the problem is to temporarily use additional activation of the "Return" function in the middle of the code. This way you will be able to debug even a program that is caught in an endless loop. At a certain point, insert the function "return 0;". This will stop the technique immediately. In this manner, you will be able to divide your code into parts. If your technique stops at the control point, the error is located somewhere further. If the technique does not stop, the error is somewhere before the control point.

## Use Trace Watcher to Track Down Variables

Sometimes, a technique acts as if it went astray. This happens because those values that you expected to find in the variables are located elsewhere. To verify that the variables do not contain any unforeseen excessive values, insert the instruction print() to display the variable values on the Trace Watcher compiler screen. Do not forget to delete the activations of print() when the final version of the technique is ready.

## Make Your Comments Concise and to the Point

Verbose comments take up a lot of time and the valuable screen space during technique analysis. That is why you should keep your comments concise.

When you are developing a complex technique during several days, the following comments will be pertinent: "this is what I thought," "this is how I planned this", "in this case the result is unknown," "here I need to improve/reduce/change the technique," or "when used in alerts, this needs to be changed."

Such comments will further save your time and help you find errors that occurred as your vision of the technique changed and various parts of your technique correspond to various parts of your idea."

# CHAPTER 8

# Advanced Possibilities of Improvian

## Multi-Data Strategies - Referencing Price Data of Other Symbols

You can significantly improve you trading strategy by using Improvian’s powerful ability to reference price or technical indicator information from any security or data feed you use in Tradecision. For example, in many cases it’s worth comparing a stock’s price with the overall exchange index before making a buy or sell decision.

You do not need to add the required symbols to your chart. If the required symbol data is available in the Data Manager database, you can refer to this data in any technique using Improvian.

The **External Function** allows obtaining an expression value (the first parameter) based on the price data of another stock (the second parameter).

For example, to obtain the Dow Jones Industrial index from Yahoo Finance, you can use the following syntax:

*DJIClose := External(“Close”, "^DJI");*

Additionally, you can calculate any indicator and even any numerical expression using the price data of the specified symbol. For example, to obtain a short SMA for QQQ's Close, you need to use the following syntax:

*QQQ_SMA9 := SMA(External("Close", "QQQ"), 9);*

The number of the external price data you reference in your technique is not limited.

Using the External function you can easily compare one security against another one, compare market indices, or look at the relationship between different groups of issues.

## Using Custom Time-Series

Custom Time Series is a method that allows you to add external data for market calculations (for example, in Indicator Builder). The data should be stored on your local hard drive in a text file.

After they are created, the **Custom Time Series** are added to the Custom Time Series category available in Improvian Editor. You can use custom time series to build your custom indicators, strategies and neural models.

For example, very often you can improve your model performance using fundamental factors and market indices as model inputs. Similarly, with custom time-series you can use 3-rd party time-series, such as the values of proprietary indicators and studies.

## Using Built-In Studies

The following analytical studies are available with a single-click:

**Auto-trends**
The Auto-trends tool is used to automatically draw trends lines over the chart. The tool provides automatic trend lines identification for minor, intermediate and major trends.

**Pivots**
This study identifies and marks a turning point in the trend on a chart. Tradecision identifies Minor, Intermediate or Major turning points. Pivots are useful as the start or end points when drawing objects (such as Fibonacci retracements) or other analytical studies are used..

**Single Day Patterns**
This kind of patterns is used for the identification of single-day patterns, such as runaway and thrust days.

**Reversal Patterns**
Automatic identification of reversal patterns, such as V-top, double-top, head and shoulders.

**Elliott Wave analysis**
An exceptional tool for those traders who have used counting Elliott waves to make trading decisions. The tool includes automatic identification of a-b-c corrections.

**Fibonacci Clusters**
The tool is used for calculating price levels using pivots and Fibonacci retracements. It allows you to determine the probability of a trend reversal for the each price level.

**Noise Removal**
The Noise Removal tool allows you to create and analyze the smoothed curve of a price chart.

## Using DLL Functions

With the help of DLL Manager you can refer to functions/subroutines stored in .dll files. DLL stands for Dynamic Link Library and refers to a file with a .dll extension. DLLs are used by applications to store functions and data required for proper operation.

You can register a function using the Register Function tool. Once properly registered, the function can be used as a standard function, for example, to create a custom indicator.

To register a new function:

1. In the **Tools** menu, click **DLL Manager**.

   The *Register function* dialog box will be displayed.

2. Click **Add.**

3. In the *Creating new function* dialog box, click **Select Library** to locate a .dll file on your PC.

4. In the **Library Function** list, select the function you want to register.

5. In the **Function Name** box, enter a name for the function.

6. In the **Return Value** list, select the type of the value to be returned. The following options are available:

   - int (Integer Value)
   - float (Floating Point Value)
   - double (Floating Point Value with Double capacity)
   - bool (Boolean).

7. In the **Parameter types from first to last** list, select the type of the function’s parameter and click **Add**.

8. Click **Save**. You can select several parameter types.

The function will be added to the **Registered functions** list and the **External DLL functions** category of the Improvian Editor.

## Analyzing Technique Interdependencies

**Technique Interdependencies** is a special dialog box intended for reviewing all trading techniques available in Tradecision. The functionality enables analyzing dependencies between techniques and the ability to edit them.

To Manage Technique Interdependencies:

1. In the **Tools**, click **Technique Interdependencies**.

2. In the **Technique Interdependencies** dialog box, select the select the Model, Strategy, or any other required technique.

3. Review the dependencies that you are interested in by selecting *Depends On* and *Used In* tabs.

You can use the following toolbar buttons:
- **Locate in the list**.
- **Select previous techniques**.
- **Show only invalid techniques**.

## Importing EasyLanguage© Techniques

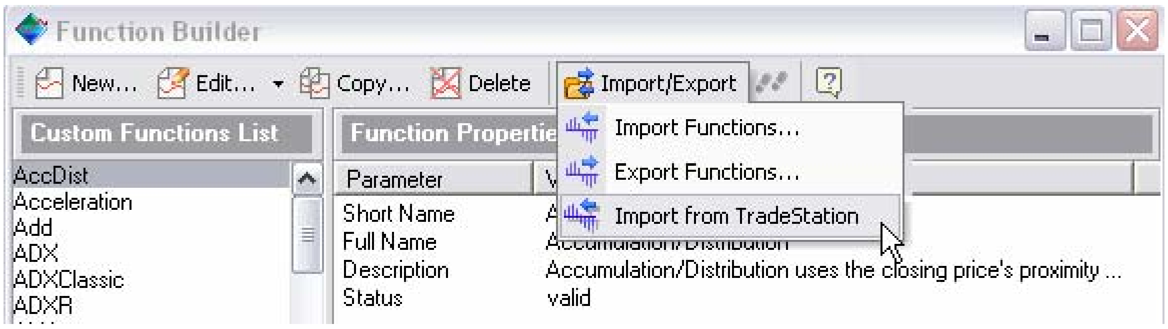
With **Function Builder**, you can import TradeStation's functions. You can only convert functions from EasyLanguage. However, there are several problems that prevent converting the EasyLanguage functions into Improvian.

- Links to unimplemented techniques in Tradecision cannot be converted without receiving an error massage. You need to try import the unimplemented function at first.
- A DLL function call will not be translated correctly. For example, JMA value1 = jrc.jma.2k(series,len1,phase1);
- Compiler Directives will not be translated correctly. For example, #BeginAlert.
- Using unsupported constructions, for example, RawAsk of Leg(count) is not supported. There is no list of such constructions available.

To import a function from TradeStation©:

1. From the **Tools** menu, select **Function Builder**.

2. Click **Import** and select **Import from TradeStation**.

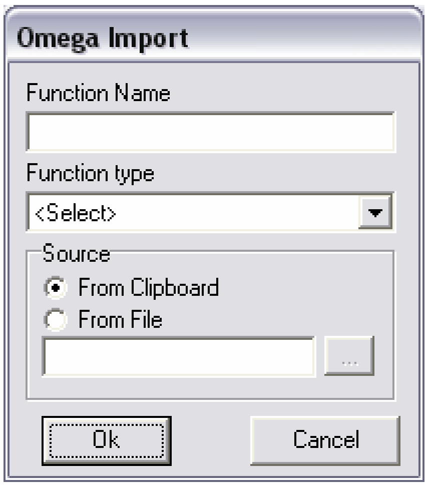


| Custom Functions List |
|---|
| AccDist |
| Acceleration |
| Add |
| ADX |
| ADXClassic |
| ADXR |

| Parameter | Value |
|---|---|
| Short Name | A |
| Full Name | Accumulation/Distribution |
| Description | Accumulation/Distribution uses the closing price's proximity ... |
| Status | valid |

3. In the *Omega Import* dialog box, enter the name of the required function and select the function type. The following function type options are available:

- Numeric,
- String,
- Boolean.



4. Choose the source from which the functions are to be imported by selecting one of the following options

- From Clipboard;
- From File.

***Note:*** *If you select the* ***From Clipboard*** *option, you will need to run TradeStation, copy the function, and then paste it in Improvian Editor. The function type and name have to be the same as in TradeStation.*

***Note****: If you select the* ***From File*** *option, you will need to locate the imported file on your PC.*

5. Click **OK**.

The **Improvian Editor** will be displayed for you to check, edit and approve the function syntax.  For details, refer to *Using Improvian Editor*.

6. In the **Improvian Editor,** click **OK**.

The function will be added to the **Custom Functions List**.

## Importing MetaStock© Formulas

Tradecision enables you to import MetaStock code into Improvian. It can be helpful if you want to use MetaStock code in your analysis or trading techniques in Tradecision.

### To insert a MetaStock© code:

For example, you want to import **Stochastic RSI, 21, 13 Smoothed** which you found in the Formulae Collection at http://forum.equis.com/ezformulae.php
The formula has the following code:

*Mov((RSI(21)-LLV(RSI(21),13))/(HHV(RSI(21),13)-*
*(LLV(RSI(21)+.00001,13))),8,E)*
**100*

To automatically translate the formula into Improvian, do the following:

1. In Tradecision, go to the **Tools** menu and select **Indicator Builder**.
2. Click **New** to create a new indicator (StochRSI).
3. Click **Edit**.

The Improvian Editor dialog box will be displayed.

4. Copy the MetaStock Stochastic RSI, 21, 13 Smoothed code to clipboard.
5. In the **Improvian Editor** dialog box, click **Edit** and select **Paste MetaStock**.

The MetaStock code will be automatically decoded into Improvian as follows:

*return Mov((RSI(C,21) - Lowest(RSI(C,21), 13)) / (Highest(RSI(C,21), 13) - (Lowest(RSI(C,21) + 0.00001, 13))), 8, E) * 100;*

6. Click **OK** to save the indicator.

***Note:*** *you cannot import all formulas from MetaStock automatically. There are several problems that prevent such easy importing. If you encounter a problem, you should review your code to find the cause of the problem.*

## Problems Relating to Translation from MetaStock© into Improvian

**1.** Those indicators that draw several lines on a one (sub) chart. Only the first line will be converted and plotted. For example,

```
Pds:= Input("EMA Periods?",1,1000,21);
Pct:= Input("Percentage Bands?",0.1,10,5);
MA:= Mov(C,Pds,E);
TBnd:= MA*(1+Pct/100);
LBnd:= MA*(1-Pct/100);
MA;TBnd;LBnd;
```

will be translated as an indicator that returns a single value:

```
input
     inp0:"EMA Periods?", 21, 1, 1000;inp1:"Percentage Bands?", 5, 0.1, 10;
end_input
var
Pds:=inp0;
Pct:=inp1;
MA:=Mov(C, Pds, E);
TBnd:=MA * (1 + Pct / 100);
LBnd:=MA * (1 - Pct / 100);

end_var
return MA;return TBnd;return LBnd;
```

As a result only a single MA value can be inserted into a chart.

**2.** Indicators using MetaStock's DLL, for example, a function call from the PowerPivot library.

For example, if you use the following piece of code
*TCL:=ExtFml("PowerPivots.TimeCapsules",L,3,1);*
The code will not be processed correctly and Improvian will report an error.

**3.** Tradecision does not support some of the MetaStock functions. The list is constantly changing.

**4.** The inability to access the variables and values of other indicators (for this purpose, Function Builder can be used). Automatic conversion for structures using fml and fmlvar functions is not supported.

**5**. Operations with different types are not allowed.
MetaStock allows implied conversion of value types. Tradecision doesn't support it which leads to its inability to convert code automatically. For example, if you want to use MetaStock's

*if(close-open,10,15)*

it will be as

*iff(close-open,10,15)*

in Improvian.

Improvian Editor will inform you about the error: *"Function iff with parameters (Numeric,Numeric,Numeric) not found…".* It means that in Improvian there is only an IFF function which has a Boolean expression as its first parameter. Manually, we can write it as follows:

*iff(close-open<>0,10,15)*

The above variant of the function has the correct expression syntax.

# Glossary of Terms

**Builders** – special program modules that allow creating and managing trading techniques in Tradecision. All these models have separate windows. It means that you cannot access a strategy or indicator, say, from Study Builder.

**Comments** – text blocks that can carry some explanatory information for a particular trading technique. Syntax: {explanatory information}.

**Expression –** an expression is any combination of reserved words and operators that represent a value.

**Formula** – is a logical arrangement of one or more functions, operators, data arrays, and so on.

For example: Average True Range strategy:

*return*
*CLOSE\0\ > (OPEN\0\ + (0.5 * AVGTRUERNG(14)));*

**Function Builder –** a special tool that allows creating and managing user functions.

**Function Parameter -** a value (enclosed in parentheses) which can be given to a function. Parameters provide the information required to calculate the function value. The parameters that can be assigned to a function are explained in the description of the function.

**Name (Identifier)** – a name of a function. It can be no longer than 1000 symbols including letters of Roman alphabet, numeric characters and underscore. It is incorrect to begin an identifier with a digit.

**Previous values** – Variable value of the previous step a\1\.

**Reference** – a reference to a variable or array used in a function declaration.

**Return statement** – A return instruction, stopping the execution of a program and returning a value.

**Syntax** - a set of rules defining how to compose sentences using the given language (Improvian) and its words. Syntax allows understanding how to write formulas based functions and operators.

**Semantics** - a branch of linguistics which studies meaning in language, including the relationship between the language, thought, and behavior. For example, if a computer understands the semantics of a document, it understands the document's meaning too, rather than just interpreting a series of characters.

**Statements -** an Improvian's statement represents a complete instruction. Statements can contain reserved words, operators, and punctuation marks, and always end into a semicolon.

```
if Length >0 then
Average:=Sum_ / Length;
else
Average:=0;
```

**Technique Validity –** A trading technique can be valid or invalid. A technique is valid if it can be executed correctly. A technique is invalid when it uses another technique that cannot be called for one of the following reasons:

- absence of the technique (it was permanently deleted);
- the parameters of the technique were changed.

**Trading Technique –** a trading strategy, alert, scan, study or indicator you want to create and/or use in Tradecision.

# Support

The Tradecision Support Team is dedicated to helping you succeed with Tradecision. Our support is easily accessible, thorough, flexible and free.

Feel free to contact us via:

**E-mail**
support@tradecision.com

**Internet**
www.tradecision.com/support/support.htm
or Live Support (online chat)

**Phone:**
(347) 416 6083 (7am - 3pm Eastern Time)
(888) 862 2759, ext. 3 (9am - 5pm Pacific Time)

**Fax**
(510) 279 5649

**Mail**
Neo Digital, Inc.
4432-E Enterprise Str
Fremont, CA, 94538, USA

Please include the following information with your technical support questions:

**A Clear, detailed description of your problem or the question**
Is the problem reproducible? If so, then how? If a dialog box with an error message was displayed, please include the full text of the dialog box, including the title in the title bar.

**Which version of Windows are you using?**
For example, Windows XP, Service Pack 2.

**Which version of Tradecision are you using?**
From the Tradecision menu, select **Help** and then click **About**. Please include the entire "version" line in your problem report.

# Disclaimer

# Thank you for using Tradecision!

We wish you the best of luck in your trading!

Should you have any questions, feel free to contact
The Tradecision Technical Support Team at
support@tradecision.com