MQL4 COURSE

By Coders' guru

-1-
WELCOME

Welcome to the MQL4 course.
In this series, I will try to strip the mystique and confusion from MQL4 by giving you comprehensive tutorials with a straight forward example.

In this series of lessons, I will show you how to use the MQL4 for building your own Expert Advisors, Custom Indicators and Scripts.

If you are programming in C (or its superset C++) then you know a lot of MQL4 before even I start my lessons, if you didn't write in any programming language before, no problem, I'll guide you to understand the concept of programming in general as well.

So, let's start from the beginning.

MQL4? What, Why and Where?

MQL4 stands for **MetaQuotes Language 4**.
MetaQuotes is the company who built the MetaTrader Trading Platform.
And to make it stronger than the other trading platforms the company extended it by a built-in programming language that enables the user (you) to write his own trading strategies.

The language enables you to create one of the following:

1- Expert Advisors.
2- Custom Indicators.
3- Scripts.

- Expert Advisor is a program which can automate trading deals for you. For example it can automate your market orders, stops orders automatically, cancels/replaces orders and takes your profit.

- Custom Indicator is a program which enables you to use the functions of the technical indicators and it cannot automate your deals.

- Script is a program designed for single function execution.
  Unlike the Advisor, scripts are being held only once (on demand), and not by ticks. And of course has no access to indicator functions.

These were "What" MQL4 is? "Why" to use MQL4?
Now, "Where" do I write MQL4?

To write your MQL4 code and as anything else in world, you can choose one of two ways, the hard way and the easy way.

1- The hard way
The hard way is using your favorite text editor and the command prompt to compile your program.
Notepad is not bad choice, but do not forget two things:
1- To save the file you have created in plain text format.
2- To save the file as .mp4 (that's to be easy to reopen it with Metaeditor), but you can save it as any extension you prefer.

After saving your program there is an extra step to make your code comes out to the light.
It's the Compiling step.
Compiling means to convert the human readable script that you have just wrote to the machine language that your computer understands.

MetaTrader has been shipped with its own compiler (the program which will convert your script to the machine language) called **MetaLang.exe.**

Metalang.exe is a console program which takes 2 parameters and output an .ex4 file (the file which Metatrader understands).
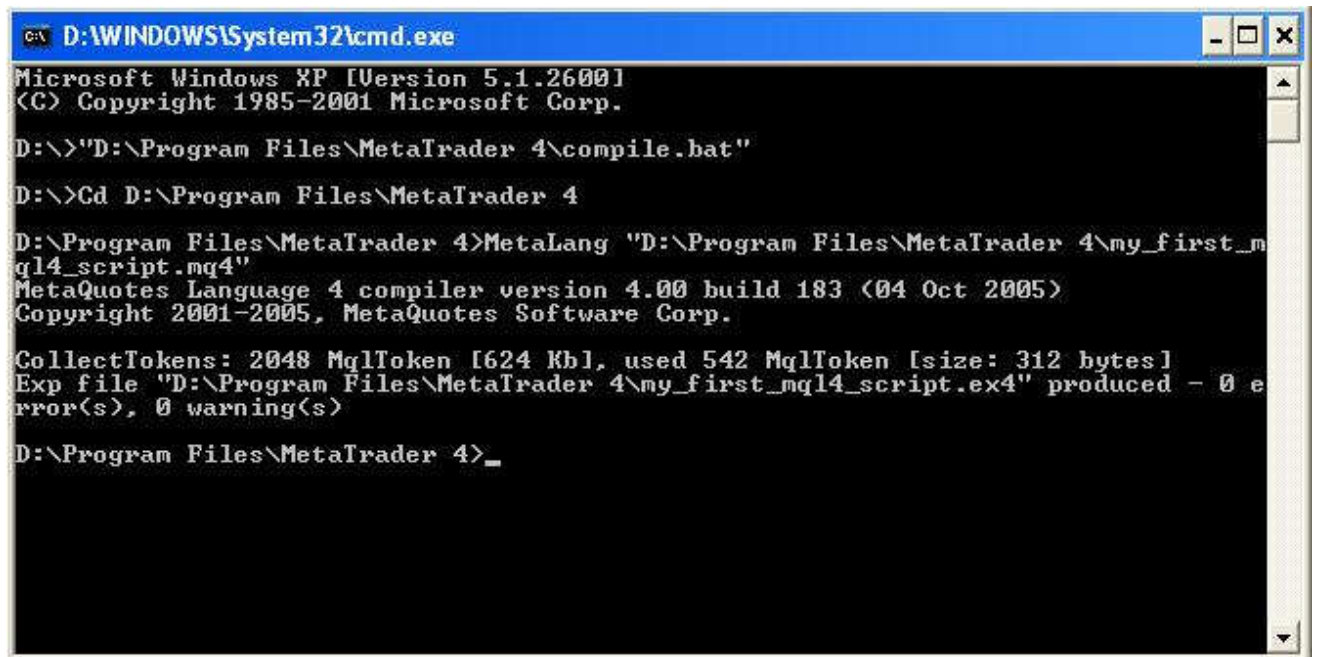The first parameter is "options" parameter and the only option available is **–q** quit
The second parameter is the full path to your .mql file.

The syntax will be in this format.
metalang [options…] filename

Example
1- Find your metalang.exe path, it will be the same path of MetaTrader (here my path is D:\Program Files\MetaTrader 4)
2- Create a batch file and name it compile.bat (or any name you prefer).
3- Write these lines into the bat file then save it.
   cd D:\Program Files\MetaTrader 4
   metalang -q "D:\Program Files\MetaTrader 4\my_first_mql4_script.mq4"
   (Don't forget to change the path to you MetaTrader installed path)
4- Run the batch file and if you are lucky person like me you will get a screen like this.

Figure 1 *Metalang compiler*

As you see you will get the output file "my_first_mql4_script.ex4"

2-The easy way
Metatrader has been shipped with a good IDE (integrated development editor) called
MetaEditor which has these features:

1- A text editor has the feature of highlighting different constructions of MQL4
language while you are writing/reading code.

2- Easy to compile your program, just click F5 and the MetaEditor will make all the
hard work for you and produces the "ex4" file.
Besides it's easy to see what the wrong in your program is (in the Error Tab – see
figure 2).

3- Built-in a dictionary book which you can access by highlight the keyword you
want to know further about it then press F1.
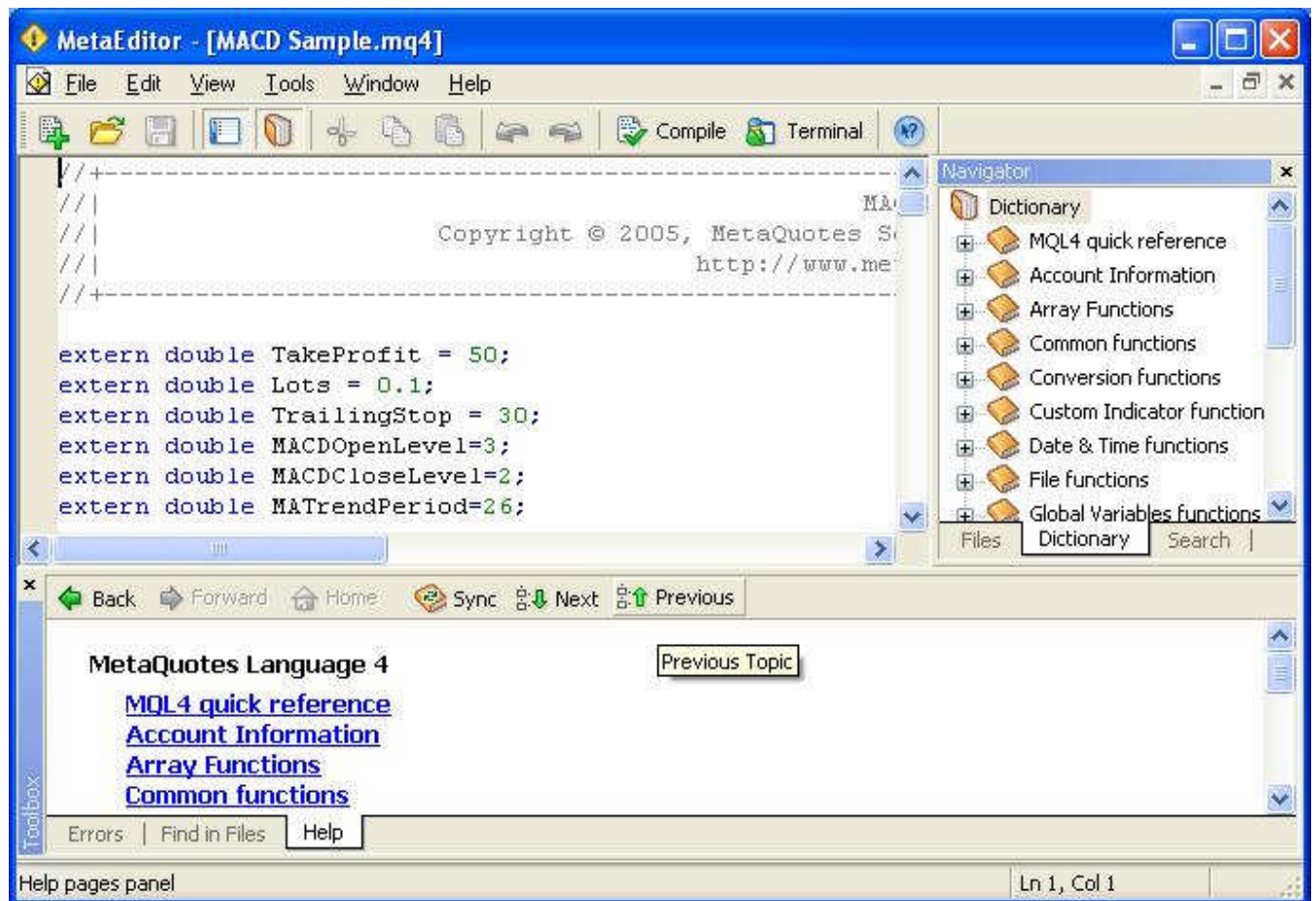
Figure 2 *MetaEditor 4*

In the coming lessons we will know more about MetaEditor.

Today I just came to say hello, tomorrow we will start the real works.
Tomorrow we will study the Syntax of MQL4?

I welcome very much the questions and the suggestions.

See you
Coders' Guru
19-10-2005

MQL4 COURSE
By Coders' guru

-Exercise 1-
WELCOME

Question 1

What are the kinds of programs you can write in MQL4?

1- Expert Advisors.
2- Custom Indicators.
3- Scripts.

……………………………………………………………………………………………

Question 2

Which sentences of the following are false, and why?

1- Expert Advisor is a program which enables you to use the functions of the technical indicators and it cannot automate your deals.

(False)

Why:
This is the definition of the Custom Indicator, while the Expert Advisor is:
Expert Advisor is a program which can automate trading deals for you. For example it can automate your market orders, stops orders automatically, cancels/replaces orders and takes your profit.

2- The source file of the code you write in MQL4 is saved as ".mqh" file, while the ".mq4" is the extension of the MQL4 header files which you can include into your code as need.

3- Metalang.exe is the compiler shipped with MQL4, its job is taking the source code you've written in MQL4 and produce a ".ex4" file which you can load it in MetaTrader.

………………………………………………………………………………………………

Question 3

You have decided not to use MetaEditor and you used Notepad to write your MQL4 code, tell me the steps needed to compile your program.
Assume:
 A- **You have installed MetaTrader in "C:\Program Files\MetaTrader 4" path.**
 B- **You save your source code file as "my_first_mql4_script.mq4" at the same path of MetaTrader.**

MQL4 COURSE

By Coders' guru
www.forex-tsd.com

-2-
SYNTAX

I hope you enjoyed the "Welcome" lesson which tried to answer the very basic questions; **what** MQL4 is, **why** MQL4 and **where** (to write) MQL4?

Always the biggest and the most important question(s) are **how**, and the entire coming lessons are the answer.

Now, I want you to empty your mind from any confusion and read carefully the next few concepts.

We are talking today about the SYNTAX rules of MQL4.
And as I told you before, *If you are programming in C (or its superset C++) then you know a lot of MQL4 before even I start my lessons*.
That's because the syntax of MQL4 is very like of the syntax of C.

The dictionary means of the word SYNTAX of a programming language is:
"The set of allowed reserved words and their parameters and the correct word order in the expression is called the syntax of language". *"**Wikipedia**"*

So, when we are studying the syntax of the language we are studying its grammar and writing rules which consist of:

- Format
- Comments
- Identifiers
- Reserved words

Let's slice the cake.

1- Format:

When you write your code, you can freely use any set of spaces, tabs and empty lines you want to separate your code and your line of code to make them readable and eyes pleasing.

For example all of these lines are valid in MQL4:

```
double  MacdCurrent, MacdPrevious, SignalCurrent;
```

```
double
MacdCurrent,
MacdPrevious,
SignalCurrent;
```

```
double                MacdCurrent,      MacdPrevious,       SignalCurrent;
```

But, as you see, the first line is more readable and easy to understand.

And as everything in the world there are exceptions to the rule:

1- You can't use new line in the "Controlling compilation"
 You will know more about "Controlling compilation" in next lesson but just remember this is an exception.

For example the next line of code is invalid and the MQL4 compiler will complain:

```
#property
copyright "Copyright © 2004, MetaQuotes Software Corp."
```

This is the valid "Controlling compilation":

```
#property copyright "Copyright © 2004, MetaQuotes Software Corp."
```

2- You can't use new line or space in the middle of Constant values, Identifiers or Keywords.

For example this line is valid:

```
extern int MA_Period=13;
```

"extren" and "int" here are Keywords , "MA_Period" is an Identifier and "13" is a Constant value..
You will know more in the next lessons.

For example the next lines are invalids:

```
extern int MA_Period=1
3;
```

```
extern int MA_Period=1        3;
```

Notice the tab between 1 and 3.

```
ex
tern int MA_Period=13;
```

2- Comments:

To make the programming world easier, any programming language has its style of writing comments.
You use Comments to write lines in your code which the compiler will ignore then but it clears your code and makes it understandable.
Assume that you write a program in the summer and in the winter you want to read it. Without comments -even you are the code's creator- you can't understand all these puzzled lines.

MQL4 (& C/C++) uses two kinds of comments styles:

1- Single line comments

The Single line comment starts with "//" and ends with the new line.
 For example:

```
//This is a comment
extern int MA_Period=13;
```

```
extern int MA_Period=13; //This is another comment
```

2- Multi-line comments

The multi-line comment start with "/*" and ends with "*/".
And you can comment more than line or more by putting "/*" at the start of the first line, and "*/" at the end of the last line.
For example:

```
/* this
is
multi
line
comment*/
```

You can also nest single line comment inside multi lines comment like that:

```
/* this
is
multi      //another comment nested here.
line
comment*/
```

This is a valid comment too:

```
extern int /*HELLO! I'm a comment*/ MA_Period=13;
```

But this is invalid comment:

```
extern int //test MA_Period=13;
```

3- Identifiers:

An identifier is the name you choose to your variables, constants and functions.

For example MA_Period here is an identifier:

```
extern int MA_Period=13;
```

There are few rules and restrictions for choosing those names:

1- The length of the Identifier must not exceed **31** characters.

2- The Identifier must begin with a letter (capital or small) or the underlining symbol _.
   So, it can't be started with a number or another symbol except the underlining symbol.

3- You can't use any reserved words as an Identifier.
   You will see the list of the reserved words too soon.

4- The identifiers' names are case sensitive.
   So, **MA_PERIOD** not the same as **ma_period** or **MA_Period**

   Let's take some examples:

Name1	Valid
_Name1	Valid
1Name	Invalid (don't start with number)
~Name1	Invalid (you can only use underline symbol)
N~ame1	Invalid (you can only use underline symbol)
i_love_my_country_and_my_country_loves_all_the_world	
	Invalid (you can't exceed the 31 characters length)
Color	Valid
color	Invalid (you can't use reversed word, and **color** is one of them)

4- Reserved words:

There are "words" which the language uses them for specific actions.

So, they are reserved to the language usage and you can't use them as an identifier name or for any other purpose.

This is the list of the reserved words (*from the MQL4 guide*):

Data types	Memory classes	Operators	Other
bool	extern	break	false
color	static	case	true
datetime		continue	
double		default	

int		else	
string		for	
void		if	
		return	
		switch	
		while	

For example the next lines of code are invalid:

```
extern int datetime =13;
int extern =20;
double continue = 0;
```

I hope you enjoyed the lesson.
The next lesson will be about the "**Data Types**".
So, Be Ready, the real hard work is coming!

I welcome very much the questions and the suggestions.

See you
Coders' Guru
20-10-2005

MQL4 COURSE

By Coders' guru
www.forex-tsd.com

-Exercise 2-
SYNTAX

Question 1

SYNTAX in MQL4 is the studying of its grammar and writing rules which consist of

1- Format.
2- Comments
3- Identifiers
4- Reserved words

………………………………………………………………………………………

Question 2

Which sentences of the following are false, and why?

1- When you write your code, you can freely use any set of spaces, tabs and empty lines you want to separate your code and your line of code to make them readable and eyes pleasing.

(True)

Why:

2- You can use "//" symbol to comment Multi lines by starts the line with "//" and ends it with "//".

(False)

3- These keywords are reserved words in MQL4 which you can use them as name for your identifiers:

bool,white,Color,case,break,do and Extern

(False)

Why: Not all the keywords here are reserved words.
bool,case and break are reserved words, while:
white and do are not reserved words.
Color is not reserved word, but "color" is the reserved one.
Extern is not reserved word, but "extern" is the reserved one.

....

Question 3

Which of the following is a valid code and which is invalid, and why?

1-
extern int /*HELLO! I'm a comment*/ MA_Period=13;

(Valid)

Why:

2-
int 3Periods;

(Invalid)

Why: You can start your identifier name with a number.

3-
string _MyString = "Welcome";

(Valid)

Why:

**4-
double
MacdCurrent,
MacdPrevious,
SignalCurrent;**

(Valid)

Why:

5- color Color= C'128,128,128' ;

(Valid)

Why:

MQL4 COURSE

By Coders' guru
www.forex-tsd.com

-3-
DATA TYPES

Welcome to my third lesson in my MQL4 course.

I hope you enjoyed the "SYNTAX" lesson, which tried to give you the answers for:
- What format you can use to write MQL4 code?
- How to make the world better by commenting your code?
- What the Identifiers are, and what are the rules of choosing them?
- What are the MQL4 Reserved words?

If you didn't read the "SYNTAX" lesson please download it from here:
http://forex-tsd.com/attachment.php?attachmentid=399
And you can download the "Welcome" lesson from here:
http://forex-tsd.com/attachment.php?attachmentid=372
Don't forget to login first.

Now, let's **enjoy** the DATA TYPES.

What's the Data type mean?

Any programming language has a set of names of the memory representation of the data. For example if the memory holds numbers between -2147483648 to 2147483647, the most of the programming languages will name this data as "**Integer**" data type.

Variables?

Variables are the names that refer to sections of memory into which data can be stored.

To help you think of this as a picture, imagine that memory is a series of different size boxes. The box size is memory storage area required in bytes.

- In order to use a box to store data, the box must be given a name; this process is known as **declaration.**
- In the declaration process you use a word tell the computer what's the kind and size of the box you want to use, this word known as **keyword**.

- It helps if you give a box a meaningful name that relates to the type of information which make it easier to find the data, this name is the **variable constant**.
- Data is placed into a box by **assigning** the data to the box.
- When we set the value of the box you have created in the same line you declared the variable; this process is known as **initialization.**

When we create a variable we are telling the computer that we want him to assign a specified memory length (in bytes) to our variable, since storing a simple number, a letter or a large number is not going to occupy the same space in memory, so the computer will ask us what's the kind of data and how much the length of the data? That is the Data type for.

For example if we said this line of code to the computer:

```
int  MyVaraible=0;
```

That's mean we are asking the computer to set a block of 4 bytes length to our variable named "MyVaraiable".

In the previous example we have used:
int ← Keyword
int ← Integer data type.
int ← Declaration
MyVaraible ← Variable's constant.
=0 ← Initialization

We will know more about variables in a coming lesson.

In MQL4, these are the kinds of Data types:

- Integer (int)
- Boolean (bool)
- Character (char)
- String (string)
- Floating-point number (double)
- Color (color)
- Datetime (datetime)

1- Integer

An integer, is a number that can start with a + or a - sign and is made of digits. And its range value is between -2147483648 to 2147483647.
MQL4 presents the integer in decimal or hexadecimal format.

For example the next numbers are Integers:

```
12, 3, 2134, 0, -230
0x0A, 0x12, 0X12, 0x2f, 0xA3, 0Xa3, 0X7C7
```

We use the keyword **int** to create an integer variable.

For example:

```
int intInteger = 0;
int intAnotherIntger = -100;
int intHexIntger=0x12;
```

Decimal and Hexadecimal:

Decimal notation is the writing of numbers in the base of 10, and uses digits (0, 1, 2, 3, 4, 5, 6, 7, 8 and 9) to represent numbers. These digits are frequently used with a decimal point which indicates the start of a fractional part, and with one of the sign symbols + (plus) or − (minus) to indicate sign.

Hexadecimal is a numeral system with a base of 16 usually written using the symbols 0–9 and A–F or a–f.
For example, the decimal numeral 79 can be written as 4F in hexadecimal.

2- Boolean

Boolean variable is a data type which can hold only two values, true and false (or their numeric representation, 0 and 1). And it occupies 1 bit of the memory.
In MQL4, false,FALSE,False,true,TRUE and True are equals.

Boolean named like this in the honor of the great mathematician Boole George.

We use the keyword **bool** to create a boolean variable.

For example:

```
bool I = true;
bool bFlag = 1;
bool bBool=FALSE;
```

3- Character

MQL4 names this Data type "Literal".
A character is one of 256 defined alphabetic, numeric, and special key elements defined in the ASCII (American Standard Code for Information Interchange) set.
Characters have integer values corresponding to location in the ASCII set.
You write the character constant by using single quotes (') surrounding the character.

For example:

```
'a' , '$' , 'Z'
```

We use the keyword **int** to create a character variable.

For example:

```
int chrA = 'A';
int chrB = '$';
```

Some characters called Special Characters can't present directly inside the single quotes because they have a reserved meanings in MQL4 language.
Here we use something called **Escape Sequence** to present those special characters,
And that by prefixing the character with the backslash character (\).

For example:

```
int chrA = '\\';     //slash character
int chrB = '\n';     //new line
```

This is the list of Escape Sequence characters used in MQL4.

carriage return	\r
new line	\n
horizontal tab	\t
reverse slash	\\
single quote	\'
double quote	\"
hexadecimal ASCII-code	\xhh

ASCII table

```
Char  Dec  Oct   Hex | Char  Dec  Oct   Hex | Char  Dec  Oct   Hex | Char Dec  Oct   Hex
-------------------------------------------------------------------------------------------
(nul)   0  0000  0x00 | (sp)  32  0040  0x20 | @     64  0100  0x40 | `     96  0140  0x60
(soh)   1  0001  0x01 | !     33  0041  0x21 | A     65  0101  0x41 | a     97  0141  0x61
(stx)   2  0002  0x02 | "     34  0042  0x22 | B     66  0102  0x42 | b     98  0142  0x62
(etx)   3  0003  0x03 | #     35  0043  0x23 | C     67  0103  0x43 | c     99  0143  0x63
(eot)   4  0004  0x04 | $     36  0044  0x24 | D     68  0104  0x44 | d    100  0144  0x64
(enq)   5  0005  0x05 | %     37  0045  0x25 | E     69  0105  0x45 | e    101  0145  0x65
(ack)   6  0006  0x06 | &     38  0046  0x26 | F     70  0106  0x46 | f    102  0146  0x66
(bel)   7  0007  0x07 | '     39  0047  0x27 | G     71  0107  0x47 | g    103  0147  0x67
(bs)    8  0010  0x08 | (     40  0050  0x28 | H     72  0110  0x48 | h    104  0150  0x68
(ht)    9  0011  0x09 | )     41  0051  0x29 | I     73  0111  0x49 | i    105  0151  0x69
(nl)   10  0012  0x0a | *     42  0052  0x2a | J     74  0112  0x4a | j    106  0152  0x6a
(vt)   11  0013  0x0b | +     43  0053  0x2b | K     75  0113  0x4b | k    107  0153  0x6b
(np)   12  0014  0x0c | ,     44  0054  0x2c | L     76  0114  0x4c | l    108  0154  0x6c
(cr)   13  0015  0x0d | -     45  0055  0x2d | M     77  0115  0x4d | m    109  0155  0x6d
(so)   14  0016  0x0e | .     46  0056  0x2e | N     78  0116  0x4e | n    110  0156  0x6e
(si)   15  0017  0x0f | /     47  0057  0x2f | O     79  0117  0x4f | o    111  0157  0x6f
(dle)  16  0020  0x10 | 0     48  0060  0x30 | P     80  0120  0x50 | p    112  0160  0x70
(dc1)  17  0021  0x11 | 1     49  0061  0x31 | Q     81  0121  0x51 | q    113  0161  0x71
(dc2)  18  0022  0x12 | 2     50  0062  0x32 | R     82  0122  0x52 | r    114  0162  0x72
(dc3)  19  0023  0x13 | 3     51  0063  0x33 | S     83  0123  0x53 | s    115  0163  0x73
(dc4)  20  0024  0x14 | 4     52  0064  0x34 | T     84  0124  0x54 | t    116  0164  0x74
(nak)  21  0025  0x15 | 5     53  0065  0x35 | U     85  0125  0x55 | u    117  0165  0x75
(syn)  22  0026  0x16 | 6     54  0066  0x36 | V     86  0126  0x56 | v    118  0166  0x76
(etb)  23  0027  0x17 | 7     55  0067  0x37 | W     87  0127  0x57 | w    119  0167  0x77
(can)  24  0030  0x18 | 8     56  0070  0x38 | X     88  0130  0x58 | x    120  0170  0x78
(em)   25  0031  0x19 | 9     57  0071  0x39 | Y     89  0131  0x59 | y    121  0171  0x79
(sub)  26  0032  0x1a | :     58  0072  0x3a | Z     90  0132  0x5a | z    122  0172  0x7a
(esc)  27  0033  0x1b | ;     59  0073  0x3b | [     91  0133  0x5b | {    123  0173  0x7b
(fs)   28  0034  0x1c | <     60  0074  0x3c | \     92  0134  0x5c | |    124  0174  0x7c
(gs)   29  0035  0x1d | =     61  0075  0x3d | ]     93  0135  0x5d | }    125  0175  0x7d
(rs)   30  0036  0x1e | >     62  0076  0x3e | ^     94  0136  0x5e | ~    126  0176  0x7e
(us)   31  0037  0x1f | ?     63  0077  0x3f | _     95  0137  0x5f | (del) 127 0177  0x7f
```

4- String

The string data type is an array of characters enclosed in double quote (").

The array of characters is an array which holds one character after another, starting at index 0. After the last character of data, a NULL character is placed in the next array location. It does not matter if there are unused array locations after that.

A NULL character is a special character (represented by the ASCII code 0) used to mark the end of this type of string.

See figure 1 for a simple representation of the string constant "hello" in the characters array.

h	e	l	l	o	NULL				
0	1	2	3	4	5	6	7	8	9

Figure 1 – Characters array

MQL4 limits the size of the string variable to 255 characters and any character above 255 characters will generate this error: (*too long string (255 characters maximum)*).

You can use any special character -mentioned above- in your string constant by prefixing it with the backslash (\).

We use the keyword **string** to create a string variable.

For example:

```
string str1 = "Hello world1, with you coders guru";
string str2 = "Copyright © 2005, \"Forex-tsd forum\".";  //Notice the use of (") character.
string str3 = "1234567890";
```

5- Floating-point number (double)

Floating point number is the Real Number (that is, a number that can contain a fractional part beside the integer part separated with (.) dot).Ex: 3.0,-115.5, 15 and 0.0001.

And its range value is between 2.2e-308 to 1.8e308.

We use the keyword **double** to create a floating-point variable.

For example:

```
double dblNumber1 = 1000000000000000;
double dblNumber3 = 1/4;
double dblNumber3 = 5.75;
```

6- Color

Color data type is a special MQL4 data type, which holds a color appears on the MetaTrader chart when you create your own Expert Advisor or Custom Indictor and the user can change it from the property tab of your Expert Advisor or Custom Indictor.

You can set the Color variable constant in three ways:

1- **By the color name:** For the well know colors (called Web Colors Set) you can assign the name of the color to the color variable, see the list of the Web Colors Set.

2- **By Character representation** (MQL4 named it this name): In this method you use the keyword (C) followed by two signal quotations ('). Between the two signal quotations you set the value of the red, green and blue (know as RGB value of the color). These values have to be between: 0 to 255. And you can write these values in decimal or hexadecimal format.

3- **By the integer value:** Every color in the Web Colors Set has its integer value which you can write it in decimal or hexadecimal format. And you can assign the Integer value of the color to the color variable. The hexadecimal color format looks like this: 0xBBGGRR where BB is the blue value, GG is green value and RR is the red value.

For example:

```
// symbol constants
C'128,128,128'   // gray
C'0x00,0x00,0xFF' // blue
// named color
Red
Yellow
Black
// integer-valued representation
0xFFFFFF        // white
16777215        // white
0x008000        // green
32768        // green
```

We use the keyword **color** to create a color variable.

For example:

```
color clr1= Red;
color clr1= C'128,128,128' ;
```

```
color clr1=32768;
```

Web Colors Set

Black	DarkGreen	DarkSlateGray	Olive	Green	Teal	Navy	Purple
Maroon	Indigo	MidnightBlue	DarkBlue	DarkOliveGreen	SaddleBrown	ForestGreen	OliveDrab
SeaGreen	DarkGoldenrod	DarkSlateBlue	Sienna	MediumBlue	Brown	DarkTurquoise	DimGray
LightSeaGreen	DarkViolet	FireBrick	MediumVioletRed	MediumSeaGreen	Chocolate	Crimson	SteelBlue
Goldenrod	MediumSpringGreen	LawnGreen	CadetBlue	DarkOrchid	YellowGreen	LimeGreen	OrangeRed
DarkOrange	Orange	Gold	Yellow	Chartreuse	Lime	SpringGreen	Aqua
DeepSkyBlue	Blue	Magenta	Red	Gray	SlateGray	Peru	BlueViolet
LightSlateGray	DeepPink	MediumTurquoise	DodgerBlue	Turquoise	RoyalBlue	SlateBlue	DarkKhaki
IndianRed	MediumOrchid	GreenYellow	MediumAquamarine	DarkSeaGreen	Tomato	RosyBrown	Orchid
MediumPurple	PaleVioletRed	Coral	CornflowerBlue	DarkGray	SandyBrown	MediumSlateBlue	Tan
DarkSalmon	BurlyWood	HotPink	Salmon	Violet	LightCoral	SkyBlue	LightSalmon
Plum	Khaki	LightGreen	Aquamarine	Silver	LightSkyBlue	LightSteelBlue	LightBlue
PaleGreen	Thistle	PowderBlue	PaleGoldenrod	PaleTurquoise	LightGrey	Wheat	NavajoWhite
Moccasin	LightPink	Gainsboro	PeachPuff	Pink	Bisque	LightGoldenRod	BlanchedAlmond
LemonChiffon	Beige	AntiqueWhite	PapayaWhip	Cornsilk	LightYellow	LightCyan	Linen
Lavender	MistyRose	OldLace	WhiteSmoke	Seashell	Ivory	Honeydew	AliceBlue
LavenderBlush	MintCream	Snow	White				

7- Datetime

Datetime data type is a special MQL4 data type, which holds a date and time data. You set the Datetime variable by using the keyword (D) followed by two signal quotations ('). Between the two signal quotations you write a character line consisting of 6 parts for value of year, month, date, hour, minutes, and seconds. Datetime constant can vary from Jan 1, 1970 to Dec 31, 2037.

For example:

```
D'2004.01.01 00:00'    // New Year
D'1980.07.19 12:30:27'
D'19.07.1980 12:30:27'
D'19.07.1980 12'       //equal to D'1980.07.19 12:00:00'
D'01.01.2004'          //equal to D'01.01.2004 00:00:00'
```

We use the keyword **datetime** to create a datetime variable.

For example:

```
datetime dtMyBirthDay= D'1972.10.19 12:00:00';
datetime dt1= D'2005.10.22 04:30:00';
```

I hope you enjoyed the lesson.

The next lesson will be about the "Operations & Expressions".

I welcome very much the questions and the suggestions.

See you
Coders' Guru
22-10-2005

MQL4 COURSE

By Coders' guru
www.forex-tsd.com

-4-

Operations & Expressions

Welcome to the fourth lesson in my course about MQL4.
The previous lesson "Data Types" presented a lot of new concepts; I hope you understand it, and above all you **enjoyed** it.

You can download the previous lesson from here:
http://forex-tsd.com/attachment.php?attachmentid=399
http://forex-tsd.com/attachment.php?attachmentid=372
http://forex-tsd.com/attachment.php?attachmentid=469
Don't forget to login first.

Now, let's enjoy the Operations & Expressions.

What's the meaning of Operations & Expressions?

You know the operations very well. If I told you that (+,-,*, /) are the basic arithmetical operators, you will remember very fast what's the operator means.

I hear you saying "OK, I know the operations; could you tell me what's the meaning of the expression?"
Identifiers (do you remember them? If not, Review the SYNTAX lesson) together with the **Operations** produce the **Expressions**.
Puzzled? Let's illustrate it in an example:

```
x = (y*z)/w;
```

x,**y**,**z** and **w**, here are **identifiers**.
=,* and / are the **operators**.
The whole line is an **expression**.

*When the expressions combined together it makes a **statement**.*
*And when the statements combined together it makes a **function** and when the functions combined together it makes a **program**.*

In the remaining of this lesson we are going to talk about the kinds operators used in MQL4.

So, let's start with the basic arithmetical operators:

1- Arithmetical operators:

In MQL4 there are 9 Arithmetical operations
This is the list of them with the usage of each:

Operator	Name	Example	Description
+	Addition operator	A = B + C;	Add **B** to **C** and assign the result to **A**.
-	Subtraction operator	A = B - C;	Subtract **C** from **B** and assign the result to **A**.
+ -	Sign changer operators	A = -A;	Change the sign of **A** from positive to negative.
*	Multiplication operator	A = B * C;	Multiply **B** and **C** and assign the result to **A**.
/	Division operator	A = B / C;	Divide **B** on **C** and assign the result to **A**.
%	Modulus operator	A =A % C;	**A** is the reminder of division of **B** on **C**. (ex: 10%2 will produce 0, 10%3 will produce 1).
++	Increment operator	A++;	Increase **A** by **1** (ex: if A =1 make it 2).
--	Decrement operator	A--;	Decrease **1** from A (ex: if A =2 make it 1).

Note: The remainder operator works by dividing the first number by the second number for the first integer results and then returns the remaining number.

For example:
10%5=0
This is because if you divide 10 by 5 you will get 2 and there no remaining value, so the remainder is 0.
10%8=2
This is because if you divide 10 by 8 you will get 1 (1*8=8), so the remainder is (10-8 = 2).
100%15=10

This is because if you divide 100 by 15 you will get 6 (6*15=90), so the remainder is (100-90=10).

What about 6%8?
It will be 6 because if you divide 6 by 8 you will get 0 (8*0=0), so the remainder is (6-0=6).

Note: You can't combine the increment and decrement operator with other expressions. For example you can't say:

```
A=(B++)*5;
```

But you can write it like that:

```
A++;
B=A*5;
```

Note: How the above example works? Let's assume:
int A=1; //set A to 1
int B;
A++; //increase A by 1, now A=2
B=A*5; //which means B=2*5

2- Assignment operators:

The purpose of any expression is producing a result and the assignment operators setting the left operand with this result.

For example:

```
A = B * C;
```

Here we multiply **B** and **C** and assign the result to **A**.
(=) here is the assignment operator.

In MQL4 there are 11 assignments operations
This is the list of them with the usage of each:

Operator	Name	Example	Description
=	Assignment operator	**A = B;**	Assign **B** to **A.**
+=	Additive Assignment operator	**A += B;**	It's equal to: **A = A + B;** Add **B** to **A** and assign the result to **A**.

-=	Subtractive Assignment operators	A -= B;	It's equal to: **A = A - B;** Subtract B from A and assign the result to A.
*=	Multiplicative Assignment operator	A *= B;	It's equal to: **A = A * B;** Multiply A and B and assign the result to A.
/=	Divisional Assignment operator	A /= B;	It's equal to: **A = A / B;** Divide A on B and assign the result to A.
%=	Modulating Assignment operator	A %= B;	It's equal to: **A = A % B;** Get the reminder of division of **A** on **B** and assign the result to **A**.
>>=	Left Shift Assignment operator	A >>= B;	It shifts the bits of **A** left by the number of bits specified in **B**.
<<=	Right Shift Assignment operator	A <<= B;	It shifts the bits of **A** right by the number of bits specified in **B**.
&=	AND Assignment operator	A &= B;	Looks at the binary representation of the values of **A** and **B** and does a bitwise AND operation on them.
|=	OR Assignment operator	A |= B;	Looks at the binary representation of the values of **A** and **B** and does a bitwise OR operation on them.
^=	XOR Assignment operator	A ^= B;	Looks at the binary representation of the values of two **A** and **B** and does a bitwise exclusive OR (XOR) operation on them.

3- Relational operators:

The relational operators compare two values (operands) and result false or true only.
It's is like the question "Is **John** taller than **Alfred**? Yes / no?"
The result will be false only if the expression produce zero and true if it produces any number differing from zero;

For example:

```
4 == 4;        //true
4 < 4;         //false
4 <= 4         //true;
```

In MQL4 there are 6 Relational operations
This is the list of them with the usage of each:

Operator	Name	Example	Description
==	Equal operator	A == B;	True if **A** equals **B** else False.
!=	Not Equal operator	A != B;	True if **A** does not equal **B** else False.
<	Less Than operators	A < B;	True if **A** is less than **B** else False.
>	Greater Than operator	A > B;	True if **A** is greater than **B** else False.
<=	Less Than or Equal operator	A <= B;	True if **A** is less than or equals **B** else False.
>=	Greater Than or Equal operator	A >= B;	True if **A** is greater than or equals **B** else False.

4- Logical operators:

Logical operators are generally derived from Boolean algebra, which is a mathematical way of manipulating the truth values of concepts in an abstract way without bothering about what the concepts actually *mean*. The truth value of a concept in Boolean value can have just one of two possible values: true or false.
MQL4 names the Logical operators as Boolean operators

MQL4 uses the most important 3 logical operators.
This is the list of them with the usage of each:

Operator	Name	Example	Description
&&	AND operator	A && B;	If either of the values are zero the value of the expression is zero, otherwise the value of the expression is 1. If the left hand value is zero, then the right hand value is not considered.
||	OR operator	A || B;	If both of the values are zero then the value of the expression is 0 otherwise the value of the expression is 1. If the left hand value is non-zero, then the right hand value is not considered.
!	NOT operator	!A;	Not operator is applied to a non-

			zero value then the value is zero, if it is applied to a zero value, the value is 1.

5- Bitwise operators:

The bitwise operators are similar to the logical operators, except that they work on a smaller scale -- binary representations of data.

The following operators are available in MQL4:

Operator	Name	Example	Description
&	AND operator	**A & B;**	Compares two bits and generates a result of 1 if both bits are 1; otherwise, it returns 0.
|	OR operator	**A | B;**	Compares two bits and generates a result of 1 if the bits are complementary; otherwise, it returns 0.
^	EXCLUSIVE-OR operator	**A ^ B;**	Compares two bits and generates a result of 1 if either or both bits are 1; otherwise, it returns 0.
~	COMPLEMENT operator	**~A;**	Used to invert all of the bits of the operand.
>>	The SHIFT RIGHT operator	**A >> B;**	Moves the bits to the right, discards the far right bit, and assigns the leftmost bit a value of 0. Each move to the right effectively divides op1 in half.
<<	The SHIFT LEFT operator	**A << B;**	Moves the bits to the left, discards the far left bit, and assigns the rightmost bit a value of 0. Each move to the left effectively multiplies op1 by 2.

Note Both operands associated with the bitwise operator must be integers.

6- Other operators:

There are some operators which used in MQL4 and don't belong to one of the previous categories:

1- The array indexing operator (**[]**).
2- The function call operator (**()**);
3- The function arguments separator operator -comma (**,**)

We will know more about the Arrays and Functions in the next lessons, so just remember these 3 operators as "Other operators".

Operators Precedence:

If you don't explicitly indicate the order in which you want the operations in a compound expression to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators with a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Thus, the two following statements are equivalent:

```
x + y / 100
x + (y / 100) //unambiguous, recommended
```

When writing compound expressions, you should be explicit and indicate with parentheses **()** which operators should be evaluated first. This practice will make your code easier to read and to maintain.

The following table shows the precedence assigned to the operators in the MQL4. The operators in this table are listed in precedence order: The higher in the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with a relatively lower precedence. Operators on the same **group** have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right. Assignment operators are evaluated right to left.

()	Function call	*From left to right*
[]	Array element selection	

!	Negation	*From left to right*

~	Bitwise negation
-	Sign changing operation

*	Multiplication	*From left to right*
/	Division	
%	Module division	

+	Addition	*From left to right*
-	Subtraction	

<<	Left shift	*From left to right*
>>	Right shift	

<	Less than	*From left to right*
<=	Less than or equals	
>	Greater than	
>=	Greater than or equals	

==	Equals	*From left to right*
!=	Not equal	

&	Bitwise AND operation	*From left to right*

^	Bitwise exclusive OR	*From left to right*

&&	Logical AND	*From left to right*

||	Logical OR	*From left to right*

=	Assignment	*From right to left*
+=	Assignment addition	
-=	Assignment subtraction	
*=	Assignment multiplication	
/=	Assignment division	
%=	Assignment module	

>>=	Assignment right shift
<<=	Assignment left shift
&=	Assignment bitwise AND
|=	Assignment bitwise OR
^=	Assignment exclusive OR

,	Comma	*From left to right*

I hope you enjoyed the lesson.
I welcome very much the questions and the suggestions.

See you
Coders' Guru
23-10-2005

MQL4 COURSE

By Coders' guru
www.forex-tsd.com

-5-
Loops & Decisions
Part 1

Welcome to the fifth lesson in my course about MQL4.

You can download the previous lesson from here:
http://forex-tsd.com/attachment.php?attachmentid=399
http://forex-tsd.com/attachment.php?attachmentid=372
http://forex-tsd.com/attachment.php?attachmentid=469
http://forex-tsd.com/attachment.php?attachmentid=481
Don't forget to login first.

The normal flow control of the program you write in MQL4 (And in others languages as well) executes from top to bottom, A statement by a statement.

A statement is a line of code telling the computer to do something.
For example:
Print("Hello World");
return 0;
A semicolon at end of the statement is a crucial part of the syntax but usually easy to forget, and that's make it the source of 90% of errors.

But the top bottom execution is not the only case and it has two exceptions,
They are the loops and the decisions.

The programs you write like -the human- decides what to do in response of circumstances changing. In these cases the flow of control jumps from one part of the program to another. Statements cause such jumps is called Control Statements.
Such controls consist of Loops and Decisions.

LOOPS

Loops causing a section of your program to be repeated a certain number of times.
And this repetition continues while some condition is true and ends when it becomes false.
When the loop end it passes the control to next statement follow the loop section.

In MQL4 there are two kinds of loops:

The for Loop

The **for** loop considered the easiest loop because all of its control elements are gathered in one place.
The **for** loop executes a section of code a fixed number of times.

For example:

```
int j;
for(j=0; j<15; j++)
        Print(j);
```

How does this work?

The **for** statement consists of **for** keyword, followed by parentheses that contain three expressions separated by semicolons:

for(j=0; j<15; j++)

These three expressions are the **initialization** expression, the **test** expression and the **increment** expression:

j=0 ← initialization expression
j<15 ← test expression
J++ ← increment expression

The **body** of the loop is the code to be executed the fixed number of the loop:

Print(j);

This executes the body of the loop in our example for 15 times.

Note: the **for** statement in not followed by a semicolon. That's because the **for** statement

and the loop body are together considered to be a program statement.

The initialization expression:

The initialization expression is executed only once, when the loop first starts. And its purpose to give the loop variable an initial value (0 in our example).

You can declare the loop variable outside (before) the loop like our example:

int j;

Or you can make the declaration inside the loop parentheses like this:

for(int j=0; j<15; j++)

The previous two lines of code are equal, except the **Scope** of each variable (you will know more about the variable declaration and scopes in the Variables lesson).
The outside declaration method makes every line in the code block to know about the variable, while the inside declaration makes only the **for** loop to know about the variable.

You can use more that one initialization expression in **for** loop by separating them with comma (,) like this:

```
int i;
int j;
for(i=0 ,j=0;i<15;i++)
   Print(i);
```

The Test expression:

The test expression always a relational expression that uses relational operators (please refer to relational operators in the previous lesson).

It evaluated by the loop every time the loop executed to determine if the loop will continue or will stop. It will continue if the result of the expression is true and will stop if it false.

In our example the **body** loop will continue printing **i** (**Print(i)**) while the case **j<15** is true.
For example the j = 0,1,2,3,4,5,6,7,8,9,10,11,12,13 and 14.
And when **j** reaches **15** the loop will stops and the control passes to the statement following the loop.

The Increment expression:

The increment expression changes the value of the loop variable (**j** in our example) by increase it value by 1.
It executed as the last step in the loop steps, after initializing the loop variable, testing the test expression and executing the body of the loop.

Figure 1 shows a flow chart of the **for** loop.
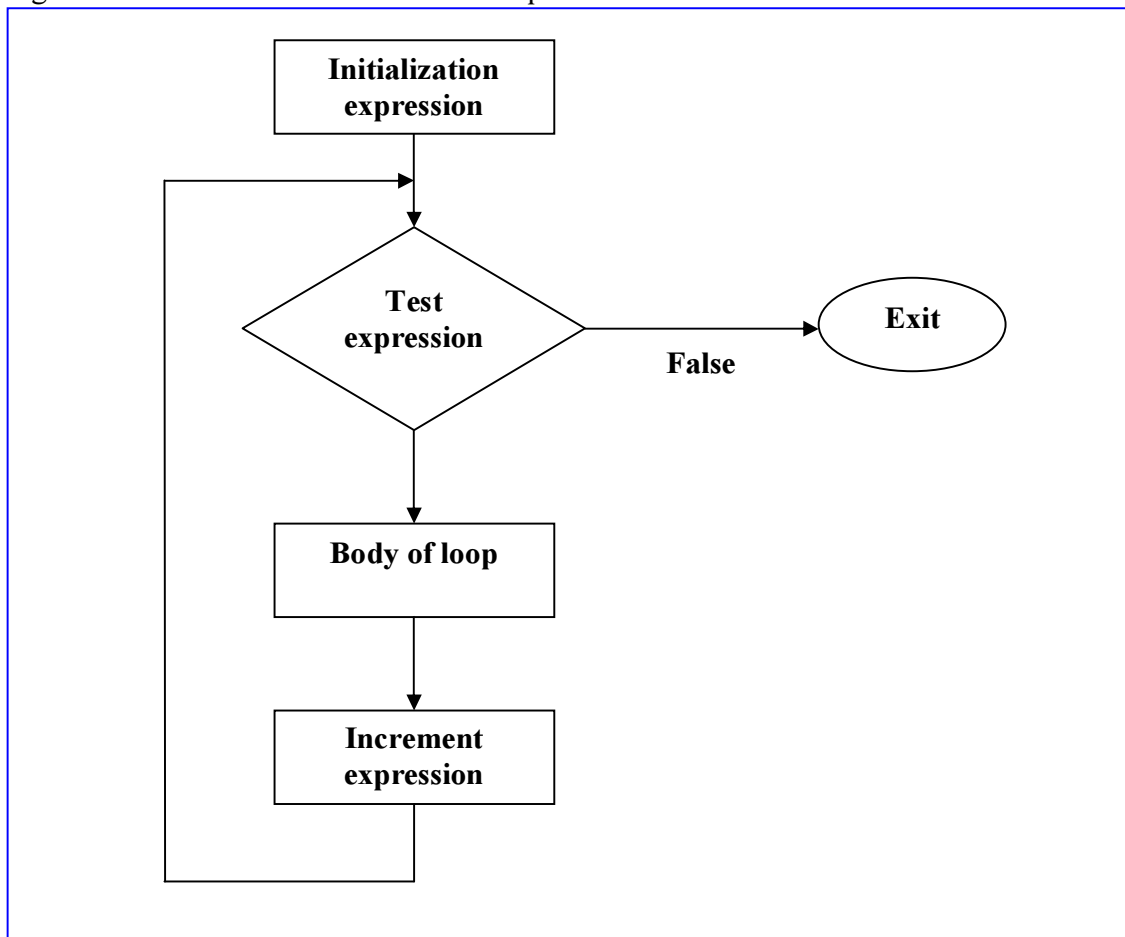
Figure 1 - Flow chart of the for loop

Like the initialization expression, in the increment expression you can use more than one increment expression in the **for** loop by separating them with comma (,) like this:

```
int i;
int j;
for(i=0 ,j=0;i<15,i<;i++,j++)
   Print(i);
```

But you can only use one test expression.

Another notice about the increment expression, it's not only can increase the variable of the loop, but it can perform and operation it like for example decrements the loop variable like this:

```
int i;
for(i=15;i>0,i<;i--)
   Print(i);
```

The above example will initialize the **i** to **15** and start the loop, every time it decreases **i** by **1** and check the test expression (**i>0**).
The program will produce these results: 15,14,13,12,11,10,9,8,7,6,5,4,3,2,1.

Multi statement in the loop body:

In our previous examples, we used only one statement in the body of the loop, this is not always the case.
You can use multi statements in the loop body delimited by braces like this:

```
for(int i=1;i<=15;i++)
{
   Print(i);
   PlaySound("alert.wav");
}
```

In the above code the body of the loop contains two statements, the program will execute the first statement then the second one every time the loop executed.
Don't forget to put a semicolon at the end of every statement.

The Break Statement:

When the keyword presents in the **for** loop (and in **while** loop and **switch** statement as well) the execution of the loop will terminate and the control passes to the statement followed the loop section.

For example:

```
for(int i=0;i<15;i++)
{
  if((i==10)
    break;
  Print(i);
}
```

The above example will execute the loop until **i** reaches **10**, in that case the break keyword

will terminate the loop. The code will produce these values: 0,1,2,3,4,5,6,7,8,9.

The Continue Statement:

The break statement takes you out the loop, while the continue statement will get you back to the top of the loop (parentheses).
For example:

```
for(int i=0;i<15; i++)
{
   if(i==10) continue;
   Print(i)
}
```

The above example will execute the loop until **i** reaches **10**, in that case the continue keyword will get the loop back to the top of the loop without printing **i** the tenth time. The code will produce these values: 0,1,2,3,4,5,6,7,8,9,11,12,13,14.

Latest note:

You can leave out some or all of the expressions in **for** loop if you want, for example:
for(;;)
This loop is like **while** loop with a test expression always set to true.
We will introduce the **while** loop to you right now.

The while Loop

The **for** loop usually used in the case you know how many times the loop will be executed. What happen if you don't know how many times you want to execute the loop?
This the **while** loop is for.

The **while** loop like the **for** loop has a Test expression. But it hasn't Initialization or Increment expressions.

This is an example:

```
int i=0;
while(i<15)
{
   Print(i);
   i++;
```

```
}
```

In the example you will notice the followings:

- The loop variable had declared and initialized before the loop, you can not declare or initialize it inside the parentheses of the **while** loop like the **for** loop.
- The **i++** statement here is not the increment expression as you may think, but the body of the loop must contain some statement that changes the loop variable, otherwise the loop would never end.

How the above example does work?

The **while** statement contains only the Test expression, and it will examine it every loop, if it's true the loop will continue, if it's false the loop will end and the control passes to the statement followed the loop section.

In the example the loop will execute till **i** reaches **16** in this case **i<15=false** and the loop ends.

Figure 2 shows a flow chart of the **while** loop.
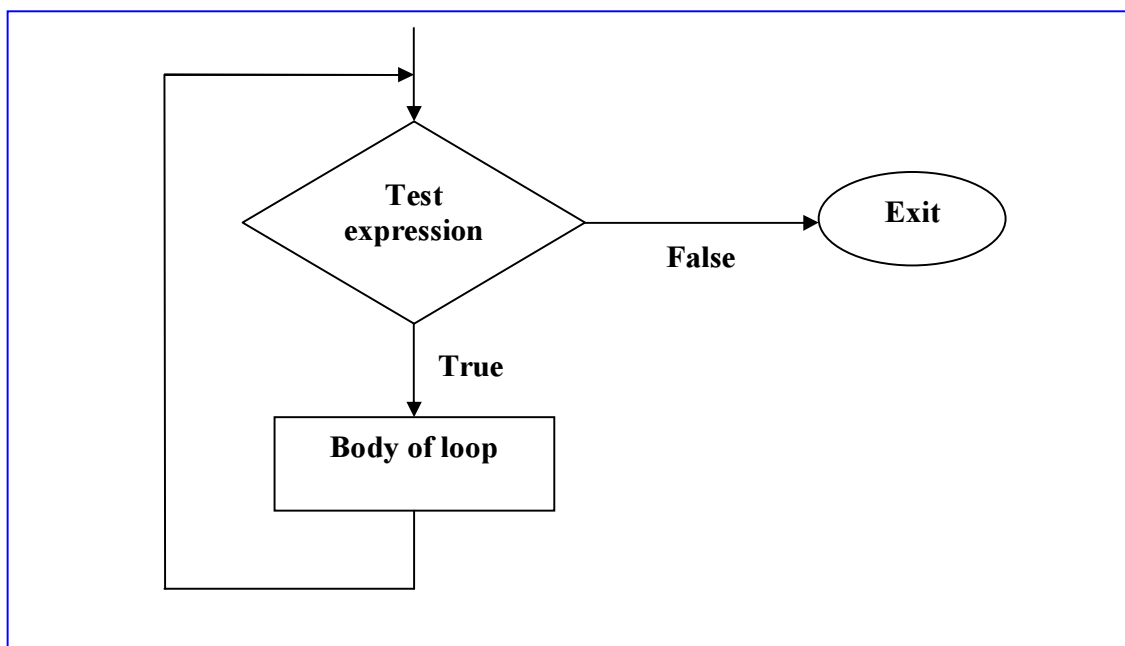
Figure 2 - Flow chart of the while loop

I told you before that the **while** loop is like the **for** loop, these are the **similar aspects**:

1. You can use **break** statement and **continue** in both of them.
2. You can single or multi statements in the body of the loop in both of them, in the case of using multi statements you have to delimit them by braces.
3. The similar copy of **for(;;)** is **while(true)**

I hope you enjoyed the lesson.
I welcome very much the questions and the suggestions.

See you
Coders' Guru
24-10-2005

MQL4 COURSE

By Coders' guru
www.forex-tsd.com

-6-
Loops & Decisions
Part 2

Welcome to the sixth lesson in my course about MQL4.
I hope you enjoyed the previous lessons.

In the previous lesson, we have talked about the Loops.
And we have seen that the Loops are one of two ways we use to change the normal flow of the program execution -from top to bottom. The second way is the Decisions.

Decisions in a program cause a one-time jump to a different part of the program, depending on the value of an expression.
These are the kinds of decisions statements available in MQL4:

The if Statement

The **if** statement is the simplest decision statement, here's an example:

```
if( x < 100 )
Print("hi");
```

Here the **if** keyword has followed by parentheses, inside the parentheses the **Test expression (x < 100)**, when the result of test expression is **true** the body of the **if** will execute (**Print("hi");**) ,and if it is false, the control passes to the statement follows the **if** block.

Figure 1 shows the flow chart of the **if** statement**:**
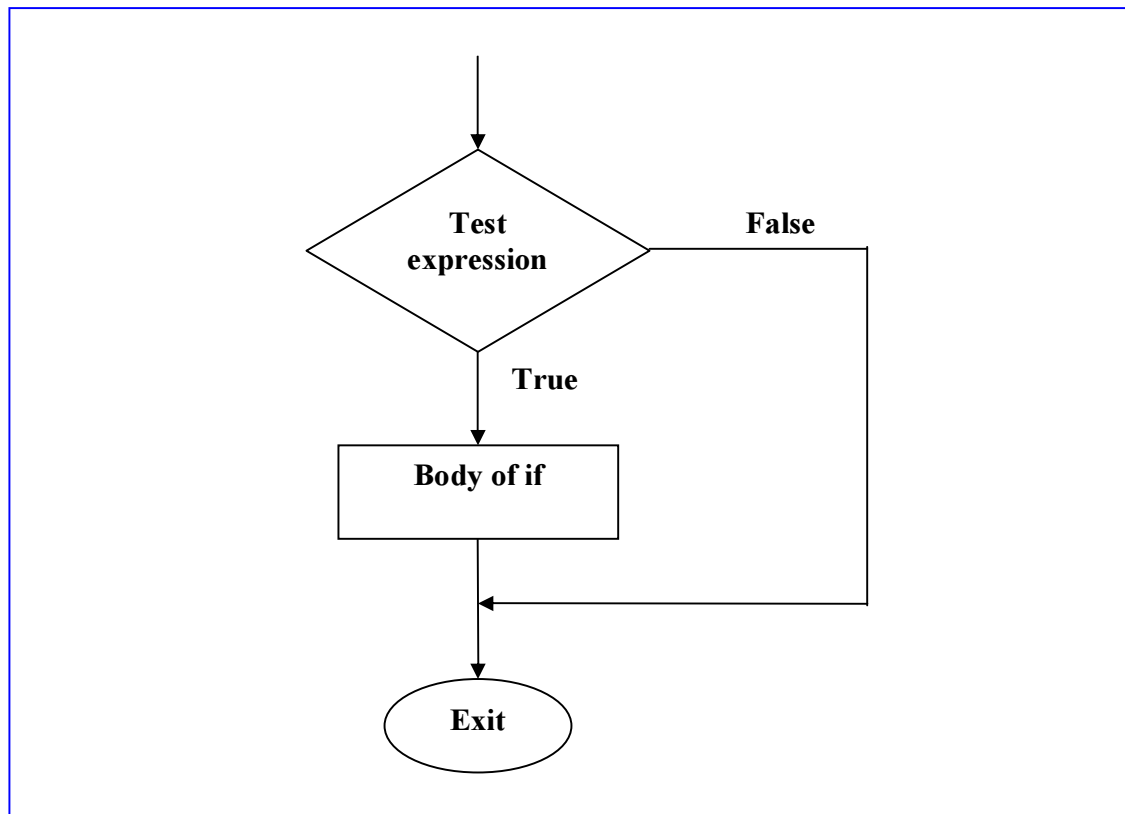
Figure 1 - Flow chart of the if statement

Multi Statements in the if Body:

Like the loops, the body of **if** can consist of more than statement delimited by braces.

For example:

```
if(current_price==stop_lose)
{
        Print("you have to close the order");
        PlaySound("warning.wav");
}
```

Notice the symbol == in the Test expression; it's one of the Relational Operators you have studied in the lesson 4, operations & expressions.
This is a source of a lot of errors, when you forget and use the assignment operator =.

Nesting:

The loops and decision structures can be basted inside one another; you can nest **ifs**

inside loops, loops inside **ifs, ifs** inside **ifs**, and so on.

Here's an example:

```
for(int i=2 ; i<10 ; i++)
        if(i%2==0)
        {
                Print("It's not a prime nomber");
                PlaySound("warning.wav");
        }
```

In the previous example the **if** structure nested inside the for loop.

*Notice: you will notice that there are no braces around the loop body, this is because the **if** statement and the statements inside its body, are considered to be a single statement.*

The if...else Statement

The **if** statement let's you to do something if a condition is true, suppose we want to do another thing if it's false. That's the **if...else** statement comes in.
It consist of **if** statement followed by statement or a block of statements, then the **else** keyword followed by another statement or a block of statements.

Like this example:

```
if(current_price>stop_lose)
        Print("It's too late to stop, please stop!");
else
        Print("you playing well today!");
```

If the test expression in the **if** statement is true, the program one message, if it isn't true, it prints the other.

Figure 2 shows the flow chart of the **if…else** statement**:**
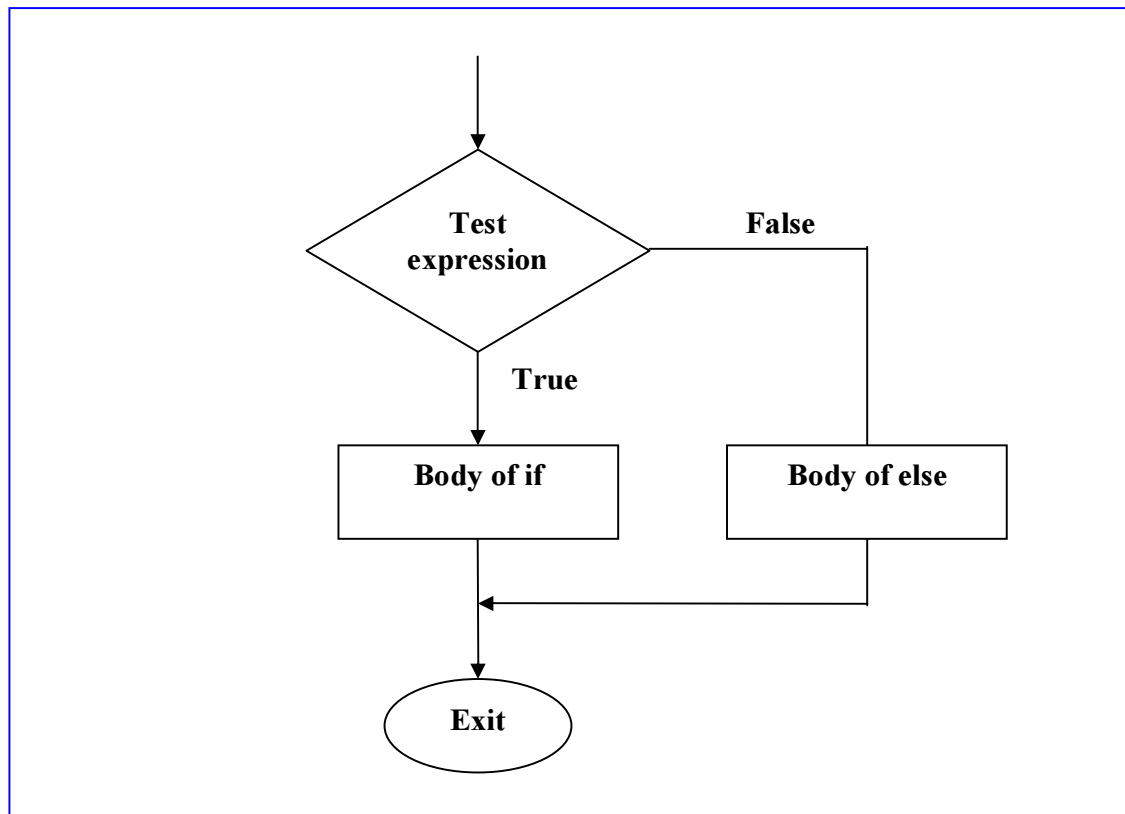
Figure 2 - Flow chart of the if..else statement

Nested if…else Statements

You can nest if…else statement in ifs statements, you can nest if…else statement in if…else statement, and so on.

Like this:

```
if(current_price>stop_lose)
        Print("It's too late to stop, please stop!");
if(current_price==stop_lose)
        Print("It's time to stop!");
else
        Print("you playing well today!");
```

There's a potential problem in nested if…else statements, you can inadvertently match an else with the wrong if.

To solve this case you can do one of two things:

1- you can delimited the **if…else** pairs with braces like this:

```
if(current_price>stop_lose)
{
        Print("It's too late to stop, please stop!");
if(current_price==stop_lose)
        Print("It's time to stop!");
else
        Print("you playing well today!");
}
```

2- If you can't do the first solution (in the case of a lot of if…else statements or you are lazy to do it) take it as rule.
Match else with the nearest if. (Here it's the line **if(current_price==stop_lose)**).

The switch Statement

If you have a large decision tree, and all the decisions depend on the value of the same variable, you can use a switch statement here.
Here's an example:

```
switch(x)
  {
  case 'A':
    Print("CASE A");
    break;
  case 'B':
  case 'C':
    Print("CASE B or C");
        break;
  default:
    Print("NOT A, B or C");
    break;
  }
```

In the above example the switch keyword is followed by parentheses, inside the parentheses you'll find the **switch constant**, this constant can be an integer, a character constant or a constant expression. The constant expression mustn't include variable for example:
case X+Y: is invalid switch constant.

How the above example works?

The switch statement matches the constant **x** with one of the **cases constants**.
In the case **x=='A'** the program will print **"CASE A"** and the **break** statement will take you the control out of the switch block.

In the cases **x=='B'** or **x=='C'**, the program will print **"CASE B or C"**. That's because there's no **break** statement after **case 'B':**.

In the case that **x !=** any of the cases constants the switch statement will execute the **default** case and print **"NOT A, B or C".**

Figure 3 shows the flow chart of the switch statement
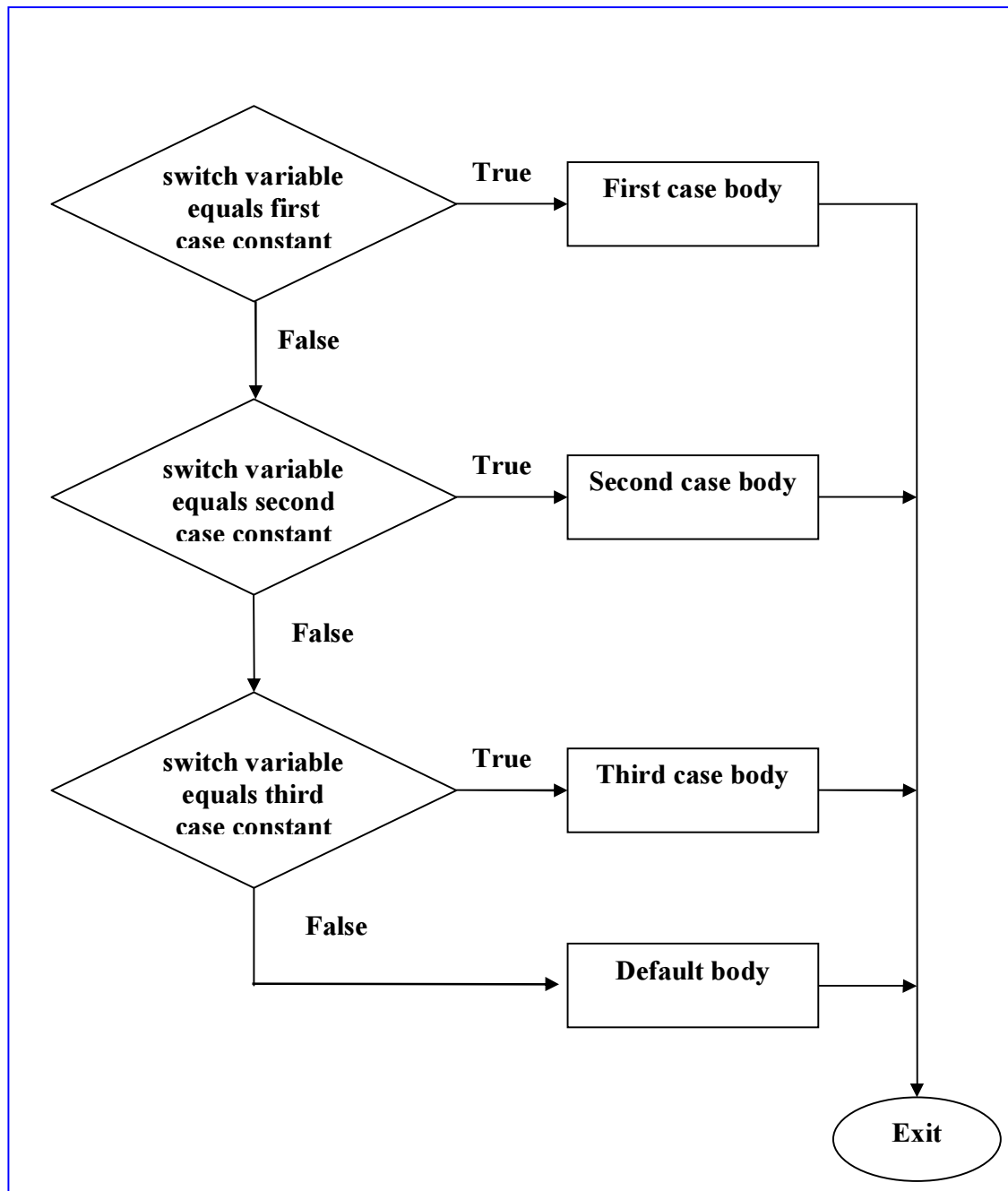
Figure 3 - Flow chart of the switch statement

I hope you enjoyed the lesson.
I welcome very much the questions and the suggestions.

See you
Coders' Guru
25-10-2005

MQL4 COURSE

By Coders' guru
www.forex-tsd.com

-7-
Functions

Welcome to the world of MQL4 Functions.
The functions in any language take two phases:
Learning them which sometimes a boring thing.
Using them which always a **lifeboat.**

I want to tell you my traditional sentence:
I hope you enjoyed the previous lessons, which you can download them from here:

1- Lesson 1 - Welcome to the MQL4 course.
   http://www.forex-tsd.com/attachment.php?attachmentid=372
2- Lesson 2 – SYNTAX.
   http://www.forex-tsd.com/attachment.php?attachmentid=399
3- Lesson 3 - MQL4 Data types.
   http://www.forex-tsd.com/attachment.php?attachmentid=469
4- Lesson 4 - MQL4 Operations & Expressions.
   http://www.forex-tsd.com/attachment.php?attachmentid=481
5- Lesson 5- Loops & Decisions (Part1).
   http://www.forex-tsd.com/attachment.php?attachmentid=504
6- Lesson 6 - Loops & Decisions (Part2).
   http://www.forex-tsd.com/attachment.php?attachmentid=547

Let's start the seventh lesson.

What's the meaning of functions?

The function is very like the sausage **machine**, you input the **meat** and the **spices** and it outs the **sausage**.
The meat and the spices are the **function parameters**; the sausage is the **function return** value. The machine itself is the **function body**.
There's only one difference between the functions and your sausage machine, some of the functions will return nothing (nothing in MQL4 called **void**).

Let's take some examples:

```
double                      // type of the sausage – return value
my_func (double a, double b, double c) // function name and parameters list (meat &
spices)
  {
  return (a*b + c);          // sausage outs - returned value
  }
```

As you see above, the function starts with the **type** of the returned value "**double**"
followed by the **function name** which followed by parentheses.
Inside the parentheses you put the meat and spices, sorry, you put the **parameters** of the
function.
Here we have put three parameters **double a, double b, double c**.
Then the function body starts and ends with braces. In our example the function body will
produce the operation **(a*b + c)**.
The **return** keyword is responsible about returning the final result.

Return keyword:

The return keyword terminate the function (like the **break** keyword does in the loop), and
it gives the control to the **function caller** (we will know it soon).
The return keyword can include an expression inside its parentheses like the above
example **return (a*b + c);** and this means to terminate the function and return the result
of the expression.
And it can be without expression and its only job in this case is to terminate the function.

Notice: Not all the functions use the return keyword, especially if there's no return value.
Like the next example:

```
void                       // void mean there's no sausage – returned value.
my_func (string s) // function name and parameters list (meat & spices)
  {
     Print(s);
  }
```

The function above will not return value, but it will print the parameter **s** you provided.
When the function has no return value you use "**void**" as the funciotn returns type.
These kinds of functions in some programming language called "**Methods**", but MQL4
calling them functions.

Function call:

We know very well now what the function is (I hope)? How to use the functions in your MQL4?
There's an extra steps after writing your function to use the function in you program.
This step is **calling it** (using it).

Assume you have a function which collects the summation of two integers.
This is the function:

```
int collect (int first_number, int second_number)
  {
     return(first_number+ second_number);
  }
```

You know how the previous function works, but you want to use it.

You use it like this:

```
int a = 10;
int b = 15;
int sum = collect(a,b);
Print (sum);
```

The example above will print **25** (is it a magic). But how did it know?
The magic line is **int sum = collect(a,b);** here you declared a variable (**sum**) to hold the function return value and gave the function its two parameters (**a,b**).
You basically **called the function.**
MQL4 when see your **function name**, it will take you parameters and go to the function and it will return –soon- with the result and place them in same line.
It's very like copying all the lines of the function instead of the place you called the function in, easy right?

Nesting functions inside function:

You can nest function (or more) inside the body of another function. That's because the caller line is treated like any normal statement (it's actually a statement).

For example:

We will use the collect function described above inside another new function which its job is printing the result of the collection:

```
void print_collection (int first_number, int second_number)
  {
     int sum = collect(first_number, second_number);
     Print(sum);
```

```
    }
```

Here we called the **collect** function inside the **print_collection** function body and printed the result. **void** means there's no return vale (do you still remember?).

MQL4 Special functions init(), deinit() and start():

In MQL4, every program begins with the function **"init()"** (initialize) and it occurs when you attach your program(Expert advisor or Custom indicator) to the MetaTrader charts or in the case you change the financial symbol or the chart periodicity. And its job is initializing the main variables of your program (you will know about the variables initialization in the next lesson).

When your program finishes its job or you close the chart window or change the financial symbol or the chart periodicity or shutdown MetaTrader terminal, the function **"deinit()"** (de-initialize) will occur.

The third function (which is the most important one) **"start()"** will occur every time new quotations are received , you spend 90 of your programming life inside this function.

We will know a lot about these functions in our **real world lessons** when we write our own Expert advisor and Custom Indictor**.**

I hope you enjoyed the lesson.
I welcome very much the questions and the suggestions.

See you
Coders' Guru
25-10-2005

MQL4 COURSE

By Coders' guru
www.forex-tsd.com

-8-
Variables

Welcome to my MQL4 variables.

I hope you enjoyed the previous lessons and I hope you are ready for the variables challenge:
I recommend you to read the "DATA TYPES" lesson before reading this lesson.
You can download it here:
http://forex-tsd.com/attachment.php?attachmentid=469

Now, let's **enjoy** the Variables.

What are the variables mean?

As I told you the secret before, the variables are the names that refer to sections of memory into which data can be stored.

To help you think of this as a picture, imagine that memory is a series of different size boxes. The box size is memory storage area required in bytes.

- In order to use a box to store data, the box must be given a name; this process is known as **declaration.**
- In the declaration process you use a word tell the computer what's the kind and size of the box you want to use, this word known as **keyword**.
- It helps if you give a box a meaningful name that relates to the type of information which make it easier to find the data, this name is the **variable constant**.
- Data is placed into a box by **assigning** the data to the box.
- When we set the value of the box you have created in the same line you declared the variable; this process is known as **initialization.**

When we create a variable we are telling the computer that we want him to assign a specified memory length (in bytes) to our variable, since storing a simple number, a letter or a large number is not going to occupy the same space in memory, so the computer will

ask us what's the kind of data and how much the length of the data? That is the Data type for.

For example if we said this line of code to the computer:

```
int  MyVaraible=0;
```

That's mean we are asking the computer to set a block of 4 bytes length to our variable named "MyVaraiable".

In the previous example we have used:
int ← Keyword
int ← Integer data type.
int ← Declaration
MyVaraible ← Variable's constant.
=0 ← Initialization

We will know more about variables in a coming lesson.

In MQL4, these are the kinds of Data types:

- Integer (int)
- Boolean (bool)
- Character (char)
- String (string)
- Floating-point number (double)
- Color (color)
- Datetime (datetime)

I've copied the previous few lines from the DATA TYPES lesson for you. To know what's the variable, now how do to declare the variables:

Declaration:

Declaring a variable means to introduce it to the world and specify its type. By using the **keywords** you have learned in the DATA TYPES lesson (int, double, char, bool, string, color and datetime) with the name you chose to the variable.

For example:

```
int  MyVaraible;
```

Here you declared a variable named **MyVaraible** which is an integer type. And before the declaration you can't use the **MyVariable** in your code. If you used it without declaration the MQL4 compiler will complain and will tell you something like this:**'MyVaraible' - variable not defined**. **1 error(s), 0 warning(s).**

Initialization:

Initializing the variable means to assign a value to it, for example **MyVaraible=0**;
You can initialize the variable at the same line of the declaration like the example:
int MyVaraible=0;

And you can declare the variable in one place and initialize it in another place like this:

```
int  MyVaraible;
…
…
MyVaraible=5;
```

But keep in your mind this fact**: the declaration must be before the initialization**.

Scopes of variables:

There are two scopes of the variables, Local and Global.

Scope means, which part of code will know about the variable and can use it.

Local variable means they are not seen to outside world where they had declared. For example the variables declared inside function are local to the function block of code, and the variables declared inside the loop or decisions block of code are local to those blocks and can be seen or used outside them.

For example:

```
double my_func (double a, double b, double c)
  {
  int d ;
  return (a*b + c);
  }
```

In the above example the variables **a**,**b**,**c** and **d** are local variables, which can be used only inside the function block of code (any thing beside the braces) and can't be used by outside code. So we can't write a line after the function above saying for example: **d=10**; because **d** is not seen to the next line of the function because it's outside it.

The second kind of the scopes is the **Global variables**, and they are the variables which had declared outside any block of code and can be seen from any part of your code.

For example:

```
int Global_Variable;
double my_func (double a, double b, double c)
  {
    return (a*b + c + Global_Variable);
  }
```

Here the variable **Global_Variable** declared outside the function (function level declaration) so, it can be seen by all the functions in you program.
The Global variables will automatically set to **zero** if you didn't initialize them.

Extern variables:

The keyword "**extern**" used to declare a special kind of variables; those kinds of variables are used to define input date of the program, which you can set them form the property of your Expert advisor or Custom indicator.

For example:

```
extern color Indicator_color = C'0x00,0x00,0xFF'; // blue
int init()
  {
   ...
  }
```

Here the variable **Indicator_color** had defined as an **extern** variable which you will see it the first time you attach your indicator (or EA) to the MetaTrader chart and which you can change it from the properties sheet windows. Look at Figure 1.
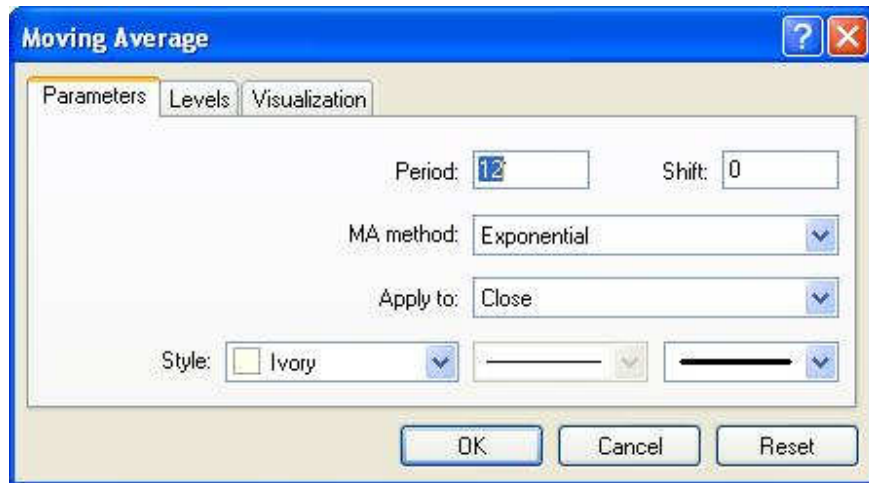
Figure 1: Property sheet of MA indicator

Here the variables **Period**, **Shift**, **MA_method**, **Apply_to** and **Style** are variables
defined using the "**extern**" keyword so they appear in the property sheet.
Any variable you want the user of your program be able to change and set, make it
extern variable.

I hope you enjoyed the lesson.

I welcome very much the questions and the suggestions.

See you
Coders' Guru
29-10-2005

MQL4 COURSE

By Coders' guru

-9-
Preprocessors

Welcome to my last **theoretical** lesson in this series.
In the next series of lessons we will start to build our first Customer Indicator,
So I recommend you to read all the nine lessons carefully before the real work starts.

Now, let's enjoy the Preprocessors:

What are the Preprocessors mean?

Preprocessors are the instructions you give to the compiler to carry them out before starting (processing) your code.

For example if you used the preprocessor directive **#include <win32.h>** that's mean you telling the compiler to include the content of the file "**win32.h**" in the place you wrote the include keyword before processing your code.

In MQL4 there are four of preprocessors directives:

1- define directive:

define directive used to generate a constant.

The constant is very like the variable with only one different, you set its value only once and you can not change its value in your code like the variable.

For example:

```
#define my_constant          100
```

As you can notice in the above example there's no assignment symbol (=) but only space between the constant name (**my_constant**) and its value (**100**).

And you can notice too that the line didn't end with semi-colon but it ended with a carriage-return character (new line).

The name of constant obeys the same rules you had learnt about choosing the identifier names (lesson 2 SYNTAX), for example you can't start the constant name with a number or exceeds 31 characters.

The value of the content can be any type you want.

The compiler will replace each occurrence of constant name in your source code with the corresponding value.

So you can use the above constant in your code like that:

```
sum = constant1 * 10;
```

2- property directive:

There are predefined constants called "**Controlling Compilation**" included in the MQL4 language, which you can set them in your program.

They are the properties of your program which you can set them using the compiler directive "**property**" and the compiler will write them in the settings of your executable program (ex4 file).

For example:

```
#property link       "http://www.forex-tsd.com"
#property copyright   "Anyone wants to use"
```

This is the list of the MQL4 predefined constants:

Constant	Type	Description
link	string	a link to the company website
copyright	string	the company name

stacksize	int	stack size
indicator_chart_window	void	show the indicator in the chart window
indicator_separate_window	void	show the indicator in a separate window
indicator_buffers	int	the number of buffers for calculation, up to 8
indicator_minimum	int	the bottom border for the chart
indicator_maximum	int	the top border for the chart
indicator_colorN	color	the color for displaying line N, where N lies between 1 and 8
indicator_levelN	double	predefined level N for separate window custom indicator, where N lies between 1 and 8
show_confirm	void	before script run message box with confirmation appears
show_inputs	void	before script run its property sheet appears; disables show_confirm property

3- include directive:

When you asking the compiler to include a file name with the "**include**" directive, it's very like when you copy the entire file content and paste it in the place of the line you write include.

 For example:

```
#include <win32.h>
```

In the above example you telling the compiler to open the file "**win32.h**" and reads all of its content and copy them in the same place of the include statement.

Note: in the above example you enclosed the file name with Angle brackets (<>) and that's mean you telling the compiler to use the default directory (usually, terminal_directory\experts\include) to search for the file **win32.h** and don't search the current directory.

If the file you want to include located at the same path of your code, you have to use quotes instead of angle brackets like this:

```
#include "mylib.h"
```

In the both cases if the file can't be found you will get an error message.

You can use include at anywhere you want but it usually used at the beginning of the source code.

Tip: *It's a good programming practice to write the frequently used code in a separate file and use include directive to put it in your code when you need (just an advice).*

4- import directive:

It's like include directive in the aspect of using outside file in your program.

But there are differences between them.

You use **import** only with MQL4 executables files (.ex4) or library files (.dll) to import their functions to your program.

For example:

```
#import "user32.dll"
  int MessageBoxA(int hWnd,string lpText,string lpCaption,
              int uType);
  int MessageBoxExA(int hWnd,string lpText,string lpCaption,
              int uType,int wLanguageId);
#import "melib.ex4"
#import "gdi32.dll"
  int    GetDC(int hWnd);
  int    ReleaseDC(int hWnd,int hDC);
#import
```

When you import functions from "ex4" file you haven't to declare their functions to be ready for use.

While importing the functions from a ".dll" file requires you to declare the functions you want to use like this:

```
int MessageBoxA(int hWnd,string lpText,string lpCaption,
              int uType);
```

And only the functions you has declared you can use in your code.

You must end the import directives with a blank import line **#import** (without parameters).

I hope you enjoyed the lesson. And I hope you are ready now for your first Custom Indicator.

I welcome very much the questions and the suggestions.

See you
Coders' Guru
30-10-2005

MQL4 COURSE

By Coders' guru

-10-
Your First Indicator
Part 1

Welcome to the practical world of MQL4 courses; welcome to your first indicator in MQL4.

I recommend you to read the previous nine lessons very carefully, before continuing with these series of courses, that's because we will use them so much in our explanations and studies of the Expert Advisors and Custom Indicators which we will create in this series of lessons.

Today we are going to create a simple indictor which will not mean too much for our trade world but it means too much to our MQL4 programming understanding.

It simply will collect the subtraction of **High []** of the price – **Low []** of the price; don't be in a hurry, you will know everything very soon.

Let's swim!

MetaEditor:

This is the program which has been shipped with MT4 (MetaTrader 4) enables you to write your programs, read MQL4 help, compile your program and More.

I've made a shortcut for MetaEditor on my desktop for easily access to the program. If you want to run MetaEditor you have three choices.

1- Run MT4, then click **F4**, choose MetaEditor from Tools menu or click its icon on the Standard toolbar (Figure 1).
2- From Start menu➔ Programs, find MetaTrader 4 group then click MetaEditor.
3- Find the MT4 installation path (usually C:\Program Files\MetaTrader 4), find the MetaEditor.exe and click it (I recommend to make a shortcut on you desktop).

Figure 1 – MetaTrader Standard Toolbar

Any method you have chosen leads you to MetaEditor as you can see in figure 2.

As you can see in figure 2, there are three windows in MetaEditor:

1- The Editor window which you can write your program in.
2- The Toolbox window which contains three tabs:
   a. Errors tab, you see here the errors (if there any) in your code.
   b. Find in files tab, you see here the files which contain the keyword you are searching for using the toolbar command "Find in files" or by clicking CTRL +SHIFT+ F hotkeys.
   c. Help tab, you can highlight the keyword you want to know more about it and click F1, and you will see the help topics in this tab.
3- The Navigator window which contains three tabs:
   a. Files tab, for easy access to the files saved in the MT4 folder.
   b. Dictionary tab enables you to access the MQL4 help system.
   c. Search tab enables you to search the MQL4 dictionary.
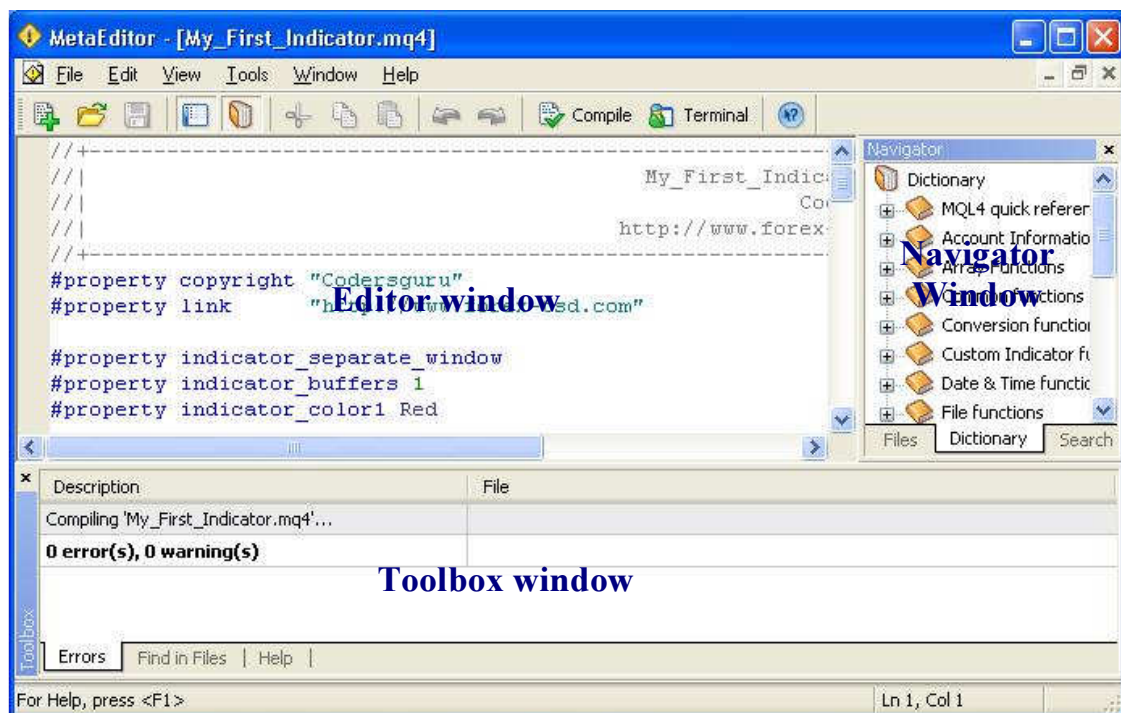
Figure 2 – MetaEditor Windows

I recommend you to navigate around the MetaEditor Menus, Toolbar and windows to be familiar with it.

Now let's enjoy creating our first custom indicator.

Custom Indicator is a program which enables you to use the functions of the technical indicators and it cannot automate your deals.

First three steps:

Now you have run your MetaEditor and navigated around its Menus, Toolbar and windows, let's USE it.

To create a custom indicator you have to start with three steps (you will learn later how to skip these boring steps (my personal opinion).

Step 1: Click **File** menu → **New** (you use CTRL+N hotkey or click the New Icon in the Standard toolbar).
You will get a wizard (Figure 3) guiding you to the next step.
Choose Custom Indicator Program option, and then click next.
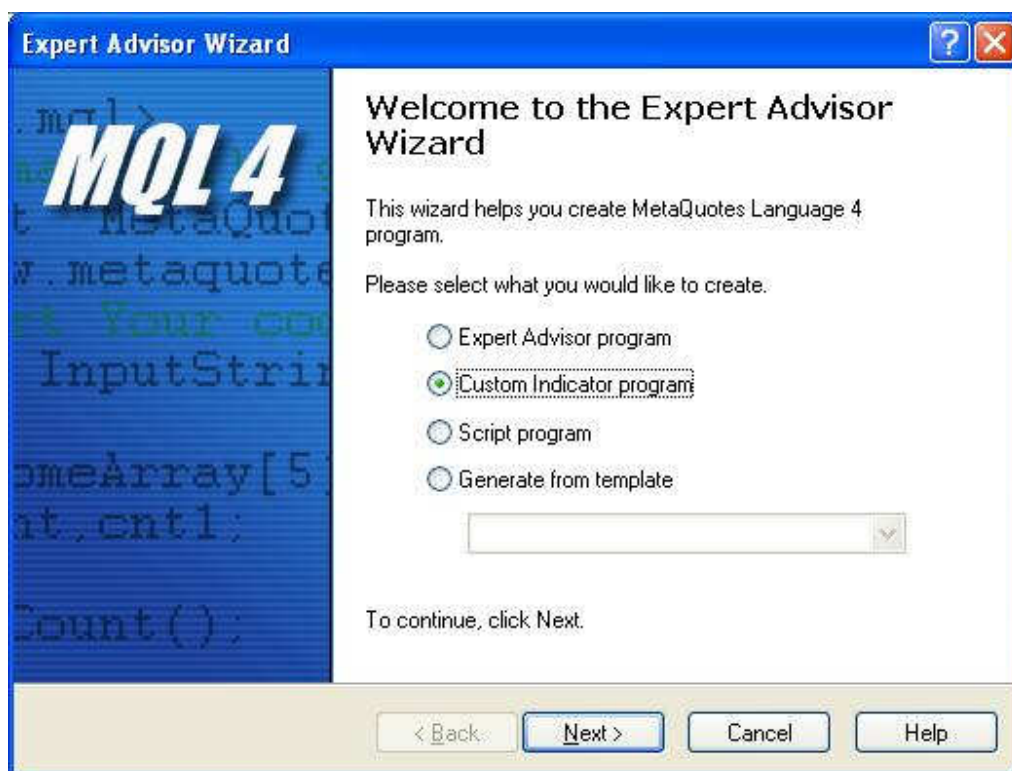
Figure 3 - New project wizard

Step 2: When you clicked **Next**, you will get the second step wizard (Figure 4) which will enable you to edit the properties of your program. In this step you can enter these properties:

1- **Name** of your program, this is the name which the world will call you program with and it will be saved as **the_name_you_have_chosen.mq4**

2- **Author** name, the creator of the program name.

3- **Link** to your web site.

4- **External** variables list: I want to pause here to remember you about external variable.

External variables are the variables which will be available to the user of you indicator to set from the properties tab of your indicator in MetaTrader. For example: **MA_Period** in the very popular **EMA** indicator. And these variables will be declared with the "**extern**" keyword in your code (Please review **Variables** lesson).

So, this section of the wizard enables you to add these kinds of variables.

In our first indicator example we will not need any external variables just write the values you see in figure 4 and let's go to step 3 by clicking **Next** button.

Figure 4 – Program properties wizard.

Step 3: The third wizard you will get when you clicked **Next** button is the Drawing properties wizard (Figure 5).

Its job is enabling you to set the dawning properties of the lines of your indicator, for example: how many lines, colors and where to draw your indicator (in the main chart or in separate windows).

This wizard contains the following options:

1- **Indicator in separate window** option: by clicking this option, your indicator will be drawn in separate windows and not on the main chart window. If you didn't check the option, your indicator will be drawn in the main chart window.
2- **Minimum** option: it will be available (enabled) only if you have checked the Indicator in separate window option, and its job is setting the bottom border for the chart.
3- **Maximum** option: it will be available (enabled) only if you have checked the Indicator in separate window option, and its job is setting the top border for the chart
4- **Indexes** List: here you add your indicator line and set its default colors.

I want you to wait to the next lesson(s) to know more about these options and don't be in a hurry.

For our first indicator example, choose Indicator in separate window option and click **Add** button, when you click add button the wizard will add a line to the indexes list like you see in figure 5.
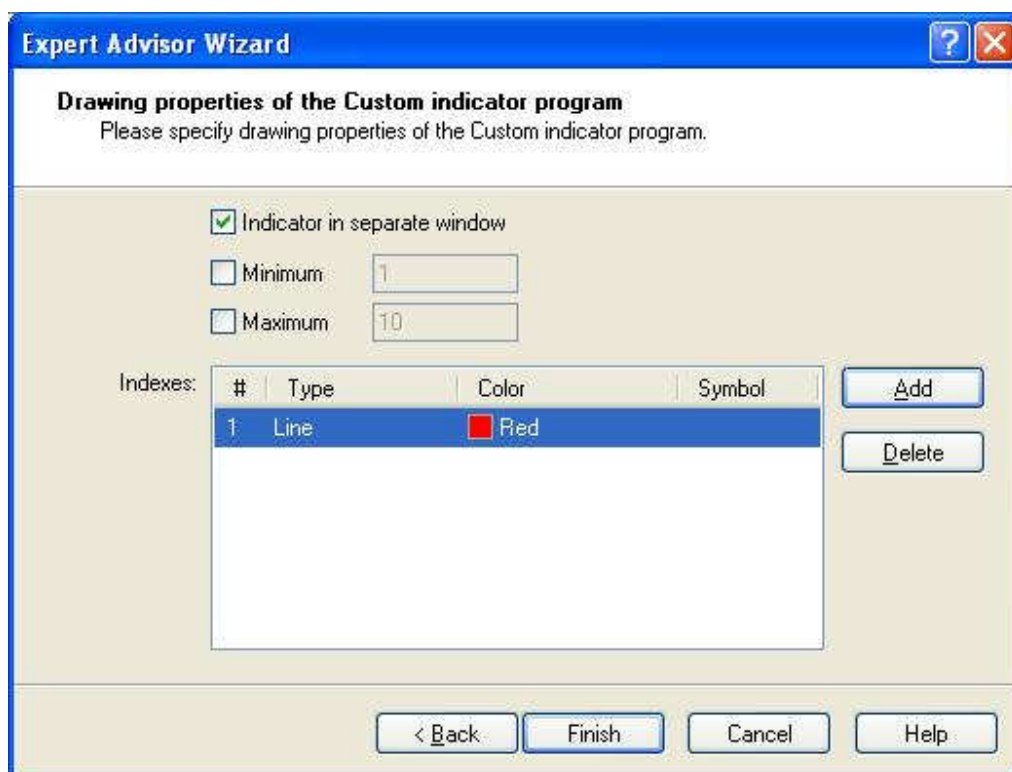
Figure 5 - Drawing properties wizard.

When you click **Finish** button the Magic will start. You will see the wizard disappeared and the focus returned to the MetaEditor environment and… guess what?

You have ready to use first indicator draft code.

This is the code you will get:

```mq4
//+------------------------------------------------------------------+
//|                                          My_First_Indicator.mq4 |
//|                                                       Codersguru |
//|                                            http://www.forex-tsd.com |
//+------------------------------------------------------------------+
#property copyright "Codersguru"
#property link      "http://www.forex-tsd.com"

#property indicator_separate_window
#property indicator_buffers 1
#property indicator_color1 Red
//---- buffers
double ExtMapBuffer1[];
//+------------------------------------------------------------------+
//| Custom indicator initialization function                         |
//+------------------------------------------------------------------+
int init()
  {
//---- indicators
   SetIndexStyle(0,DRAW_LINE);
   SetIndexBuffer(0,ExtMapBuffer1);
//----
   return(0);
  }
//+------------------------------------------------------------------+
//| Custor indicator deinitialization function                       |
//+------------------------------------------------------------------+
int deinit()
  {
//----

//----
   return(0);
  }
//+------------------------------------------------------------------+
//| Custom indicator iteration function                              |
//+------------------------------------------------------------------+
int start()
  {
   int     counted_bars=IndicatorCounted();
//----

//----
   return(0);
  }
//+------------------------------------------------------------------+
```

As you see in the above code, the wizard has written a lot of code for you, now I have to thank the wizard and to thank you too.

In the next lesson we will discover every line of code you have seen above and add our code to make our first indicator. To this lesson I hope you be ready!

Please don't forget to download the source code of the first indicator and warm yourself for the next lesson.

I welcome very much the questions and the suggestions.

See you
Coders' Guru
01-11-2005

MQL4 COURSE

By Coders' guru
www.forex-tsd.com

-11-
Your First Indicator
Part 2

Welcome to the second part of **"Your First Indicator"** lesson.
In the previous lesson we didn't write any line of code, that's because the New Project
Wizard wrote all the code for us. Thanks!

Today we are going to add few lines to the code the wizard had generated to make our
program more useful.
Afterwards, we are going to explain the whole of the code line by line.

Let's coding

Code we have added:

We have added the code which in a **bold dark blue** to our previous code:

```
//+------------------------------------------------------------------+
//|                                           My_First_Indicator.mq4 |
//|                                                       Codersguru |
//|                                          http://www.forex-tsd.com |
//+------------------------------------------------------------------+
#property copyright "Codersguru"
#property link      "http://www.forex-tsd.com"

#property indicator_separate_window
#property indicator_buffers 1
#property indicator_color1 Red

//---- buffers
double ExtMapBuffer1[];

//+------------------------------------------------------------------+
//| Custom indicator initialization function                         |
//+------------------------------------------------------------------+
int init()
  {
//---- indicators
   SetIndexStyle(0,DRAW_LINE);
   SetIndexBuffer(0,ExtMapBuffer1);
   string short_name = "Your first indicator is running!";
   IndicatorShortName(short_name);
```

```
//----
   return(1);
   }
//+------------------------------------------------------------------+
//| Custor indicator deinitialization function                       |
//+------------------------------------------------------------------+
int deinit()
   {
//----

//----
   return(0);
   }
//+------------------------------------------------------------------+
//| Custom indicator iteration function                              |
//+------------------------------------------------------------------+
int start()
   {
   int     counted_bars=IndicatorCounted();

//---- check for possible errors
   if (counted_bars<0) return(-1);
//---- last counted bar will be recounted
   if (counted_bars>0) counted_bars--;

   int     pos=Bars-counted_bars;

   double dHigh , dLow , dResult;
   Comment("Hi! I'm here on the main chart windows!");

//---- main calculation loop
   while(pos>=0)
     {
         dHigh = High[pos];
         dLow = Low[pos];
         dResult = dHigh - dLow;
         ExtMapBuffer1[pos]= dResult ;
         pos--;
     }
//----
   return(0);
   }
//+------------------------------------------------------------------+
```

How will we work?

We will write the line(s) of the code we are going to explain then we will explain them afterwards, if there are no topics, we will explain the line(s) of code directly. But at the most of the time we will pause to discuss some general topics.

I want to here your suggestion about this method please!

Now let's crack this code line by line.

```
//+------------------------------------------------------------------+
//|                                        My_First_Indicator.mq4 |
//|                                                    Codersguru |
//|                                        http://www.forex-tsd.com |
//+------------------------------------------------------------------+
```

Comments:

The first five lines of code (which are in gray color) are comments.
You use Comments to write lines in your code which the compiler will ignore them.
You are commenting your code for a lot of reasons:

- To make it clearer
- To document some parts like the copyright and creation date etc.
- To make it understandable.
- To tell us how the code you have written is work.
- …

You can write comments in two ways:

Single line comments: The Single line comment starts with "//" and ends with the new line.

Multi-line comments: The multi-line comment start with "/*" and ends with "*/" and you can comment more than one line.

In our program the MQL4 wizard gathered from the data we entered the **name** of the program, **author** and the **link** and wrote them as comments at the top of our program.

```
#property copyright "Codersguru"
#property link        "http://www.forex-tsd.com"

#property indicator_separate_window
#property indicator_buffers 1
#property indicator_color1 Red
```

Property directive:

As you notice all of these lines start with the word (**#property**). That's because they are kind of the **Preprocessors directives** called **property directives**.

The **Preprocessors** are the instructions you give to the compiler to carry them out before starting (processing) your code.

The **property directives** are predefined constants called "**Controlling Compilation**" built in the MQL4 language; their job is setting the properties of your program. For example: is your Indicator will appear in the main chart window or in a separate window? Who is the writer of the program?

Note: *The preprocessors lines end with a carriage-return character (new line) not a semi-colon symbol.*

We will try to discuss here the property directives available in MQL4.

link:

This property setting the web link to your web site which you asked to enter it in step 2 in the Expert Advisor Wizard (review the previous lesson).

The data type of this property is string.

copyright:

It's the name of the author of the program, same as the link property you asked to enter it in step 2 in the Expert Advisor Wizard.

The data type of this property is string.

stacksize:

It's an integer value sets the memory size for every thread, the default value is 16384**.**

The data type of this property is integer.

indicator_chart_window:

When you set this property, your indicator will be drawn in the main chart window (Figure 1). You have to choose one of two options for your Indicators, drawing them in the main chart windows by using this property, or drawing them in separate windows by choosing **the indicator_separate_window**. You can't use the both of them at the same time.

The data type of this property is void, which means it takes no value.

indicator_separate_window:

When you set this property, your indicator will be drawn in a separate window (Figure 1). You can set the scale of the separate indicator window using two properties **indicator_minimum** for the minimum value and **indicator_maximum** for the maximum value of the scale.

And you can set the level of your indicators on these scales using the property **indicator_levelN** where's the **N** is the indicator number.

Both of the properties **indicator_chart_window** and **indicator_separate_window** are void data type, which mean they don't take value and you just write them.

In our program we will draw our indicator in a separate window:

`#property indicator_separate_window`

Figure 1

indicator_minimum:

With the aid of this property we are setting the minimum value of the separate windows scale, which is the bottom border of the windows. For example:

#propery indicator_minimum   0

#propery indicator_ maximum   100

Here we have set the bottom border of the window to 0 and the top border to 100 (see indicator_ maximum), hence we have a scale ranged from 0 to 100 in our separate window which we are drawing our indicator.

The data type of this property is integer.

indicator_maximum:

With the aid of this property we are setting the maximum value of the separate windows scale, which is the top border of the windows.

This value must be greater than the **indicator_minimum** value.

The data type of this property is integer.

indicator_levelN:

With the aid of this property we are setting the level of the indicator in the scale we have created with the properties **indicator_minimum** and **indicator_maximum**, this value must be greater than the **indicator_minimum** value and smaller than the **indicator_maximum** value.

N is the indicator number which we are setting its level, it must range from 1 to 8 (because we are allowed only to use up to 8 indicator buffers in our program, so we can set the **indicator_level** for each of them using its number). For example:

#propery indicator_minimum   0

#propery indicator_minimum   100

#propery indicator_level1   10   //set the first indicator buffer level

#propery indicator_level2   65.5   //set the second indicator buffer level

The data type of this property is double.

indicator_buffers:

With the aid of this property we are setting the number of memories spaces (Arrays) allocated to draw our line(s). When we set the number (ranged from **1** up to **8**) we are telling MQL4: "Please allocate a memory space for me to draw my indicator line".

In our program we used only one buffer.

```
#property indicator_buffers 1
```

That's because we will draw only one line.

indicator_colorN:

We can use up to 8 lines in our indicator, you can set the color of each of them using this property **indicator_colorN** , where the **N** is the line number which defined by **indicator_buffers**.

The user of your Indicator can change this color from the properties dialog of your Indicator (Figure 2).

In our program the indicator line color will be red.

```
#property indicator_color1 Red
```

The data type of this property is color.
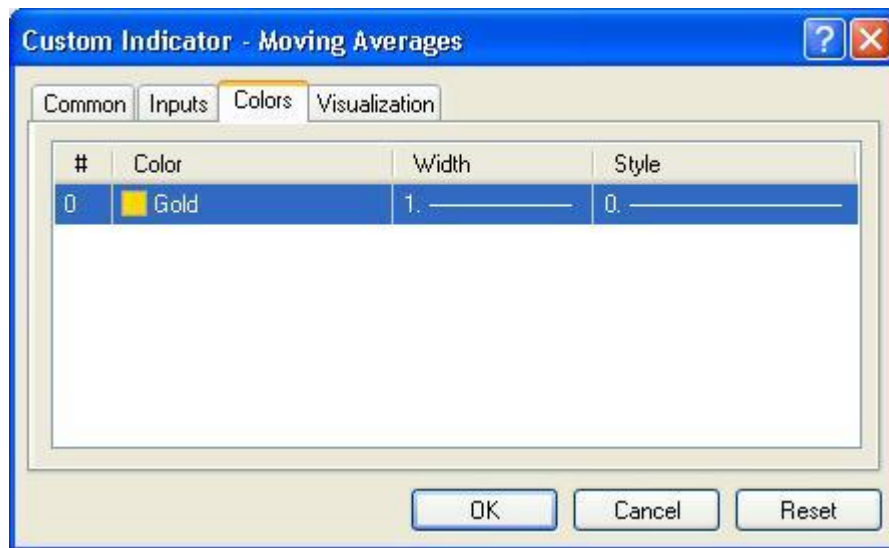
Figure 2

```
double ExtMapBuffer1[];
```

Arrays:

In our life we usually group similar objects into units, in the programming we also need to group together the data items of the same type. We use Arrays to do this task.

Arrays are very like the list tables, you group the items in the table and access them the number of the row. Rows in the Arrays called Indexes.

To declare an array you use a code like that:

int my_array[50];

Here, you have declared an array of integer type, which can hold up to 50 items.

You can access each item in the array using the index of the item, like that:

My_array[10] = 500;

Here, you have set the item number 10 in the array to 500.

You can initialize the array at the same line of the declaration like that:

int my_array[5] = {1,24,15,66,500};

In our program we used this line of code:

```
double ExtMapBuffer1[];
```

Here we have declared and array of double type. We will use array to calculate our values which we will draw them on the chart.

```
int init()
{
}
```

Special functions:

Functions are blocks of code which like a machine takes inputs and returns outputs (Please review lesson 7 – Functions).

In MQL4 there are three special functions

init():

Every program will run this function before any of the other functions, you have to put here you initialization values of you variables.

start():

Here's the most of the work, every time a new quotation have received your program will call this function.

deinit():

This is the last function the program will call before it shutdown, you can put here any removals you want.

```
    SetIndexStyle(0,DRAW_LINE);
    SetIndexBuffer(0,ExtMapBuffer1);
    string short_name = "Your first indicator is running!";
    IndicatorShortName(short_name);
```

Custom indicator functions:

I can't give you a description for all of the indicators functions in this lesson. But we will use them all in our next lessons with more details. So, we will study here the functions used in our program.

SetIndexStyle:

void SetIndexStyle(int index, int type, int style=EMPTY, int width=EMPTY, color clr=CLR_NONE)

This function will set the style of the drawn line.

The **index** parameter of this function ranges from 1 to 7 (that's because the array indexing start with 0 and we have limited 8 line). And it indicte which line we want to set its style.

The **type** parameter is the shape type of the line and can be one of the following shape type's constants:

DRAW_LINE (draw a line)

DRAW_SECTION (draw section)

DRAW_HISTOGRAM (draw histogram)

DRAW_ARROW (draw arrow)

DRAW_NONE (no draw)

The **style** parameter is the pen style of drawing the line and can be one of the following styles' constants:

STYLE_SOLID (use solid pen)

STYLE_DASH (use dash pen)

STYLE_DOT (use dot pen)

STYLE_DASHDOT (use dash and dot pen)

STYLE_DASHDOTDOT (use dash and double dots)

Or it can be **EMPTY** (default) which means it will be no changes in the line style**.**

The **width** parameter is the width of line and ranges from 1 to 5. Or it can be **EMPTY** (default) which means the width will not change.

The **clr** parameter is the color of the line. It can be any valid color type variable. The default value is **CLR_NONE** which means empty state of colors.

In our line of code:

```
SetIndexStyle(0,DRAW_LINE);
```

We have set the index to 0 which means we will work with the first (and the only) line.

And we have set the shape type of our line to DRAW_LINE because we want to draw a line in the chart.

And we have left the other parameters to their default values.

SetIndexBuffer:

bool SetIndexBuffer(int index, double array[])

This function will set the array which we will assign to it our indicator value to the indicator buffer which will be drawn.

The function takes the index of the buffer where's 0 is the first buffer and 2 is the second, etc. Then it takes the name of the array.
It returns **true** if the function succeeds and **false** otherwise.
In our program the array which will hold our calculated values is `ExtMapBuffer1`.
And we have only one indicator buffer (`#property indicator_buffers 1`). So it will be the buffer assigned.

IndicatorShortName:

void IndicatorShortName(string name)

This function will set the text which will be showed on the upper left corner of the chart window (Figure 3) to the text we have inputted.
In our program we declared a string variable and assigned the value "You first indicator is running" to it, then we passed it to the **IndicatorShortName** function.

```
string short_name = "Your first indicator is running!";
IndicatorShortName(short_name);
```

Figure 3

```
return(0);
```

This is the return value of the **init()** function which terminate the function and pass the program to the execution of the next function **start().**

```
int deinit()
  {
//----

//----
   return(0);
  }
```

Nothing new to say about **deinit()** function.

We will continue with remaining of the code in the next lesson.
I hope you enjoyed the lesson and I welcome your questions.

See you
Coders' Guru
06-11-2005

MQL4 COURSE

By Coders' guru
www.forex-tsd.com

-12-
Your First Indicator
Part 3

Welcome to the third part of **"Your First Indicator"** lesson.
In the previous lesson we studied the code of our first indicator line by line and we
reached the function **dinit()**.

I hope you've came from the previous lessons with a clear idea about what we have done.

Today we are going to study **start()** function and its content. And –*finally*- we will
compile and run our first Indicator.

Are you ready? Let's hack the code line by line:

Our Code:

```
//+------------------------------------------------------------------+
//|                                              My_First_Indicator.mq4 |
//|                                                         Codersguru |
//|                                           http://www.forex-tsd.com |
//+------------------------------------------------------------------+
#property copyright "Codersguru"
#property link       "http://www.forex-tsd.com"

#property indicator_separate_window
#property indicator_buffers 1
#property indicator_color1 Red

//---- buffers
double ExtMapBuffer1[];

//+------------------------------------------------------------------+
//| Custom indicator initialization function                          |
//+------------------------------------------------------------------+
int init()
  {
//---- indicators
   SetIndexStyle(0,DRAW_LINE);
   SetIndexBuffer(0,ExtMapBuffer1);
   string short_name = "Your first indicator is running!";
   IndicatorShortName(short_name);
//----
```

```
      return(1);
   }
//+------------------------------------------------------------------+
//| Custor indicator deinitialization function                       |
//+------------------------------------------------------------------+
int deinit()
   {
//----


//----
   return(0);
   }
//+------------------------------------------------------------------+
//| Custom indicator iteration function                              |
//+------------------------------------------------------------------+
int start()
   {
   int     counted_bars=IndicatorCounted();

//---- check for possible errors
   if (counted_bars<0) return(-1);
//---- last counted bar will be recounted
   if (counted_bars>0) counted_bars--;

   int     pos=Bars-counted_bars;

   double dHigh , dLow , dResult;
   Comment("Hi! I'm here on the main chart windows!");

//---- main calculation loop
   while(pos>=0)
     {
        dHigh = High[pos];
        dLow = Low[pos];
        dResult = dHigh - dLow;
        ExtMapBuffer1[pos]= dResult ;
        pos--;
     }
//----
   return(0);
   }
//+------------------------------------------------------------------+
```

```
int start()
 {...
  return(0);
 }
```

As I told you before, we will spend 90% of programming life inside the braces of **start()**
function. That's because it's the most important MQL4 Special functions.

On the contrary of the **init()** and **deinit** function, **start()** will not be called (by the
terminal client) only one time, But every time a new quotation arrives to MetaTrader
terminal client, every time the **start()** function has been called.

start() function returns an integer value like all of the MQL4 special function, where **0** means no error and any number else means an error has been occurred.

```
int     counted_bars=IndicatorCounted();
```

Here, we have defined the variable **counted_bars** as an integer type, and we have assigned to it the returned value of the function **IndicatorCounted()**.

int IndicatorCounted()

This function will return an integer type value holds the count of the **bars** which our indicator has been calculated them.

In the first launch of your indicator this count will be **0** because the indicator didn't calculate any bars yet. And after that it will be the count of total bars on the chart **-1**. (Please see the function **Bars** below).

```
if (counted_bars<0) return(-1);
```

We have got the number of **counted_bars** in the previous line of code by using **IndicatorCounted()** function.
This number must be 0 or greater if there's no errors have been occurred. If it's less than 0 that's means we have an error and we have to terminate the **start()** function using the return statement.

```
if (counted_bars>0) counted_bars--;
```

We are checking here if the **counted_bars** are greater than 0.
If that's true we decrease this number by subtracting 1 from it.
That's because we want to recount the last bar again.

We use the decrement operator (please review Lesson 4 - Operations & Expressions) for decreasing the value of **counted_bars** by **1**.

Note: We can write the expression counted_bars-- like this:

```
counted_bars = counted_bars-1;
```

```
int     pos=Bars-counted_bars;
```

Here, we are declaring the variable **pos** to hold the number of times our calculation loop

will work (see while loop later in this lesson). That's by subtracting the **counted_bars** from the count of total bars on the chart, we get the total bars count using **Bars()** function.

It's a good time to discuss **Bars()** function and its brother.

Pre-defined MQL4 variables:

Ask, Bid, Bars, Close, Open, High, Low, Time and Volume are functions although MQL4 called them Pre-defined variables. And I'll proof to you why they are functions.

Variable means a space in memory and data type you specify.
Function means do something and return some value, For example **Bars** collects and returns the number of the bars in chart. So, is it a variable?

Another example will proof for you that they are not variables:
If you type and compile this line of code:

```
Bars=1;
```

You will get this error: ***'Bars' - unexpected token***
That's because they are not variables hence you can't assign a value to them.

Another proof, the next line of code is a valid line and will not generate and error in compiling:

```
Alert(Bars(1));
```

You can't pass parameter to a variable, parameters passed only to the functions.

I'm so sorry for the lengthiness, let's discuss every function.

int Bars

This function returns an integer type value holds count of the total bars on the current chart.

double Ask

This function (used in your Expert Advisors) returns a double type value holds the buyer's price of the currency pair.

double Bid

This function (used in your Expert Advisor) returns a double type value holds the seller's price of the currency pair.

*Note: For example, USD/JPY = 133.27/133.32 the left part is called the **bid** price (that is a price at which the trader sells), the second (the right part) is called the **ask** price (the price at which the trader buys the currency).*

double Open[]

This function returns a double type value holds the opening price of the referenced bar. Where opening price is the price at the beginning of a trade period (year, month, day, week, hour etc)
For example: Open[0] will return the opening price of the current bar.

double Close[]

This function returns a double type value holds the closing price of the referenced bar. Where closing price is the price at the end of a trade period
For example: Close[0] will return the closing price of the current bar.

double High[]

This function returns a double type value holds the highest price of the referenced bar. Where it's the highest price from prices observed during a trade period.
For example: High [0] will return the highest price of the current bar.

double Low[]

This function returns a double type value holds the lowest price of the referenced bar. Where it's the lowest price from prices observed during a trade period.
For example: Low [0] will return the lowest price of the current bar.

double Volume[]

This function returns a double type value holds the average of the total amount of currency traded within a period of time, usually one day.
For example: Volume [0] will return this average for the current bar.

int Digits

This function returns an integer value holds number of digits after the decimal point (usually 4).

double Point

This function returns a double value holds point value of the current bar (usually 0.0001.

datetime Time[]

This function returns a datetime type value holds the open time of the referenced bar.
For example: Time [0] will return the open time of the current bar.

```
double dHigh , dLow , dResult;
```

We declared three double type variables which we will use them later. Notice the way we used to declare the three of them at the same line by separating them by coma.

```
Comment("Hi! I'm here on the main chart windows!");
```

This line of code uses the **Comment** function to print the text "Hi! I'm here on the main chart windows!" on the left top corner of the main chart (figure1).
There are two similar functions:

void Comment(...)

This function takes the values passed to it (they can be any type) and print them on the left top corner of the chart (figure 1).

void Print (...)

This function takes the values passed to it (they can be any type) and print them to the expert log (figure 2).

void Alert(...)

This function takes the values passed to it (they can be any type) and display them in a dialog box (figure 3)

Figure 1 – Comment
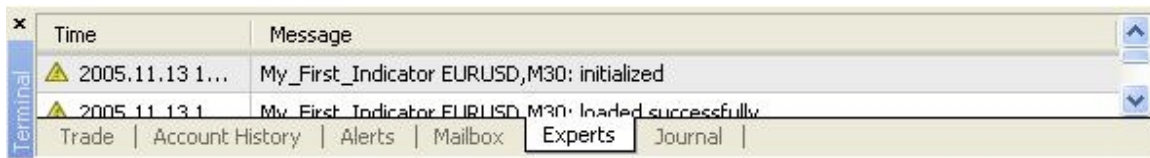
Figure 2- Expert log

Figure 3 - Alerts

```
while(pos>=0)
    {
        dHigh = High[pos];
        dLow = Low[pos];
        dResult = dHigh - dLow;
        ExtMapBuffer1[pos]= dResult ;
        pos--;
    }
```

Now, it's the time to enter the loop for calculating our indicator points to draw them.
Any value we assign to the array ExtMapBuffer1[] will be drawn on the chart (because
we have assign this array to the drawn buffer using **SetIndexBuffer** function).

Before we enter the loop we have got the number of times the loop will work by
subtracting the **counted_bars** from the total count of the bars on chart.
The number of times the loop will work called *Loop variable* which it's **pos** variable in
our example.

We use the loop variable as a current bar of the calculation for example **High[pos]** will
return the highest price of the pos bar.

In the loop body we assign to the variable **dHigh** the value of the highest price of the
current loop variable.

And assign to the variable **dLow** the value of the lowest price of the current loop variable.
The result of subtracting **dLow** from **dHigh** will be assigned to the variable **dResult**.

Then we using the **dResult** to draw or indicator line, by assigning it to the drawn buffer array **ExtMapBuffer1[]**.

The last line of the loop is a decrement expression which will decrease the loop variable pos by 1 every time the loop runs. And when this variable reaches **-1** the loop will be terminated.

Finally, we can compile our indicator. Press **F5** or choose **Compile** from file menu. That will generate the executable file "My_First_indicator.ex4" which you can load in your terminal client.

To load your indicator click **F4** to bring the terminal client. Then From the **Navigator** window find the **My_First_indicator** and attach it to the chart (figure4).

Note: The indicator will not do much, but it believed that the subtraction of the highest and lowest of the price gives us the market's volatility.

Figure 4 – My_First_Indicator

I hope you enjoyed your first indicator. And be prepared to your first Expert Advisor in the next lesson(s).

I welcome very much your questions and suggestions.

Coders' Guru
13-11-2005

MQL4 COURSE

By Coders' guru
www.forex-tsd.com

-13-
Your First Expert Advisor
Part 1

In the previous lesson we created our first indicator. Although this indicator wasn't useful for us as trader, but it was very useful for us as programmers.

The indicators –in general- are very important for the technical analysis of the market in trying to predict the future prices.
But with the indicators we observe the chart then use our hands to sell, buy and modify our orders manually. You have to set in front of your terminal, and keep your eyes widely open.

If you get tired, want to drink a cup of tea or even take a short vacation. You have to consider one of these solutions:

You may rent someone to observe the terminal for you and calling your mobile phone every five minutes to tell you what's going on. If this employee is an expert, he will cost you the pips you earn. And if he is novice one, he will cost you your capital.

The second solution is using a program to automate your trades.
That's what the Expert Advisor is for.

The Expert advisor is a program wrote in MQL4 (we are studying MQL4 huh?) uses your favorite indicators and trade methods to automate your orders.
It can buy, sell and modify the orders for you. It enables you to drink a cup of tea and save the salary you gave out to your employee or the bunch of flowers you bring to your assistant wife.

Today we are going to create our first expert advisor so let's go.

First two steps:

Step1:

If you didn't open your MetaEditor yet, I think it's the time to run it.

From the MetaEditor File menu click **New** (you can use **CTRL+N** hotkey or click the **New** icon in the standard toolbar). That will pop up the new program wizard which you have seen when you created your first indicator (Figure 1).

This time we will choose the first option **"Expert Advisor program"** then click **Next** button.
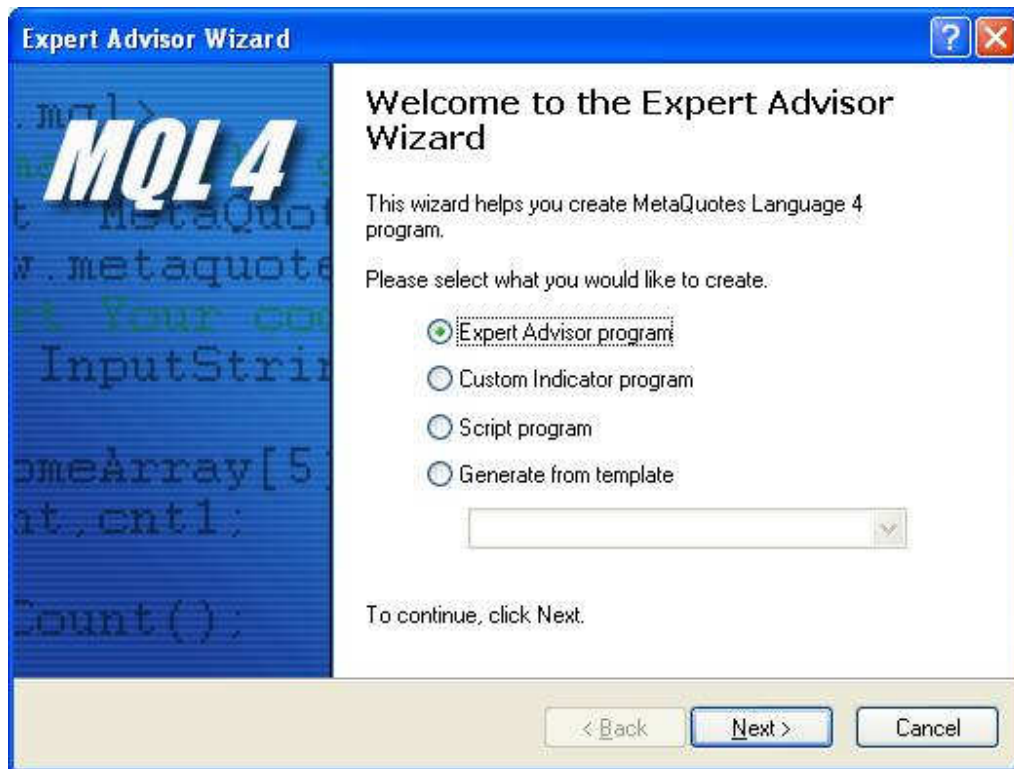
Figure 1 – the first step wizard

Step2:

When you clicked **Next**, you have got the general properties wizard (Figure 2).
This wizard enables you to set the properties of your expert advisor and to set the external variables you will use in your expert advisor.

In this step you can set these properties:

1- **Name** of your expert advisor, in our sample we will use **My_First_EA**.
2- **Author** name of the program, type your name (I typed mine in the sample).
3- **Link** to your web site.
4- **External variables list:**

This is the list of the external variables you allow the user of your expert advisor to change them from the **Expert properties** window.

To add a new variable you click the **Add** button, clicking it will add a new record to the external variables list. Every record has three fields:

Name: double click this field to set the name (identifier) of the variable.
Type: double click this field to set the data type of the variable.
Initial value: double click this field to give your variable initialization value. This field is *optional* which means you can leave it without setting

In our sample we have added three variables:

Varaible → *Type* → *initial value*

TakeProfit → double → 350
Lots → double → 0.1
TrailingStop → double → 35
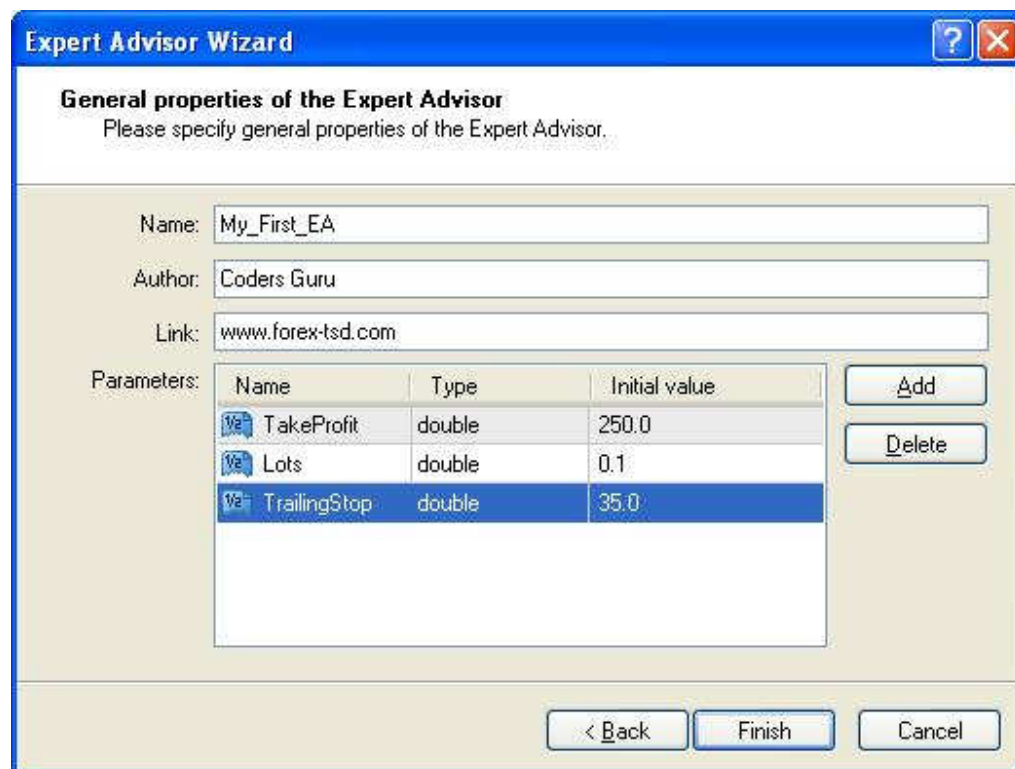
Figure 2 – the second step wizard

Now click the **Finish** button to close the wizard, and MetaEditor will bring to you the code created by the wizard and saves the file **My_First_EA.mq4** in the **MetaTrader 4 \experts** path.

*Note: you have to put the expert advisors in **MetaTrader 4\experts** path and the indicators in **MetaTrader 4\experts\indicators** path, otherwise they will not work.*

This is the code we have got from the wizard:

```mq4
//+------------------------------------------------------------------+
//|                                                  My_First_EA.mq4 |
//|                                                      Coders Guru |
//|                                            http://www.forex-tsd.com |
//+------------------------------------------------------------------+
#property copyright "Coders Guru"
#property link      "http://www.forex-tsd.com"

//---- input parameters
extern double    TakeProfit=250.0;
extern double    Lots=0.1;
extern double    TrailingStop=35.0;
//+------------------------------------------------------------------+
//| expert initialization function                                   |
//+------------------------------------------------------------------+
int init()
  {
//----

//----
   return(0);
  }
//+------------------------------------------------------------------+
//| expert deinitialization function                                 |
//+------------------------------------------------------------------+
int deinit()
  {
//----

//----
   return(0);
  }
//+------------------------------------------------------------------+
//| expert start function                                            |
//+------------------------------------------------------------------+
int start()
  {
//----

//----
   return(0);
  }
//+------------------------------------------------------------------+
```

As you see above, the code generated by the wizard is a template for you to add your code without bothering you by typing the main functions from scratch.

Now let's add our own code:

```mq4
//+------------------------------------------------------------------+
//|                                                  My_First_EA.mq4 |
```

```
//|                                                       Coders Guru |
//|                                      http://www.forex-tsd.com |
//+------------------------------------------------------------------+
#property copyright "Coders Guru"
#property link      "http://www.forex-tsd.com"

//---- input parameters
extern double     TakeProfit=250.0;
extern double     Lots=0.1;
extern double     TrailingStop=35.0;
//+------------------------------------------------------------------+
//| expert initialization function                                   |
//+------------------------------------------------------------------+
int init()
  {
//----

//----
   return(0);
  }
//+------------------------------------------------------------------+
//| expert deinitialization function                                 |
//+------------------------------------------------------------------+
int deinit()
  {
//----

//----
   return(0);
  }

int Crossed (double line1 , double line2)
   {
      static int last_direction = 0;
      static int current_dirction = 0;

      if(line1>line2)current_dirction = 1; //up
      if(line1<line2)current_dirction = 2; //down


      if(current_dirction != last_direction) //changed
      {
            last_direction = current_dirction;
            return (last_direction);
      }
      else
      {
            return (0);
      }
   }
//+------------------------------------------------------------------+
//| expert start function                                            |
//+------------------------------------------------------------------+
int start()
  {
//----
```

```
    int cnt, ticket, total;
    double shortEma, longEma;


    if(Bars<100)
      {
       Print("bars less than 100");
       return(0);
      }
    if(TakeProfit<10)
      {
       Print("TakeProfit less than 10");
       return(0);  // check TakeProfit
      }


    shortEma = iMA(NULL,0,8,0,MODE_EMA,PRICE_CLOSE,0);
    longEma = iMA(NULL,0,13,0,MODE_EMA,PRICE_CLOSE,0);

    int isCrossed  = Crossed (shortEma,longEma);

    total  = OrdersTotal();
    if(total < 1)
      {
        if(isCrossed == 1)
          {

ticket=OrderSend(Symbol(),OP_BUY,Lots,Ask,3,0,Ask+TakeProfit*Point,
"My EA",12345,0,Green);
            if(ticket>0)
              {
                if(OrderSelect(ticket,SELECT_BY_TICKET,MODE_TRADES))
Print("BUY order opened : ",OrderOpenPrice());
              }
            else Print("Error opening BUY order : ",GetLastError());
            return(0);
          }
          if(isCrossed == 2)
          {

            ticket=OrderSend(Symbol(),OP_SELL,Lots,Bid,3,0,
Bid-TakeProfit*Point,"My EA",12345,0,Red);
            if(ticket>0)
              {
                if(OrderSelect(ticket,SELECT_BY_TICKET,MODE_TRADES))
Print("SELL order opened : ",OrderOpenPrice());
              }
            else Print("Error opening SELL order : ",GetLastError());
            return(0);
          }
          return(0);
      }
    for(cnt=0;cnt<total;cnt++)
      {
        OrderSelect(cnt, SELECT_BY_POS, MODE_TRADES);
```

```
        if(OrderType()<=OP_SELL && OrderSymbol()==Symbol())
          {
           if(OrderType()==OP_BUY)   // long position is opened
             {
              // should it be closed?
              if(isCrossed == 2)
                  {
                   OrderClose(OrderTicket(),OrderLots(),Bid,3,Violet);
// close position
                   return(0); // exit
                  }
              // check for trailing stop
              if(TrailingStop>0)
                {
                 if(Bid-OrderOpenPrice()>Point*TrailingStop)
                   {
                    if(OrderStopLoss()<Bid-Point*TrailingStop)
                      {
                       OrderModify(OrderTicket(),OrderOpenPrice(),Bid-
Point*TrailingStop,OrderTakeProfit(),0,Green);
                       return(0);
                      }
                   }
                }
             }
           else // go to short position
             {
              // should it be closed?
              if(isCrossed == 1)
                {
                 OrderClose(OrderTicket(),OrderLots(),Ask,3,Violet);
// close position
                 return(0); // exit
                }
              // check for trailing stop
              if(TrailingStop>0)
                {
                 if((OrderOpenPrice()-Ask)>(Point*TrailingStop))
                   {
                    if((OrderStopLoss()>(Ask+Point*TrailingStop)) ||
(OrderStopLoss()==0))
                      {

OrderModify(OrderTicket(),OrderOpenPrice(),Ask+Point*TrailingStop,
OrderTakeProfit(),0,Red);
                       return(0);
                      }
                   }
                }
             }
          }
     }
   return(0);
  }
//+------------------------------------------------------------+
```

*Note: don't copy and paste the code above because it **warped** and will not work for you, use the code provided with lesson in www.forex-tsd.com .*

Scared?

Don't be scared of the 160 lines you have seen above, we will know everything about this code line by line, I promise it's an easy task.

Test the Expert Advisor:

Before studying our expert advisor code we have to be check is it profitable one or not.

*Note: Our expert advisor will work with **EURUSD** pairs in **4 Hours** timeframe.*
*So compile the expert advisor by pressing **F5** and load it in MetaTrader.*

You can test your expert advisor using two methods:

1- Live trading

In live trading testing the results are more accurate but you have to spend days (and maybe months) to know is the expert advisor profitable or not.

You have to enable the expert advisor to automate your trades.
To enable it click **Tools → Option** menu (or use **CTRL+O** hotkey), that's will bring the **Options** windows (figure 3), click **Expert Advisors** tab and enable these options:

Enable Expert Advisors
Allow live trading

And click Ok button
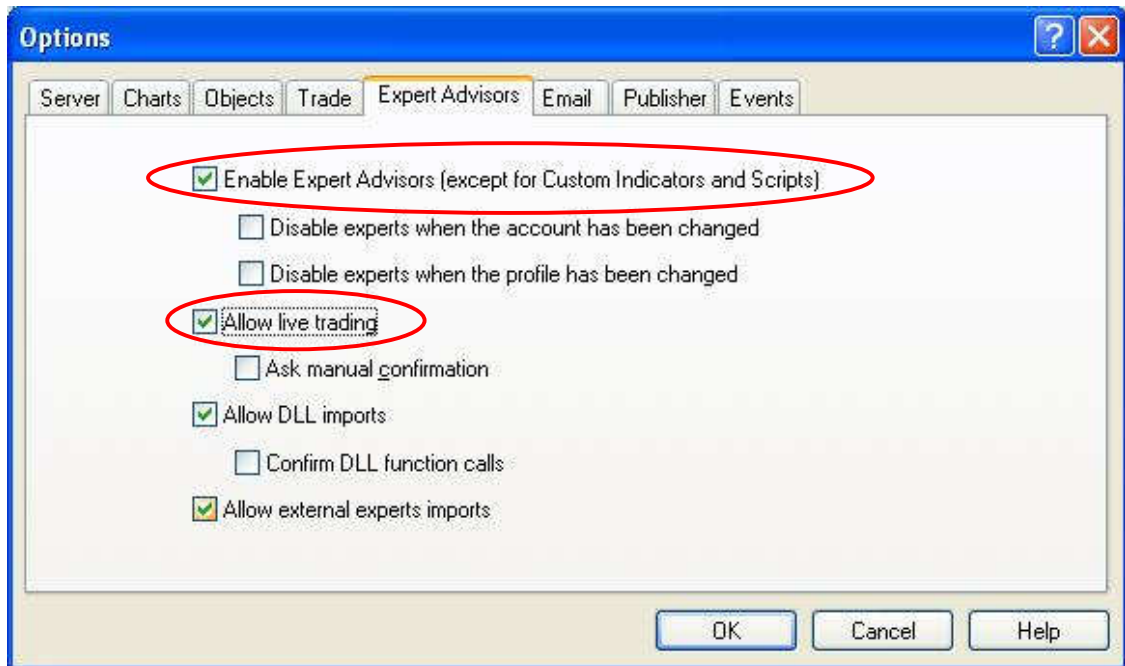
Figure 3 – Enabling expert advisor auto trade

You will see the **Smile** symbol beside the expert advisor name which means your expert advisor is working and ready to trade for you (Figure 4).

Figure 4 – Expert advisor is enabled

2- Strategy Tester:

The second method of testing your expert advisor which is less accurate but will not take time is the Strategy tester. We will know everything about Strategy tester later, let's now bring its window by pressing **F6** (Figure 5).
When you get the window enter these options:

Symbol: EURUSD.
Period: H4 (4 Hours).
Model: Open price only.
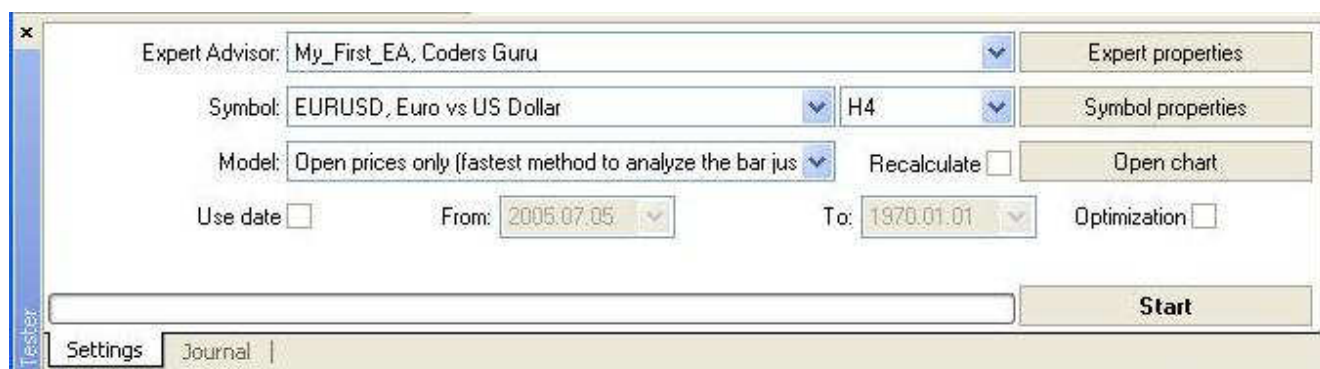
Figure 5 – Strategy tester window

Now click **Start** button to start testing the expert advisor.

You will see a progress bar and the two tabs (Settings and Journal) became five tabs.
We interested in **Report** tab (Figure 6); click it to see your profit.

We have a lot to say and to do in the next lesson; I hope you are ready for the challenge.
I welcome very much the questions and the suggestions.

See you
Coders' Guru
24-11-2005

MQL4 COURSE

By Coders' guru
www.forex-tsd.com

-14-
Your First Expert Advisor
Part 2

Welcome to the second part of creating Your First Expert Advisor lesson.
In the previous part we have taken the code generated by the *new program wizard* and
added our own code which we are going to crack it today line by line.

Did you wear your *coding gloves*? Let's crack.

*No**te**: I have to repeat that our expert advisor is for educational purpose only and will
not (or aimed to) make profits.*

The code we have:

```
//+------------------------------------------------------------------+
//|                                                 My_First_EA.mq4 |
//|                                                      Coders Guru |
//|                                            http://www.forex-tsd.com |
//+------------------------------------------------------------------+
#property copyright "Coders Guru"
#property link      "http://www.forex-tsd.com"

//---- input parameters
extern double    TakeProfit=250.0;
extern double    Lots=0.1;
extern double    TrailingStop=35.0;
//+------------------------------------------------------------------+
//| expert initialization function                                   |
//+------------------------------------------------------------------+
int init()
  {
//----


//----
   return(0);
  }
//+------------------------------------------------------------------+
//| expert deinitialization function                                 |
//+------------------------------------------------------------------+
int deinit()
  {
//----
```

```
//----
   return(0);
   }

int Crossed (double line1 , double line2)
   {
       static int last_direction = 0;
       static int current_direction = 0;

       if(line1>line2)current_direction = 1; //up
       if(line1<line2)current_direction = 2; //down



       if(current_direction != last_direction) //changed
       {
           last_direction = current_direction;
           return (last_direction);
       }
       else
       {
           return (0);
       }
   }
//+------------------------------------------------------------------+
//| expert start function                                            |
//+------------------------------------------------------------------+
int start()
   {
//----


   int cnt, ticket, total;
   double shortEma, longEma;


   if(Bars<100)
     {
      Print("bars less than 100");
      return(0);
     }
   if(TakeProfit<10)
     {
      Print("TakeProfit less than 10");
      return(0);  // check TakeProfit
     }


   shortEma = iMA(NULL,0,8,0,MODE_EMA,PRICE_CLOSE,0);
   longEma = iMA(NULL,0,13,0,MODE_EMA,PRICE_CLOSE,0);

   int isCrossed  = Crossed (shortEma,longEma);

   total  = OrdersTotal();
   if(total < 1)
     {
        if(isCrossed == 1)
```

```mql4
            {
ticket=OrderSend(Symbol(),OP_BUY,Lots,Ask,3,0,Ask+TakeProfit*Point,
"My EA",12345,0,Green);
            if(ticket>0)
              {
                if(OrderSelect(ticket,SELECT_BY_TICKET,MODE_TRADES))
Print("BUY order opened : ",OrderOpenPrice());
              }
            else Print("Error opening BUY order : ",GetLastError());
            return(0);
          }
          if(isCrossed == 2)
            {

            ticket=OrderSend(Symbol(),OP_SELL,Lots,Bid,3,0,
Bid-TakeProfit*Point,"My EA",12345,0,Red);
            if(ticket>0)
              {
                if(OrderSelect(ticket,SELECT_BY_TICKET,MODE_TRADES))
Print("SELL order opened : ",OrderOpenPrice());
              }
            else Print("Error opening SELL order : ",GetLastError());
            return(0);
          }
          return(0);
      }
   for(cnt=0;cnt<total;cnt++)
     {
      OrderSelect(cnt, SELECT_BY_POS, MODE_TRADES);
      if(OrderType()<=OP_SELL && OrderSymbol()==Symbol())
        {
        if(OrderType()==OP_BUY)   // long position is opened
          {
           // should it be closed?
           if(isCrossed == 2)
                {
                 OrderClose(OrderTicket(),OrderLots(),Bid,3,Violet);
// close position
                 return(0); // exit
                }
           // check for trailing stop
           if(TrailingStop>0)
             {
              if(Bid-OrderOpenPrice()>Point*TrailingStop)
                {
                 if(OrderStopLoss()<Bid-Point*TrailingStop)
                   {
                    OrderModify(OrderTicket(),OrderOpenPrice(),Bid-
Point*TrailingStop,OrderTakeProfit(),0,Green);
                    return(0);
                   }
                }
             }
          }
        else // go to short position
          {
```

```
            // should it be closed?
            if(isCrossed == 1)
              {
               OrderClose(OrderTicket(),OrderLots(),Ask,3,Violet);
// close position
               return(0); // exit
              }
            // check for trailing stop
            if(TrailingStop>0)
              {
               if((OrderOpenPrice()-Ask)>(Point*TrailingStop))
                 {
                  if((OrderStopLoss()>(Ask+Point*TrailingStop)) ||
(OrderStopLoss()==0))
                    {

OrderModify(OrderTicket(),OrderOpenPrice(),Ask+Point*TrailingStop,
OrderTakeProfit(),0,Red);
                     return(0);
                    }
                 }
              }
           }
        }
     }
   return(0);
  }
//+------------------------------------------------------------+
```

The idea behind our expert advisor.

Before digging into cracking our code we have to explain the idea behind our expert advisor. Any expert advisor has to decide when to **enter** the market and when to **exit**. And the idea behind any expert advisor is what the entering and exiting **conditions** are? Our expert advisor is a simple one and the idea behind it is a simple too, let's see it.

We use two EMA indicators, the first one is the EMA of 8 days (sort EMA) and the second one is the EMA of 13 days (long EMA).

Note: using those EMAs or any thought in this lesson is not a recommendation of mine, they are for educational purpose only.

Entering (Open):

Our expert advisor will enter the market when the **short EMA** line crosses the **long EMA** line, the direction of each lines will determine the order type:
If the **short EMA** is **above** the **long EMA** will **buy** (long).
If the **short EMA** is **below** the **long EMA** we will **sell** (short).

We will open only **one** order at the same time.

Exiting (Close):

Our expert advisor will close the **buy** order when the **short EMA** crosses the **long EMA** and the **short EMA** is **below** the **long EMA**.
And will close the **sell** order when the **short EMA** crosses the **long EMA** and the **short EMA** is **above** the **long EMA**.

Our order (buy or sell) will automatically be closed too when the **Take profit** or the **Stop loss** points are reached.

Modifying:

Beside entering (opening) and exiting (closing) the market (positions) our expert advisor has the ability to modify existing positions based on the idea of **Trailing stop** point.
We will know how we implemented the idea of Trialing stop later in this lesson.

Now let's resume our code cracking.

```
//---- input parameters
extern double     TakeProfit=250.0;
extern double     Lots=0.1;
extern double     TrailingStop=35.0;
```

In the above lines we have asked the wizard to declare three **external** variables (which the user can set them from the properties window of our expert advisor).
The three variables are double data type. We have initialized them to default values (the user can change these values from the properties window, but it recommended to leave the defaults).

I have to pause again to tell you a little story about those variables.

Stop loss:

It's a limit point you set to your order when reached the order will be closed, this is useful to minimize your lose when the market going against you.
Stop losses points are always set below the current asking price on a buy or above the current bid price on a sell.

Trailing Stop

It's a kind of stop loss order that is set at a percentage level below (for a long position) or above (for a short position) the market price. The price is adjusted as the price fluctuates.
We will talk about this very important concept later in this lesson.

Take profit:

It's similar to stop lose in that it's a limit point you set to your order when reached the order will be closed

There are, however, two differences:

- There is no "trailing" point.
- The exit point must be set above the current market price, instead of below.
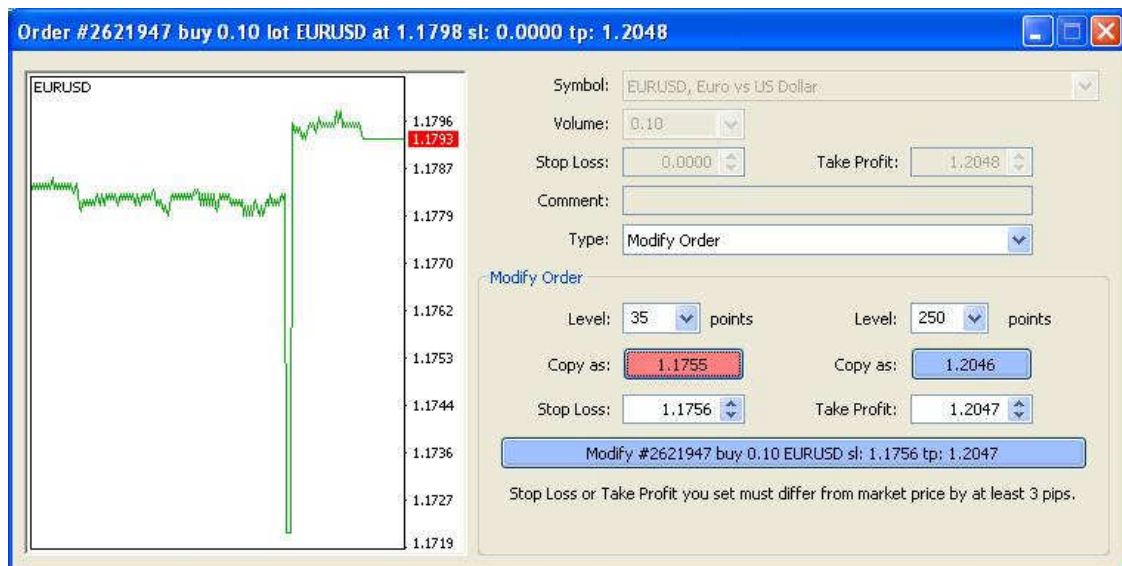
Figure 1 – Setting Stop loss and Take profit points

```
int Crossed (double line1 , double line2)
   {
      static int last_direction = 0;
      static int current_direction = 0;

      if(line1>line2)current_direction = 1; //up
      if(line1<line2)current_direction = 2; //down



      if(current_direction != last_direction) //changed
      {
            last_direction = current_direction;
            return (last_direction);
      }
      else
      {
            return (0);
      }

   }
```

As I told you before, the idea behind our expert advisor is monitor the crossing of the short EMA and the long EMA lines. And getting the direction of the crossing (which line is above and which line is below) which will determine the type of the order (buy, sell, buy-close and sell-close).

For this goal we have created the *Crossed* function.

The *Crossed* function takes two double values as parameters and returns an integer. The first parameter is the value of the first line we want to monitor (the short EMA in our case) and the second parameter is the value of the second we want to (the long EMA).

The function will monitor the two lines every time we **call** it by saving the direction of the two lines in static variables to remember their state between the repeated calls.

- It will return **0** if there's no change happened in the last directions saved.
- It will return **1** if the direction has changed (the lines crossed each others) and the first line is above the second line.
- It will return **2** if the direction has changed (the lines crossed each others) and the first line is below the second line.

Note: You can use this function in your coming expert advisor to monitor any two lines and get the crossing direction.

Let's see how did we write it?

```
int Crossed (double line1 , double line2)
```

The above line is the function declaration, it means we want to create *Crossed* function which takes two double data type **parameters** and **returns** an integer.

When you **call** this function you have to pass to it two double parameters and it will return an integer to you.

You have to declare the function before using (calling) it. The place of the function doesn't matter, I placed it above **start()** function, but you can place it anywhere else.

```
static int last_direction = 0;
static int current_direction = 0;
```

Here we declared two static integers to hold the current and the last direction of the two lines. We are going to use these variables (they are static variables which means they save their value between the repeated calls) to check if there's a change in the direction of the lines or not.

we have initialized them to **0** because we don't want them to work in the first call to the function (if they worked in the first call the expert advisor will open an order as soon as we load it in the terminal).

```
if(current_direction != last_direction) //changed
```

In this line we compare the two static variables to check for changes between the last call of our function and the current call.
If *last_direction* not equal *current_direction* that's mean there's a change happened in the direction.

```
last_direction = current_direction;
return (last_direction);
```

In this case (*last_direction* not equal *current_direction*) we have to reset our *last_direction* by assigning to it the value of the *current_direction*.
And we will return the value of the *last_direction*. This value will be **1** if the first line is above the second line and **2** if the first line is below the second line.

```
else
   {
      return (0);
   }
```

Else (*last_direction* is equal *current_direction*) there's no change in the lines direction and we have to return **0**.

Our program will call this function in its **start()** function body and use the returned value to determine the appropriate action.

In the coming part of this lesson we will know how did we call the function, and we will know a lot about the very important trading functions.
To that day I hope you the best luck.

I welcome very much the questions and the suggestions.

See you
Coders' Guru
01-12-2005

-15-
Your First Expert Advisor
Part 3

In the previous two parts of this lesson, we have introduced our expert advisor and knew the idea behind it.
And in the *Appendix 2* we have studied the *Trading Functions* which we will use some of them today in our expert advisor.

Today we will continue cracking the remaining code of the expert advisor.
I hope you have cleared your mind for our discovering mission.

The code we have:

```
//+------------------------------------------------------------------+
//|                                                My_First_EA.mq4 |
//|                                                    Coders Guru |
//|                                        http://www.forex-tsd.com |
//+------------------------------------------------------------------+
#property copyright "Coders Guru"
#property link      "http://www.forex-tsd.com"

//---- input parameters
extern double    TakeProfit=250.0;
extern double    Lots=0.1;
extern double    TrailingStop=35.0;
//+------------------------------------------------------------------+
//| expert initialization function                                   |
//+------------------------------------------------------------------+
int init()
  {
//----


//----
   return(0);
  }
//+------------------------------------------------------------------+
//| expert deinitialization function                                 |
//+------------------------------------------------------------------+
int deinit()
  {
//----


//----
   return(0);
  }

int Crossed (double line1 , double line2)
```

```
    {
        static int last_direction = 0;
        static int current_direction = 0;

        if(line1>line2)current_direction = 1; //up
        if(line1<line2)current_direction = 2; //down



        if(current_direction != last_direction) //changed
        {
            last_direction = current_direction;
            return (last_direction);
        }
        else
        {
            return (0);
        }
    }
//+------------------------------------------------------------------+
//| expert start function                                            |
//+------------------------------------------------------------------+
int start()
  {
//----


    int cnt, ticket, total;
    double shortEma, longEma;


    if(Bars<100)
      {
        Print("bars less than 100");
        return(0);
      }
    if(TakeProfit<10)
      {
        Print("TakeProfit less than 10");
        return(0);  // check TakeProfit
      }


    shortEma = iMA(NULL,0,8,0,MODE_EMA,PRICE_CLOSE,0);
    longEma = iMA(NULL,0,13,0,MODE_EMA,PRICE_CLOSE,0);

    int isCrossed  = Crossed (shortEma,longEma);

    total  = OrdersTotal();
    if(total < 1)
      {
        if(isCrossed == 1)
          {

ticket=OrderSend(Symbol(),OP_BUY,Lots,Ask,3,0,Ask+TakeProfit*Point,
"My EA",12345,0,Green);
            if(ticket>0)
              {
                if(OrderSelect(ticket,SELECT_BY_TICKET,MODE_TRADES))
Print("BUY order opened : ",OrderOpenPrice());
              }
```

```
              else Print("Error opening BUY order : ",GetLastError());
              return(0);
           }
           if(isCrossed == 2)
           {

              ticket=OrderSend(Symbol(),OP_SELL,Lots,Bid,3,0,
Bid-TakeProfit*Point,"My EA",12345,0,Red);
              if(ticket>0)
                 {
                  if(OrderSelect(ticket,SELECT_BY_TICKET,MODE_TRADES))
Print("SELL order opened : ",OrderOpenPrice());
                 }
              else Print("Error opening SELL order : ",GetLastError());
              return(0);
           }
           return(0);
        }
     for(cnt=0;cnt<total;cnt++)
        {
         OrderSelect(cnt, SELECT_BY_POS, MODE_TRADES);
         if(OrderType()<=OP_SELL && OrderSymbol()==Symbol())
            {
            if(OrderType()==OP_BUY)   // long position is opened
               {
               // should it be closed?
               if(isCrossed == 2)
                   {
                    OrderClose(OrderTicket(),OrderLots(),Bid,3,Violet);
// close position
                    return(0); // exit
                   }
               // check for trailing stop
               if(TrailingStop>0)
                  {
                  if(Bid-OrderOpenPrice()>Point*TrailingStop)
                     {
                      if(OrderStopLoss()<Bid-Point*TrailingStop)
                        {
                         OrderModify(OrderTicket(),OrderOpenPrice(),Bid-
Point*TrailingStop,OrderTakeProfit(),0,Green);
                         return(0);
                        }
                     }
                  }
               }
            else // go to short position
               {
               // should it be closed?
               if(isCrossed == 1)
                   {
                    OrderClose(OrderTicket(),OrderLots(),Ask,3,Violet);
// close position
                    return(0); // exit
                   }
               // check for trailing stop
               if(TrailingStop>0)
                  {
                  if((OrderOpenPrice()-Ask)>(Point*TrailingStop))
                     {
```

```
                          if((OrderStopLoss()>(Ask+Point*TrailingStop)) ||
(OrderStopLoss()==0))
                          {

OrderModify(OrderTicket(),OrderOpenPrice(),Ask+Point*TrailingStop,
OrderTakeProfit(),0,Red);
                              return(0);
                          }
                      }
                  }
              }
          }
      }
    return(0);
   }
//+-------------------------------------------------------------+
```

```
int cnt, ticket, total;
```

In this line we declared three *integer* type variables.
We used one line to declare the three of them because they are the same type (You can't use a single line declaration with dissimilar data types).

Note: To declare multi variables in a single line you start the line with the declaration **keyword** which indicates the type of the variables then separate the variables **identifiers** (names) with **comma**.
You can divide the above line into three lines like that
int cnt;
int ticket;
int total;

We will use the *cnt* variable as a counter in our "opened orders checking loop".
We will use the *ticket* variable to hold the ticket number returned by *OrderSend* function.
And we will use the *total* variable to hold the number of already opened orders.

```
double shortEma, longEma;
```

Again, we used a single line to declare two *double* variables.
We will use these variables to hold the value of short EMA and long EMA.
As you remember (I hope) from the previous part we uses the *Crossing* of short and long EMAs as buying and selling conditions and as closing conditions too.

```
if(Bars<100)
   {
     Print("bars less than 100");
     return(0);
   }
```

We want to work with a normal chart and we assume that the normal chart must have more than 100 bars. That's because the insufficient numbers of bars will not enable or EMA indicators to work right.

We have got the number of bars on the chart using the *Bars* function and checked this number to find is it less than 100 or not. If it's less than 100 we will do two things:
We will tell the user what's the wrong by writing this message to the Experts log *"bars less than 100"* (*Figure 1*).
Then we will terminate the *start* function by the line *retrun(0);*

That's the way we refuse to work with less than 100 bars on the chart.

Figure 1 – Experts log

```
if(TakeProfit<10)
    {
     Print("TakeProfit less than 10");
     return(0);  // check TakeProfit
    }
```

We don't want too to work with insufficient value of *TakeProfit*.
The *TakeProfit* variable is an external variable and that's mean the user can change its default value from the expert advisor properties windows.
We want to protect the user of our expert advisor from his bad choices.
We assume that any value less than 10 for the *TakeProfi variable* will be bad choice and for that we have checked the *TakeProfit* value the user set to find is it less than 10 or not.

If it's less than 10 we will inform the user what's the wrong by printing him the message *"TakeProfit less than 10"*, and we will terminate the *start* function by *return(0)* statement.

```
shortEma = iMA(NULL,0,8,0,MODE_EMA,PRICE_CLOSE,0);
longEma = iMA(NULL,0,13,0,MODE_EMA,PRICE_CLOSE,0);
```

Well, everything is OK, the bars on the chart were more than 100 bars and the value of *TakeProfit* the user supplied was more than 10.

We want now to calculate the short and long EMAs of the current bar.
We use the MQL4 built-in technical indicator function **iMA** which calculates the moving average indicator.
I have to pause here for a couple of minutes to tell you more details about **iMA** function.

iMA:

Syntax:

double **iMA**(string **symbol**, int **timeframe**, int **period**, int **ma_shift**, int **ma_method**, int **applied_price**, int **shift**)

Description:

The *iMA* function calculates the moving average indicator and returns its value (double data type).

Note: A moving average is an average price of a certain currency over a certain time interval (in days, hours, minutes etc) during an observation period divided by these time intervals.

Parameters:

This function takes **7** parameters:

string **symbol**:
The symbol name of the currency pair you trading (Ex: EURUSD and USDJPY).
If you want to use the current symbol use NULL as a parameter.

int **timeframe:**
The time frame you want to use for the moving average calculation.
You can use one of these time frame values:

Constant	Value	Description
PERIOD_M1	1	1 minute.
PERIOD_M5	5	5 minutes.
PERIOD_M15	15	15 minutes.
PERIOD_M30	30	30 minutes.
PERIOD_H1	60	1 hour.
PERIOD_H4	240	4 hour.
PERIOD_D1	1440	Daily.
PERIOD_W1	10080	Weekly.
PERIOD_MN1	43200	Monthly.
0 (zero)	0	Time frame used on the chart.

If you want to use the current timeframe, use **0** as a parameter.

Note: You can use the integer representation of the period or its constant name.
For example, the line:
*iMA(NULL, **PERIOD_H4**,8,0,MODE_EMA,PRICE_CLOSE,0);*
is equal to
*iMA(NULL,**240**,8,0,MODE_EMA,PRICE_CLOSE,0);*
But it's recommended to use the constant name to make your code clearer.

int **period:**
The number of days you want to use for the moving average calculation.

int **ma_shift:**
The number of the bars you want to shift the moving average line relative to beginning of the chart:
0 means no shift (*Figure 2*)
A positive value shifts the line to right (*Figure 3*)
A negative value shifts the line to left (*Figure 4*)

int **ma_method**:
The moving average method you want to use for the moving average calculation,
It can be one of these values:

Constant	Value	Description
MODE_SMA	0	Simple moving average.
MODE_EMA	1	Exponential moving average.
MODE_SMMA	2	Smoothed moving average.
MODE_LWMA	3	Linear weighted moving average.

int **applied_price**:
The price you want to use for the moving average calculation,
It can be one of these values:

Constant	Value	Description
PRICE_CLOSE	0	Close price.
PRICE_OPEN	1	Open price.
PRICE_HIGH	2	High price.
PRICE_LOW	3	Low price.
PRICE_MEDIAN	4	Median price, (high+low)/2.
PRICE_TYPICAL	5	Typical price, (high+low+close)/3.
PRICE_WEIGHTED	6	Weighted close price, (high+low+close+close)/4.

int **shift**:
The number of bars (relative to the current bar) you want to use for the moving average calculation. Use **0** for the current bar.

Figure 2 : ma_ shift = 0

Figure 3: ma_shift = 10

Figure 4: ma_shift = -10

```
shortEma = iMA(NULL,0,8,0,MODE_EMA,PRICE_CLOSE,0);
longEma = iMA(NULL,0,13,0,MODE_EMA,PRICE_CLOSE,0);
```

Now you know what the above lines means.

We assigned to the variable *shortEma* the value of:
8 days closing price based exponential moving average of the current bar.
Which we can briefly call it 8EMA

And assigned to the variable *longEma* the value of:
13 days closing price based exponential moving average of the current bar.
Which we can briefly call it 13EMA

```
int isCrossed  = Crossed (shortEma,longEma);
```

Note: The *Crossed* function takes two double values as parameters and returns an integer. The first parameter is the value of the first line we want to monitor (the short EMA in our case) and the second parameter is the value of the second we want to (the long EMA). The function will monitor the two lines every time we call it by saving the direction of the two lines in static variables to remember their state between the repeated calls.

It will return **0** if there's no change happened in the last directions saved.

It will return **1** if the direction has changed (the lines crossed each others) and the first line is above the second line.

It will return **2** if the direction has changed (the lines crossed each others) and the first line is below the second line.

Here we declared an integer variable *isCrossed* to hold the return value of the function *Corssed*. We will use this value to open and close orders.

```
total  = OrdersTotal();
if(total < 1)
    {
        .......
    }
```

We assigned the *OrdersTotal* return value to the variable *total*.

Note: The *OrdersTotal* function returns the number of opened and pending orders. If this number is **0** that means there are no orders (market or pending ones) has been opened. Please review appendix 2

Then we checked this number (total) to find if there was already opened orders or not. The *if block* of code will work only if the total value is lesser than 1 which means there's no already opened order.

```
if(isCrossed == 1)
        {

ticket=OrderSend(Symbol(),OP_BUY,Lots,Ask,3,0,Ask+TakeProfit*Point,
"My EA",12345,0,Green);
            if(ticket>0)
              {
               if(OrderSelect(ticket,SELECT_BY_TICKET,MODE_TRADES))
Print("BUY order opened : ",OrderOpenPrice());
              }
            else Print("Error opening BUY order : ",GetLastError());
            return(0);
        }
```

In the case the *shortEma* crossed the *longEma* and the *shortEma* is above the *longEma* we will buy now.
We are using the *OrderSend* function to open a buy order.

Note: The *OrderSend* function used to open a sell/buy order or to set a pending order. It returns the ticket number of the order if succeeded and **-1** in failure.

<u>Syntax</u>:
int OrderSend(string **symbol**, int **cmd**, double **volume**, double **price**, int **slippage**, double **stoploss**, double **takeprofit**, string **comment**=NULL, int **magic**=0, datetime **expiration**=0, color **arrow_color**=CLR_NONE)

 Please review appendix 2

These are the parameters we used with our *OrderSend* function:

symbol:
We used the *Sybmol* function to get the symbol name of the current currency and passed it to the *OrderSend* function.

cmd:
We used *OP_BUY* because we want to open a Buy position.

volume:
We used the *Lots* value that the use has been supplied.

price:
We used the *Ask* function to get the current ask price and passed it to the OrderSend function.

slippage:
We used **3** for the *slippage* value.

stoploss:
We used **0** for *stoploss* which mean there's no *stoploss* for that order.

takeprofit:
We multiplied the *TakeProfit* value the user has been supplied by the return value of the *Point* function and added the result to the *Ask* price.

Note: *Point* function returns the point size of the current currency symbol.
For example: if you trade EURUSD the point value = .0001 and if you trade EURJPY the point value should be .01
So, you have to convert your *stoploss* and *takeprofit* values to points before using them with *OrderSend* or *OrderModify* functions.

---

comment:
We used *"My EA"* string for the comment

magic:
We used the number *12345* for the order *magic* number.

expiration:
We didn't set an *expiration* date to our order, so we used **0**.

arrow_color:
We set the color of opening arrow to *Green* (Because we like the money and the money is green)

The *OrderSend* function will return the ticket number of the order if succeeded, so we check it with this line:

```
if(ticket>0)
```

We used the *OrderSelect* function to select the order by its ticket number that's before we used the *OrderOpenPrice* function which returns the open price of the selected order.

Everything is OK, the *OrderSend* returned a proper ticket number (greater than 0) and the *OrderSelect* successfully selected the order. It's the good time to tell the user this good new by printing to him the message *"BUY order opened : "* plus the opened price of the order.

Note: For more details about *OrderSelect* and *OrderOpenPrice*, please review appendix 2

Else, if the *OrderSend* function returned **-1** which means there was an error opening the order, we have to tell the user this bad news by printing him the message: *"Error opening BUY order : "*, plus the error number returned by the function *GetLastError*.
And in this case we have to terminate the start function by using *return(0)*.

```
if(isCrossed == 2)
        {
            ticket=OrderSend(Symbol(),OP_SELL,Lots,Bid,3,0,
Bid-TakeProfit*Point,"My EA",12345,0,Red);
            if(ticket>0)
              {
               if(OrderSelect(ticket,SELECT_BY_TICKET,MODE_TRADES))
Print("SELL order opened : ",OrderOpenPrice());
              }
            else Print("Error opening SELL order : ",GetLastError());
            return(0);
        }
```

Here, the opposite scenario, the *shortEma* crossed the *longEma* and the *shortEma* is **below** the *longEma* we will sell now.

We use the *OrderSend* function to open a Sell order. Can you guess what the differences between the Buy parameters and Sell parameters in *OrderSend* are? Right! You deserve 100 pips as a gift.

These parameters haven't been changed:

symbol is the same**.**
volume is the same**.**
slippage is the same.
stoploss is the same.
comment is the same.
magic is the same.
expiration is the same.

These parameters have been changed (that's why you've got the gift):

cmd:
We used *OP_SELL* because we want to open a Sell position.

price:
We used the *Bid* function to get the current bid price and passed it to the *OrderSend* function.

takeprofit:
We multiplied the *TakeProfit* value the user has been supplied by the return value of the *Point* function and **subtracted** the result from the *Bid* price.

arrow_color:
We set the color of opening arrow to *Red* (Because we like the money and the money is green but we want a different color for selling position).

We can briefly repeat what will happen after calling the *OrderSend* with the above parameters:

The *OrderSend* returns the ticket number if succeeded.
We check this number to find is it greater than **0** which means no errors.
We used the *OrderSelect* to select the order before using *OrderOpenPrice*.
Everything is OK, we tell the user this good news by printing to him the message *"Sell order opened : "* plus the opened price of the order.

Else, if the *OrderSend* function returned **-1** we tell the user this bad news by printing him the message: *"Error opening SELL order : "*, plus the error number and terminate the *start* function by using *return(0).*

- Wait a minute! (You said).
- I've noticed that there's a line after the last *if* block of code you have just explained above.
- Where? (I'm saying).
- Here it's:

```
return(0);
```

Yes! Great but I have no pips to gift you this time.
Look carefully to these blocks (braces):

```
if(total < 1)
    {
      if(isCrossed == 1)
        {
         .....
        }
        if(isCrossed == 2)
        {

          .....
        }
          return(0);  ← Right!
    }
```

If the *shorEma* crossed the *longEma* upward we open Buy order.
If the *shorEma* crossed the *longEma* downward we open Sell order.

What if they haven't been crossed yet? That's the job of this line.
We terminate the start function if there was no crossing.
(In the case that *isCrossed* not equal 1 or 2).

Now we are ready to open Buy and Sell positions.

In the coming part we will crack the remaining of the code and will discuss in details the
MetaTrader Strategy Tester.

I hope you enjoyed the lesson.

I welcome very much your questions and suggestions.

Coders' Guru
24-12-2005

-16-
Your First Expert Advisor
Part 4

We have reached the edge of the moon in our way to the truth, I'm not under the impact of alcoholic poisoning (I don't drink at all), but I'm so happy to reach the last part of explaining or first expert advisor. **Yes**! This is the last part of the expert advisor lesson.

I hope you enjoyed the journey discovering how to write our simple yet important expert advisor.
Let's take the final step.

The code we have:

```
//+------------------------------------------------------------------+
//|                                                 My_First_EA.mq4 |
//|                                                     Coders Guru |
//|                                            http://www.forex-tsd.com |
//+------------------------------------------------------------------+
#property copyright "Coders Guru"
#property link      "http://www.forex-tsd.com"

//---- input parameters
extern double    TakeProfit=250.0;
extern double    Lots=0.1;
extern double    TrailingStop=35.0;
//+------------------------------------------------------------------+
//| expert initialization function                                   |
//+------------------------------------------------------------------+
int init()
  {
//----

//----
   return(0);
  }
//+------------------------------------------------------------------+
//| expert deinitialization function                                 |
//+------------------------------------------------------------------+
int deinit()
  {
//----

//----
   return(0);
  }
```

```
int Crossed (double line1 , double line2)
   {
      static int last_direction = 0;
      static int current_direction = 0;

      if(line1>line2)current_direction = 1; //up
      if(line1<line2)current_direction = 2; //down


      if(current_direction != last_direction) //changed
      {
            last_direction = current_direction;
            return (last_direction);
      }
      else
      {
            return (0);
      }
   }
//+------------------------------------------------------------------+
//| expert start function                                            |
//+------------------------------------------------------------------+
int start()
  {
//----


    int cnt, ticket, total;
    double shortEma, longEma;


    if(Bars<100)
      {
       Print("bars less than 100");
       return(0);
      }
    if(TakeProfit<10)
      {
       Print("TakeProfit less than 10");
       return(0);  // check TakeProfit
      }


    shortEma = iMA(NULL,0,8,0,MODE_EMA,PRICE_CLOSE,0);
    longEma = iMA(NULL,0,13,0,MODE_EMA,PRICE_CLOSE,0);

    int isCrossed  = Crossed (shortEma,longEma);

    total  = OrdersTotal();
    if(total < 1)
      {
       if(isCrossed == 1)
         {

ticket=OrderSend(Symbol(),OP_BUY,Lots,Ask,3,0,Ask+TakeProfit*Point,
"My EA",12345,0,Green);
            if(ticket>0)
              {
               if(OrderSelect(ticket,SELECT_BY_TICKET,MODE_TRADES))
Print("BUY order opened : ",OrderOpenPrice());
```

```
            }
         else Print("Error opening BUY order : ",GetLastError());
         return(0);
      }
      if(isCrossed == 2)
      {

         ticket=OrderSend(Symbol(),OP_SELL,Lots,Bid,3,0,
Bid-TakeProfit*Point,"My EA",12345,0,Red);
         if(ticket>0)
            {
             if(OrderSelect(ticket,SELECT_BY_TICKET,MODE_TRADES))
Print("SELL order opened : ",OrderOpenPrice());
            }
         else Print("Error opening SELL order : ",GetLastError());
         return(0);
      }
      return(0);
   }
for(cnt=0;cnt<total;cnt++)
  {
   OrderSelect(cnt, SELECT_BY_POS, MODE_TRADES);
   if(OrderType()<=OP_SELL && OrderSymbol()==Symbol())
     {
      if(OrderType()==OP_BUY)   // long position is opened
        {
         // should it be closed?
         if(isCrossed == 2)
            {
             OrderClose(OrderTicket(),OrderLots(),Bid,3,Violet);
// close position
             return(0); // exit
            }
         // check for trailing stop
         if(TrailingStop>0)
           {
            if(Bid-OrderOpenPrice()>Point*TrailingStop)
              {
               if(OrderStopLoss()<Bid-Point*TrailingStop)
                 {
                  OrderModify(OrderTicket(),OrderOpenPrice(),Bid-
Point*TrailingStop,OrderTakeProfit(),0,Green);
                  return(0);
                 }
              }
           }
        }
      else // go to short position
        {
         // should it be closed?
         if(isCrossed == 1)
            {
             OrderClose(OrderTicket(),OrderLots(),Ask,3,Violet);
// close position
             return(0); // exit
            }
         // check for trailing stop
         if(TrailingStop>0)
           {
            if((OrderOpenPrice()-Ask)>(Point*TrailingStop))
              {
```

```
                    if((OrderStopLoss()>(Ask+Point*TrailingStop)) ||
(OrderStopLoss()==0))
                        {

OrderModify(OrderTicket(),OrderOpenPrice(),Ask+Point*TrailingStop,
OrderTakeProfit(),0,Red);
                            return(0);
                        }
                    }
                }
            }
        }
    }
    return(0);
  }
//+------------------------------------------------------------------+
```

In the previous lesson, we checked the *OrdersTotal* is less than **1** in order to open a **Buy** or a **Sell** orders in the case that there were no already opened orders.
We have used this code:

```
if(total < 1)
    {
       if(isCrossed == 1)
         {
          .....
         }
       if(isCrossed == 2)
         {

           .....
         }
          return(0);
    }
```

This was the *Open New Order routine*. Today we will study *Modify-Close Opened Orders routine*.

```
for(cnt=0;cnt<total;cnt++)
    {
     ....
    }
```

In the above block of code we used a *for* loop to go through all the already opened orders. We start the loop from the *cnt* = **0** and the end of the loop is the *total* number of already orders. Every loop cycle we increase the number of *cnt* by **1** (*cnt++*).So, *cnt* will hold in every cycle the poison of the order (0,1,2,3 etc) which we will use with *OrderSelect* function to select each order by its position.

Our today's mission is studying what's going inside the heart of the above loop.

```
OrderSelect(cnt, SELECT_BY_POS, MODE_TRADES);
      if(OrderType()<=OP_SELL && OrderSymbol()==Symbol())
        {
         ....
        }
```

The *OrderSelect* function used to select an opened order or a pending order by the ticket number or by index.

We used the *OrderSelect* here before using the *OrderType* and *OrderSymbol* functions because if we didn't use *OrderSelect,* the *OrderType* and *OrderSymbol* functions will not work.

Note: You have to use *OrderSelect* function before the trading functions which takes no parameters:
OrderMagicNumber, OrderClosePrice, OrderCloseTime, OrderOpenPrice, OrderOpenTime, OrderComment, OrderCommission, OrderExpiration, OrderLots, OrderPrint, OrderProfit, OrderStopLoss, OrderSwap, OrderSymbol, OrderTakeProfit, OrderTicket and OrderType

---

We used *SELECT_BY_POS* selecting type which means we want to select the order by its index (position) not by its ticket number.

Note: The index of the first order is **0** and the index of the second one is **1** index etc.

And we used *MODE_TRADES* mode which means we will select from the currently trading orders (opened and pending orders) not from the history.

The *OrderType* function returns the type of selected order that will be one of:
OP_BUY, OP_SELL, OP_BUYLIMIT, OP_BUYSTOP, OP_SELLLIMIT or
OP_SELLSTOP

We checked the type of the order to find is it *equal* or *lesser* than *OP_SELL.*
Which means it maybe one of two cases: *OP_SELL* or *OP_BUY* (because *OP_SELL*=1 and *OP_BUY* = 0). We did that because we will not work with pending orders.

We want too to work only with the order opened in the chart we loaded our expert advisor on, so we check the *OrderSymbol* of the order with the return value of *Symbol* function which returns the current chart symbol. If they are equal it means we are working with the currently loaded symbol.

So, all the coming code will work only if the *OrderType* is *OP_SELL* or *OP_BUY* and the *Symbol = OrderSymbol.*

```
if(OrderType()==OP_BUY)   // long position is opened
   {
    ....
   }
```

We are working only with two types of orders, the first type is *OP_BUY.*
The code above means:

Is there a long (Buy) position opened? If yes! Execute this block of code....
Let's see what will do in the case of a long position has been opened

```
if(isCrossed == 2)
   {
      OrderClose(OrderTicket(),OrderLots(),Bid,3,Violet);
      // close position
      return(0); // exit
   }
```

We have opened a **Buy** order when the *shortEma* crossed the *longEma* **upward**.
It's a logical to close this position when the *shortEma* and *longEma* crosses each others in reversal direction (**downward**).

So, we checked the *isCrossed* to find is it = **2** which means the reversal has been occurred and in this case we close the **Buy** order.

We used the *OrderClose* function to close the order. *OrderClose* function closes a specific opened order by its ticket. (Review appendix 2).

We've got the ticket number of the selected order using the *OrderTicket* function and passed it as the first parameter for *OrderClose* function.
The second parameter in the *OrderClose* is *Lots* (the number of lots); we used the *OrderLots* function to get the lots value of the selected order.
The third parameter in *OrderClose* is the preferred close price and we used the *Bid* function to get the bid price of the selected order.
The fourth parameter is the *slippage* value and we used **3**.
The fifth parameter is the color of the closing arrow and we used *Violet* color.

We didn't forget to terminate the *start* function with *return(0)* statement.

```
// check for trailing stop
         if(TrailingStop>0)
           {
            if(Bid-OrderOpenPrice()>Point*TrailingStop)
              {
               if(OrderStopLoss()<Bid-Point*TrailingStop)
                 {
                  OrderModify(OrderTicket(),OrderOpenPrice(),Bid-
Point*TrailingStop,OrderTakeProfit(),0,Green);
                  return(0);
                 }
              }
           }
```

Note: We are still inside the block of: *if(OrderType()==OP_BUY)*.

We are going to apply our trailing stop technique for the opened **Buy** position in this block of code.

Firstly, we have checked the *TrailingStop* variable the user supplied to check was it a valid value or not (greater than **0**).

Then we applied our trailing stop technique for the opened **Buy** orders which is:

> We **modify** the *stoploos* of the order when the **subtraction** of the current **bid** price and the **opened price** of order is **greater** than the *TrailingStop*
> **and**
> the current *stoploss* is **lesser** than the **subtraction** of the current **bid** price and the *TrailingStop*.

We used the *OrderModify* function to make the desired modification.
These are parameters we used with *OrderModify*:

ticket: We've got the current order ticket with *OrderTicket* function.

price: We've got the open price of the order with *OrderOpenPrice* function.

stoploss: Here's the real work! Because we are in a **Buy** position we set our new *stoploss* to the value of the **subtraction** of the current **bid** price and the *TrailingStop*.
That's our way trailing the *stoploss* point every time we make profits.

> **Note**: Stop losses are always set **BELOW** the current bid price on a **buy** and **ABOVE** the current asking price on a **sell**.

takeprofit: No changes, we've got the current profit value of the order with *OrderTakeProfit* function.

expiration: We didn't set an *expiration* date to our order, so we used **0**.

arrow_color: Still *Green* color.

Finally we terminate the start function.

```
else // go to short position
    {
     ....
    }
```

> **Note**: *else* here belongs to the code:
>
> ```
> if(OrderType()==OP_BUY)   // long position is opened
>     {
>      ....
>     }
> ```

We have studied the case of the type of the order is a **Buy** order.
So, we are working now a **Sell** order type.
Let's see what are we going to do in the case of a short (Sell) position has been already opened?

```
if(isCrossed == 1)
   {
```

```
         OrderClose(OrderTicket(),OrderLots(),Ask,3,Violet);
         // close position
         return(0); // exit
   }
```

We've opened a **Sell** order when the *shortEma* crossed the *longEma* **downward**.
It's the time to close this position when the *shortEma* and *longEma* crosses each others in
reversal direction (**upward**). Which happens in the case of *isCrossed* = **1**.

We used the *OrderClose* function to close the order, we used the same parameters we use
in the case of closing a **Buy** order except the third parameters the preferred close price, in
this case is the **Ask** price.
Then we terminated the *start* function.

```
// check for trailing stop
           if(TrailingStop>0)
             {
              if((OrderOpenPrice()-Ask)>Point*TrailingStop)
                {
                 if((OrderStopLoss()>(Ask+Point*TrailingStop)) ||
(OrderStopLoss()==0))
                    {
OrderModify(OrderTicket(),OrderOpenPrice(),Ask+Point*TrailingStop,
OrderTakeProfit(),0,Red);
                     return(0);
                    }
                }
             }
```

We are going to apply our trailing stop technique for the opened **Sell** position in this
block of code.

Firstly, we've checked the *TrailingStop* variable the user supplied to check was it a valid
value or not (greater than **0**).
Then we applied our trailing stop technique for the opened **Sell** orders which is:

> We **modify** the *stoploss* of the order when the **subtraction** of the order's **opened price**
> and the current **ask** price is **greater** than the *TrailingStop*
> **and**
> the current *stoploss* is **greater** than the **addition** of the current **ask** price and the
> *TrailingStop*.

We used the *OrderModify* function to make the desired modification. And used the same
parameters we use in the case of modifying already opened Buy order except the third
parameter which indicates our *stoploss* value:
We set our new *stoploss* to the value of the **addition** of the current **ask** price and the
TrailingStop.

And the fifth parameter which indicates the color of the arrow in this case is *Red*.

Then we terminated the *start* function using *return(0)*;

```
return(0);
```

This line terminates the *start* function in all other case; there are no conditions to open new positions and there are no needs to close or modify the already opened orders. Just don't forget it.

I hope you enjoyed the lesson.
I welcome very much your questions and suggestions.

Coders' Guru
28-12-2005

MQL4 COURSE

By Coders' guru

(Appendix 1)

Bars

I have got a lot of questions about the mystery of the bars count.
I'm going to describe everything about Bars in this appendix.

What's a BAR?

The bar is the dawning unit on the chart which you can specify by choosing the period of the timeframe.

For example: The 30 minutes timeframe will draw a bar every 30 minutes.

MetaTrader (and other platforms) uses the values of to high, low, open and close prices to draw the bar start and end boundaries. (Figure 1)
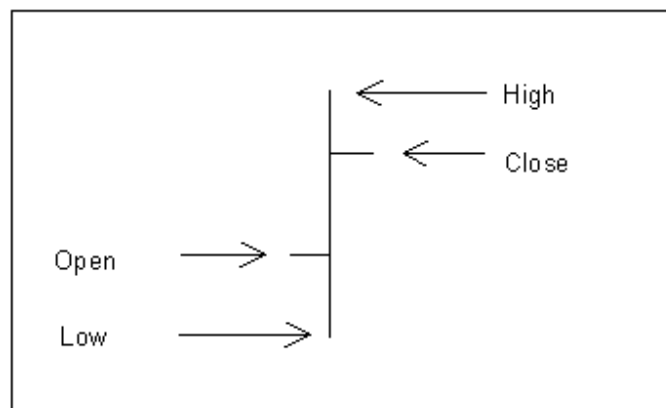
Figure 1 - Bar chart dawning

How the bars indexed in MQL4?

MQL4 indexes the bars from **0** (for the current bar) then 1, 2, 3 etc to the count of the bars.
So, if you want to work with the current bar you use the index **0**.
And the index of the previous bar (of the current bar) is **1**.
And the index 100 is the bar 100 in the history (100 bars ago).
And the index of last bar is the count of all bars on chart.

MQL4 BARS count functions:

In MQL4 there is a variety of functions working with the count of bars:

int Bars

This function returns the number of the bars of the current chart.

Note that, changing the timeframe will change this count.

int iBars(string symbol, int timeframe)

This function returns the number of bars on the specified currency pairs symbol and timeframe.
Assume you are working on 30M timeframe and you want to get the count of **Bars** on 1H time frame, you use this line:

```
iBars(NULL, PERIOD_H1));
```

Note: If you used it like this:

```
iBars(NULL,0));
```

*It returns the same number as **Bars** function.*

int IndicatorCounted()

When you are writing your indicator, you know now how to get the number of bars.
You use this number in your calculation and line dawning.

But it's useful to know if you have counted the bar before or it's the first time you count it.
That's because, if you have counted it before you don't want to count it again and want to work only with the new bars.

In this case you use the function **IndicatorCounted(),** which returns the number of bars have been counted by your indicator.

At the first run of your indicator this count will be zero, that's because your indicator didn't count any bars yet.
Afterwards, it will equal to the count of Bars – 1. That's because the last bar not counted yet (Figure 2).

Figure 2 – Program output

Let's write a small program to show you what's going on.

```
//+------------------------------------------------------------------+
//|                                                        Bars.mq4 |
//|                                                      Codersguru |
//|                                          http://www.forex-tsd.com |
//+------------------------------------------------------------------+
#property copyright "Codersguru"
#property link      "http://www.forex-tsd.com"
#property indicator_chart_window
//+------------------------------------------------------------------+
//| Custom indicator initialization function                        |
//+------------------------------------------------------------------+
int init()
   {
//---- indicators

//----
   return(1);
   }
//+------------------------------------------------------------------+
//| Custor indicator deinitialization function                      |
//+------------------------------------------------------------------+
int deinit()
   {
//----

//----
   return(0);
   }
//+------------------------------------------------------------------+
//| Custom indicator iteration function                             |
//+------------------------------------------------------------------+
int start()
   {
//----
   Alert( "counted_bars = " , IndicatorCounted()); //The count of bars
have been counted
   Alert( "Bars = " , Bars); //The count of the bars on the chart
   Alert( "iBars = " , iBars(NULL,0)); //the same as Bars function
   Alert( "First Bar Open = " , Open[Bars-1]); //Open price of the
first bar
```

```
   Alert( "Last Bar Open = " , Open[0]); //Open price of the last bar
(current bar)
//----
   return(0);
   }
//+------------------------------------------------------------------+
```

Note: This program produces the image you have seen in figure 2

I hope the Bars count is clear now.
I welcome very much the questions and the suggestions.

See you
Coders' Guru
21-11-2005

MQL4 COURSE

By Coders' guru
www.forex-tsd.com

(Appendix 2)
Trading Functions

In this appendix you will find the description of the **25** MQL4 trading functions.
I decided to write this appendix before writing the third part of "**Your First Expert Advisor**" lesson because you have to know these important functions before cracking the remaining of the code.

OrderSend:

Syntax:

> int **OrderSend**(string **symbol**, int **cmd**, double **volume**, double **price**, int **slippage**, double **stoploss**, double **takeprofit**, string **comment**=NULL, int **magic**=0, datetime **expiration**=0, color **arrow_color**=CLR_NONE)

Description:

The *OrderSend* function used to open a sell/buy order or to set a pending order.
It returns the ticket number of the order if succeeded and **-1** in failure.
Use *GetLastError* function to get more details about the error.

> **Note**: The ticket number is a unique number returned by *OrderSend* function which you can use later as a reference of the opened or pending order (for example you can use the ticket number with *OrderClose* function to close that specific order).

> **Note**: *GetLastError* function returns a predefined number of the last error occurred after an operation (for example when you call *GetLastError* after *OrderSend* operation you will get the error number occurred while executing *OrderSend*).
>
> Calling *GetLastError* will reset the last error number to 0.
>
> You can find a full list of MQL4 errors numbers in *stderror.mqh* file.
> And you can get the error description for a specific error number by using *ErrorDescription* function which defined at *stdlib.mqh* file.

Parameters:

This function takes **11** parameters:

string **symbol**:
The symbol name of the currency pair you trading (Ex: EURUSD and USDJPY).

> **Note**: Use *Symbol()* function to get currently used symbol and *OrderSymbol* function to get the symbol of current selected order.

int **cmd**:
An integer number indicates the type of the operation you want to take; it can be one of these values:

Constant	Value	Description
OP_BUY	0	Buying position.
OP_SELL	1	Selling position.
OP_BUYLIMIT	2	Buy limit pending position.
OP_SELLLIMIT	3	Sell limit pending position.
OP_BUYSTOP	4	Buy stop pending position.
OP_SELLSTOP	5	Sell stop pending position.

> **Note:** You can use the integer representation of the value or the constant name.
> For example:
> *OrderSend(Symbol(),0,…)* is equal to *OrderSend(Symbol(),OP_BUY,…)* .
> But it's recommended to use the constant name to make your code clearer.

double **volume**:
The number of lots you want to trade.

double **price**:
The price you want to open the order at.
Use the functions *Bid* and *Ask* to get the current bid or ask price.

int **slippage**:
The slippage value you assign to the order.

> **Note**: *slippage* is the difference between estimated transaction costs and the amount actually paid.
> *slippage* is usually attributed to a change in the spread. (*Investopedia.com*).

double **stoploss**:
The price you want to close the order at in the case of losing.

double **takeprofit**:
The price you want to close the order at in the case of making profit.

string **comment**:

The comment string you want to assign to your order (*Figure 1*).

The default value is *NULL* which means there's no comment assigned to the order.

Note: Default value of a parameter means you can leave (don't write) it out, and MQL4 will use a predefined value for this parameter.

For example we can write *OrderSend* function with or without *comment* parameter like this:

*OrderSend(Symbol(),OP_BUY,Lots,Ask,3,Ask-25*Point,Ask+25*Point,**"My order comment"**,12345,0,Green);*

Or like this:

OrderSend(Symbol(),OP_BUY,Lots,Ask,3,Ask-25*Point,Ask+25*Point,12345,0,Green);

int **magic**:

The magic number you assign to the order.

Note: Magic number is a number you assign to your order(s) as a reference enables you to distinguish between the different orders. For example the orders you have opened by your expert advisor and the orders have opened manually by the user.

Figure 1 - Comment

datetime **expiration**:

The time you want your pending order to expire at.

The default time is **0** which means there's no exportation.

Note: The time here is the server time not your local time, to get the current server time use *CurTime* function and to get the local time use *LocalTime* function.

color **arrow_color**:

The color of opening arrow (*Figure 2*), the default value is *CLR_NONE* which means there's no arrow will be drawn on the chart.
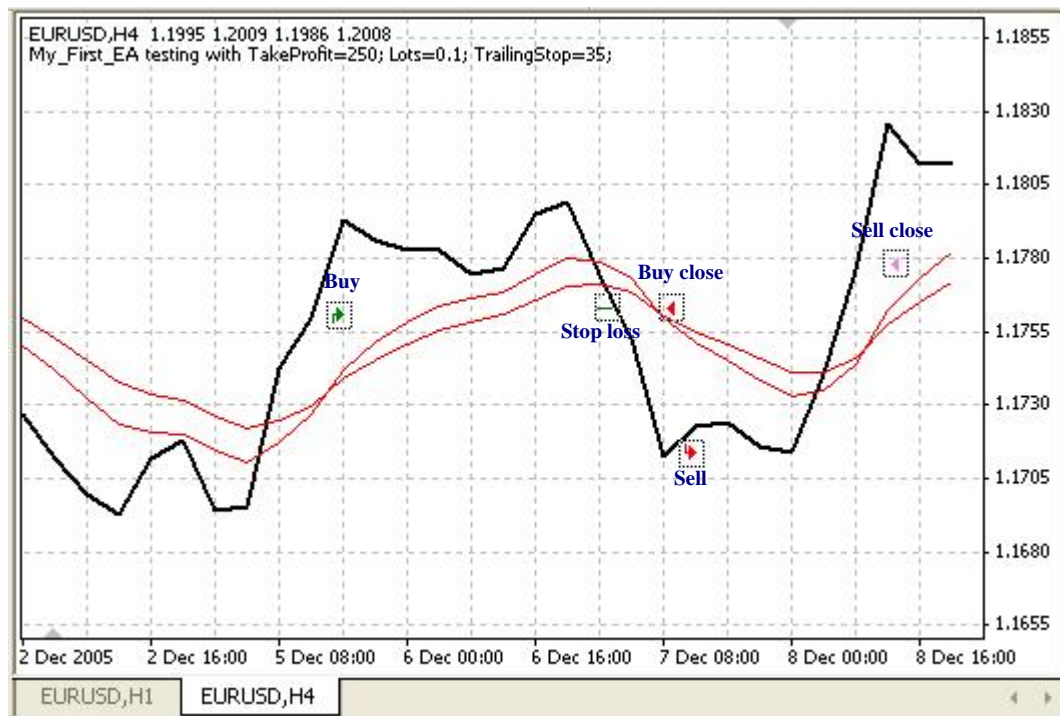
Figure 2 – Arrows color

Example:

```
int ticket;
if(iRSI(NULL,0,14,PRICE_CLOSE,0)<25)
  {
   ticket=OrderSend(Symbol(),OP_BUY,1,Ask,3,Ask-25*Point,Ask+25*Point,"My
order #2",16384,0,Green);
   if(ticket<0)
     {
      Print("OrderSend failed with error #",GetLastError());
      return(0);
     }
  }
```

OrderModify:

Syntax:

```
bool OrderModify( int ticket, double price, double stoploss, double takeprofit,
datetime expiration, color arrow_color=CLR_NONE)
```

Description:

The *OrderModify* function used to modify the properties of a specific opened order or pending order with the new values you pass to the function.
It returns true if the order successfully modified and false in failure.
Use *GetLastError* function to get more details about the error.

Parameters:

This function takes **6** parameters:

int **ticket**:
The ticket number of the order you want to modify.

> Note: This number has been assigned to the order by the function *OrderSend*.
> You can use the function *OrderTicket* to get the ticket number of the current order.

double **price**:
The price you want to set for the order.

> **Note**: Use the function *OrderOpenPrice* to get the open price for the current order.

double **stoploss**:
The price you want to close the order at in the case of losing.

double **takeprofit**:
The price you want to close the order at in the case of making profit.

> Note: We usually use the *OrderModify* function to change the *stoploss* and/or *takeprofit* values and that called *Trailing*.

datetime **expiration**:
The time you want your pending order to expire at.
Use **0** if you don't want to set an expiration time.

color **arrow_color**:
The color of the arrow, the default value is *CLR_NONE* which means there's no arrow will be drawn on the chart.

Example:

```
if(TrailingStop>0)
   {
   SelectOrder(12345,SELECT_BY_TICKET);
   if(Bid-OrderOpenPrice()>Point*TrailingStop)
    {
     if(OrderStopLoss()<Bid-Point*TrailingStop)
       {
```

```
        OrderModify(OrderTicket(),Ask-10*Point,Ask-
35*Point,OrderTakeProfit(),0,Blue);
        return(0);
        }
    }
  }
```

OrderClose:

Syntax:

bool **OrderClose**(int **ticket**, double **lots**, double **price**, int **slippage**,
color **Color**=CLR_NONE)

Description:

The *OrderClose* function used to close a specific opened order (by its ticket).
It returns true if the order successfully closed and false in failure.
Use *GetLastError* function to get more details about the error.

Parameters:

This function takes **5** parameters:

int **ticket**:
The ticket number of the order you want to close.

double **lots**:
The number of lots you use in the order.

Note: Use the *OrderLots* function to get the lots value of the current order.

double **price**:
The price you want to open the order at.
Use the functions **Bid** and **Ask** to get the current bid or ask price.

int **slippage**:
The *slippage* value of the order.

color **Color**:
The color of closing arrow, the default value is *CLR_NONE* which means there's no
arrow will be drawn on the chart.

```
if(iRSI(NULL,0,14,PRICE_CLOSE,0)>75)
   {
    OrderClose(order_id,1,Ask,3,Red);
   return(0);
   }
```

OrderSelect:

Syntax:

bool **OrderSelect**(int **index**, int **select**, int **pool**=MODE_TRADES)

Description:

The *OrderSelect* function used to select an opened order or a pending order by the ticket number or by index.
It returns true if the order successfully selected and false in failure.
Use *GetLastError* function to get more details about the error.

Note: You have to use *OrderSelect* function before the trading functions which takes no parameters:
OrderMagicNumber, OrderClosePrice, OrderCloseTime, OrderOpenPrice, OrderOpenTime, OrderComment, OrderCommission, OrderExpiration, OrderLots, OrderPrint, OrderProfit, OrderStopLoss, OrderSwap, OrderSymbol, OrderTakeProfit, OrderTicket and OrderType

Parameters:

This function takes **3** parameters:

int **index**:
The index or the ticket number of the order you want to select. It depends on the second parameter (selecting type).

int **select**:
The type of selecting operation (by index or by ticket number).
It can be one of two values:
SELECT_BY_POS: use the position (index) of the order.
SELECT_BY_TICKET – use the ticket number of the order.

int **pool**:
If you used the *SELECT_BY_POS* selecting type, you have to determine which pool (data) you will select from:

MODE_TRADES: select from the currently trading orders (opened and pending orders). This is the default value.
MODE_HISTORY: select from the history (closed and canceled orders).

Example:

```
if(OrderSelect(12470, SELECT_BY_TICKET)==true)
   {
    Print("order #12470 open price is ", OrderOpenPrice());
    Print("order #12470 close price is ", OrderClosePrice());
   }
  else
    Print("OrderSelect failed error code is",GetLastError());
```

OrderDelete:

Syntax:

```
bool OrderDelete( int ticket)
```

Description:

The *OrderDelete* function used to delete a pending order.
It returns true if the order successfully deleted and false in failure.
Use *GetLastError* function to get more details about the error.

Parameters:

This function takes only **1** parameter:

int **ticket**:
The ticket number of the order you want to delete.

Example:

```
if(Ask>var1)
   {
    OrderDelete(order_ticket);
    return(0);
   }
```

OrderCloseBy:

Syntax:

bool **OrderCloseBy**(int **ticket**, int **opposite**, color **Color**=CLR_NONE)

Description:

The *OrderCloseBy* function used to close a specific opened order by opening an opposite direction order.
It returns true if the order successfully closed and false in failure.
Use *GetLastError* function to get more details about the error.

Parameters:

This function takes **3** parameters:
int **ticket**:
The ticket number of the order you want to close.

int **opposite**:
The ticket number of the order you want to open in the opposite direction.

color **Color**:
The color of closing arrow, the default value is CLR_NONE which means no arrow will be drawn on the chart.

Example:

```
if(iRSI(NULL,0,14,PRICE_CLOSE,0)>75)
    {
     OrderCloseBy(order_id,opposite_id);
     return(0);
    }
```

OrderType:

Syntax:

int **OrderType**( )

Description:

The *OrderType* function returns the type of selected order that will be one of:
OP_BUY, OP_SELL, OP_BUYLIMIT, OP_BUYSTOP, OP_SELLLIMIT or
OP_SELLSTOP (see *OrderSend* function)

The order must be selected by *OrderSelect* before calling *OrderType.*

Parameters:

This function doesn't take any parameters and returns an integer date type (the type of selected order).

Example:

```
int order_type;
  if(OrderSelect(12, SELECT_BY_POS)==true)
    {
     order_type=OrderType();
     // ...
    }
  else
    Print("OrderSelect() returned error - ",GetLastError());
```

HistoryTotal:

Syntax:

```
int HistoryTotal( )
```

Description:

The *HistoryTotal* function searches the account history loaded in the terminal and returns the number of closed orders.

Note: We usually use this function with the ***OrderSelect*** function to get information about a specific order in the history.

Parameters:

This function doesn't take any parameters and returns an integer (the number of closed orders in the history).
Use *GetLastError* function to get more details about the error.

Example:

```
// retrieving info from trade history
  int i,hstTotal=HistoryTotal();
  for(i=0;i<hstTotal;i++)
    {
     //---- check selection result
```

```
  if(OrderSelect(i,SELECT_BY_POS,MODE_HISTORY)==false)
    {
     Print("Access to history failed with error (",GetLastError(),")");
     break;
    }
  // some work with order
  }
```

OrderClosePrice:

Syntax:

```
double OrderClosePrice( )
```

Description:

The *OrderClosePrice* function returns the close price of selected order.
The order must be selected by *OrderSelect* before calling *OrderClosePrice.*

Parameters:

This function doesn't take any parameters and returns a double data type (the close price of the selected order).

Example:

```
if(OrderSelect(ticket,SELECT_BY_POS)==true)
   Print("Close price for the order ",ticket," = ",OrderClosePrice());
 else
   Print("OrderSelect failed error code is",GetLastError());
```

OrderCloseTime:

Syntax:

```
datetime OrderCloseTime( )
```

Description:

The *OrderCloseTime* function returns the close time of the selected order.
If the return value is **0** that means the order hasn't been closed yet otherwise it has been closed and retrieved from the history.
The order must be selected by *OrderSelect* before calling *OrderCloseTime.*

This function doesn't take any parameters and returns a datetime data type (the close time of the selected order).

Example:

```
if(OrderSelect(10,SELECT_BY_POS,MODE_HISTORY)==true)
   {
    datetime ctm=OrderOpenTime();
    if(ctm>0) Print("Open time for the order 10 ", ctm);
    ctm=OrderCloseTime();
    if(ctm>0) Print("Close time for the order 10 ", ctm);
   }
  else
    Print("OrderSelect failed error code is",GetLastError());
```

OrderComment:

Syntax:

```
string OrderComment( )
```

Description:

The *OrderCloseTime* function returns the comment string for the selected order.

Note: This comment has been assigned when you opened the order with *OrderSend* or has been assigned by the server. Sometimes the server append its comment at the end of you comment string.

The order must be selected by *OrderSelect* before calling *OrderCloseTime.*

Parameters:

This function doesn't take any parameters and returns a string data type (the comment string of the selected order).

Example:

```
string comment;
  if(OrderSelect(10,SELECT_BY_TICKET)==false)
   {
    Print("OrderSelect failed error code is",GetLastError());
```

```
   return(0);
   }
 comment = OrderComment();
 // ...
```

OrderCommission:

Syntax:

```
double OrderCommission( )
```

Description:

The *OrderCommission* function returns the commission amount of the selected order.
The order must be selected by *OrderSelect* before calling *OrderCommission*.

Parameters:

This function doesn't take any parameters and returns a double data type (the commission amount of the selected order).

Example:

```
if(OrderSelect(10,SELECT_BY_POS)==true)
   Print("Commission for the order 10 ",OrderCommission());
 else
   Print("OrderSelect failed error code is",GetLastError());
```

OrderExpiration:

Syntax:

```
datetime OrderExpiration( )
```

Description:

The *OrderExpiration* function returns the expiration time of the selected pending order that you have set in *OrderSend.*
The order must be selected by *OrderSelect* before calling *OrderExpiration.*

This function doesn't take any parameters and returns a datetime data type (the expiration time of the selected pending order).

Example:

```
if(OrderSelect(10, SELECT_BY_TICKET)==true)
   Print("Order expiration for the order #10 is ",OrderExpiration());
 else
   Print("OrderSelect failed error code is",GetLastError());
```

OrderLots:

Syntax:

```
double OrderLots( )
```

Description:

The *OrderLots* function returns the lots value of the selected order that you have set in *OrderSend* (*volume* parameter).
The order must be selected by *OrderSelect* before calling *OrderLots*.

Parameters:

This function doesn't take any parameters and returns a datetime data type (the lots value of the selected order).

Example:

```
if(OrderSelect(10,SELECT_BY_POS)==true)
   Print("lots for the order 10 ",OrderLots());
 else
   Print("OrderSelect failed error code is",GetLastError());
```

OrderMagicNumber:

Syntax:

```
int OrderMagicNumber( )
```

The *OrderMagicNumber* function returns the magic number of the selected order that you have set in *OrderSend*.
The order must be selected by *OrderSelect* before calling *OrderMagicNumber*.

Parameters:

This function doesn't take any parameters and returns an integer data type (the magic number of the selected order).

Example:

```
if(OrderSelect(10,SELECT_BY_POS)==true)
   Print("Magic number for the order 10 ", OrderMagicNumber());
 else
   Print("OrderSelect failed error code is",GetLastError());
```

OrderOpenPrice:

Syntax:

```
double OrderOpenPrice( )
```

Description:

The *OrderOpenPrice* function returns the open price of the selected order.
The order must be selected by *OrderSelect* before calling *OrderOpenPrice*.

Parameters:

This function doesn't take any parameters and returns a double data type (the open price of the selected order).

Example:

```
if(OrderSelect(10, SELECT_BY_POS)==true)
   Print("open price for the order 10 ",OrderOpenPrice());
 else
   Print("OrderSelect failed error code is",GetLastError());
```

OrderOpenTime:

Syntax:

datetime **OrderOpenTime**( )

Description:

The *OrderOpenTime* function returns the open time of the selected order.
The order must be selected by *OrderSelect* before calling *OrderOpenTime.*

Parameters:

This function doesn't take any parameters and returns a datetime data type (the open time of the selected order).

Example:

```
if(OrderSelect(10, SELECT_BY_POS)==true)
   Print("open time for the order 10 ",OrderOpenTime());
 else
   Print("OrderSelect failed error code is",GetLastError());
```

OrderPrint:

Syntax:

void **OrderPrint**( )

Description:

The *OrderPrint* function prints the selected order data to the expert log file.
The order must be selected by *OrderSelect* before calling *OrderPrint.*

Parameters:

This function doesn't take any parameters and doesn't return any value (void).

Note: *void* means the function doesn't return any value, so, you can't assign it to a variable like this:
int i = OrderPrint(); //no meaning line, although the compiler will not complain.

```
if(OrderSelect(10, SELECT_BY_TICKET)==true)
   OrderPrint();
 else
   Print("OrderSelect failed error code is",GetLastError());
```

OrderProfit:

Syntax:

```
double OrderProfit( )
```

Description:

The *OrderProfit* function returns the profit of the selected order.
The order must be selected by *OrderSelect* before calling *OrderProfit.*

Parameters:

This function doesn't take any parameters and returns a double data type (the profit of the selected order).

Example:

```
if(OrderSelect(10, SELECT_BY_POS)==true)
   Print("Profit for the order 10 ",OrderProfit());
 else
   Print("OrderSelect failed error code is",GetLastError());
```

OrderStopLoss:

Syntax:

```
double OrderStopLoss( )
```

Description:

The *OrderStopLoss* function returns the *stoploss* price of the selected order that you have set in *OrderSend* or modified with *OrderModify.*
The order must be selected by *OrderSelect* before calling *OrderStopLoss.*

Parameters:

This function doesn't take any parameters and returns a double data type (the *stoploss* price of the selected order).

Example:

```
if(OrderSelect(ticket,SELECT_BY_POS)==true)
   Print("Stop loss value for the order 10 ", OrderStopLoss());
 else
   Print("OrderSelect failed error code is",GetLastError());
```

OrdersTotal:

Syntax:

```
int OrdersTotal( )
```

Description:

The *OrdersTotal* function returns the number of opened and pending orders. If this number is **0** that means there are no orders (market or pending ones) has been opened.

Parameters:

This function doesn't take any parameters and returns an integer data type (the number of opened and pending orders).

Example:

```
int handle=FileOpen("OrdersReport.csv",FILE_WRITE|FILE_CSV,"\t");
 if(handle<0) return(0);
 // write header
 FileWrite(handle,"#","open price","open time","symbol","lots");
 int total=OrdersTotal();
 // write open orders
 for(int pos=0;pos<total;pos++)
   {
    if(OrderSelect(pos,SELECT_BY_POS)==false) continue;

FileWrite(handle,OrderTicket(),OrderOpenPrice(),OrderOpenTime(),OrderSymbol(),OrderLots());
   }
 FileClose(handle);
```

OrderSwap:

Syntax:

```
double OrderSwap( )
```

Description:

The *OrderSwap* function returns the swap value of the selected order.
The order must be selected by *OrderSelect* before calling *OrderSwap*.

A *swap* involves the exchange of principal and interest in one currency for the same in another currency. Currency swaps were originally done to get around the problem of exchange controls. (*Investopedia.com*).

Parameters:

This function doesn't take any parameters and returns a double data type (the swap value of the selected order).

Example:

```
if(OrderSelect(order_id, SELECT_BY_TICKET)==true)
   Print("Swap for the order #", order_id, " ",OrderSwap());
 else
   Print("OrderSelect failed error code is",GetLastError());
```

OrderSymbol:

Syntax:

```
string OrderSymbol( )
```

Description:

The *OrderSymbol* function returns the string representation of currency pair of the selected order (Ex: EURUSD and USDJPY).
The order must be selected by *OrderSelect* before calling *OrderSymbol*.

Parameters:

This function doesn't take any parameters and returns a string data type (the string representation of currency pair of the selected order).

```
if(OrderSelect(12, SELECT_BY_POS)==true)
   Print("symbol of order #", OrderTicket(), " is ", OrderSymbol());
 else
   Print("OrderSelect failed error code is",GetLastError());
```

OrderTakeProfit:

Syntax:

```
double OrderTakeProfit( )
```

Description:

The *OrderTakeProfit* function returns the *takeprofit* price of the selected order that you have set in *OrderSend* or modified with *OrderModify*.
The order must be selected by *OrderSelect* before calling *OrderTakeProfit*.

Parameters:

This function doesn't take any parameters and returns a double data type (the *takeprofit* price of the selected order).

Example:

```
if(OrderSelect(12, SELECT_BY_POS)==true)
   Print("Order #",OrderTicket()," profit: ", OrderTakeProfit());
 else
   Print("OrderSelect() âَهًë îꀉلê´- ",GetLastError());
```

OrderTicket:

Syntax:

```
int OrderTicket( )
```

Description:

The *OrderTicket* function returns the ticket number of the selected order.
The order must be selected by *OrderSelect* before calling *OrderTicket*.

Parameters:

This function doesn't take any parameters and returns an integer data type (the ticket number of the selected order).

Example:

```
if(OrderSelect(12, SELECT_BY_POS)==true)
   order=OrderTicket();
 else
   Print("OrderSelect failed error code is",GetLastError());
```

I hope the trading functions are clearer now.

I welcome very much your questions and suggestions.

Coders' Guru
20-12-2005