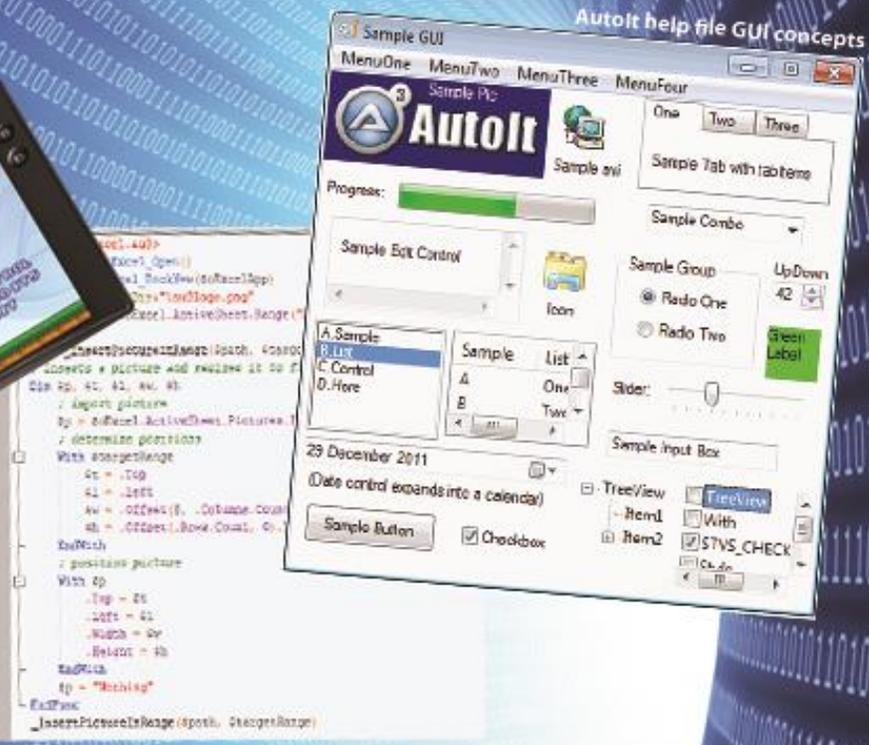# Learn To Program Using FREE Tools with AUTOIT

Foreword

When I released the first version of AutoIt in 1999 I did not think that it would attract a global following. Today the AutoIt forum has over 60,000 members who have generated over a million posts for coders of all levels. JFish, one of our forum members, recently created this book "Learn To Program with FREE Tools Using AutoIt". It provides an excellent introduction to programming and AutoIt. I would recommend it to anyone who wants to understand the basics along with plenty of examples.

- Jon, AutoIt creator [i]

Contents

Prologue

Section I: About me and the intended audience

I was always very competent when it came to using computers but I never wrote (or understood) a single line of code.   I was very interested in learning - but how?

I followed the usual path of going to the bookstore and sifting through all the "Teach yourself programming" books. It was dizzying. There were so many, covering so many different languages, that I did not know where to start. I eventually settled on "Programming for Dummies" – an uninspired rudimentary manual that raised more questions for me than it answered. Months and years of occasionally picking up books and putting them down resulted in even more frustration.   Then I enrolled in a beginner's class for Java programming at the Harvard Extension School.   It turns out that all I needed was the basics fed to me in a digestible way to start me on the path of being able to write my first program.

If you are like I was – someone who has the desire and has poked around a lot with various sources of information but never really made the connection to the material that allows you to succeed – then this book is for you.

Section 2: The approach

This book takes the approach of teaching you about **_all_** programming languages by introducing you to AutoIt – a free automation and scripting language for Windows.   The tools and programs that we will use can only run on the Windows platform. If you have a Mac all is not lost - you can run Windows on a virtual environment through a program like Parallels and install these tools on your virtualized Windows environment.

Some books take the approach of starting with a single application and adding to it. I find that this can be very confusing. If you don't understand any piece of the application your confusion may interfere with subsequent lessons. We will start off by studying core concepts and will demonstrate them in actual programs that you can build and run.

NOTE: **Be patient**. I know the urge is to just skip all of the foundational pages and get to a program to see it work but you won't do yourself any favors if you don't understand *how* it works. Believe me; I have done the same thing.

Section 3: Why this programming language?

We are using AutoIt because it a very powerful free tool.   The other reasons are that AutoIt has an incredibly well documented help file and user forum. You can use the help file to figure out how to do just about anything (this book tracks very closely to the help file). If you ever get stuck, you can visit the AutoIt online forum and ask a question. As long as you follow the forum rules and are courteous you will almost always get a super-fast response that guides you in the right direction.   AutoIt is also very powerful. You can create full featured applications very quickly. As a testament to that there is a

section in the forum that contains example applications and it is filled with many stunning entries that you can download, use, and even change (assuming the author provides permission to do so) to suit your own needs. And did I mention it was FREE? Don't worry if you have never heard of it before and want to program a specific language like C#, Java, PHP, etc. This book will provide you with the ability to switch languages very quickly. That is because all languages have a similar foundation of core concepts.

Chapter 1: Types of Data

Programs are basically sets of instructions for computers.   To make a program that does what we want it to do we need to understand some of the basic types of data that are used.   Every programming language has this concept.

NOTE: You will often see it referred to as "data types" or "datatypes." The transformation of the layman's description "types of data" to the more technical term of art "datatypes" is fairly benign because they are almost identical. That won't always be the case as we explore other technical vocabulary words that we will need to understand throughout the materials but I will always try to make sure the root meaning is well understood so that you don't have to keep flipping back pages to figure out a definition introduced in a prior section.

To better understand different types of data consider for a moment that source code, the code that we write as programmers when building our applications, has to be human readable (else how could we work on it?).   As such, we are primarily dealing with letters and numbers – the components of our everyday language.   We also expect to see output from a program in human readable form.   For example, if I want to write a calculator application I will need to create a program that understands how to display numbers and perform mathematical operations on them and then display the results.

Just like in the real world, within programming there are many different types of numbers. There are whole numbers, those that have a decimal point, negative numbers, positive numbers etc. Likewise, when we think of letters we know that we could have a single letter, a grouping of letters that forms a word or a sentence, etc.   Programming languages account for these nuances by referring to the different options as types. For example, a whole number without any decimal places after it is referred to as an integer. That vocabulary word is not terribly important at the moment other than to note that there are other names for other types of numbers.

So what's in a name? What difference does it make what I call a number that I want to use? Consider the calculator example for just a moment. I am writing a program for a calculator and I decide to use only integers (or whole numbers).   Would that work? What would happen if I divided the number 1 by the number 2? The answer is .5 and that is not a whole number. Therefore, strictly using the datatype integer that only allows for whole numbers in my calculator could lead to problems. It could produce the wrong result or crash the entire application.

NOTE: When something unforeseen and unintended occurs in your program we call that a bug. Bugs are inevitable – especially when you are just starting out. Being careful and planning can help you reduce the number of bugs and get a solid program up and running in no time.

Some programming languages are syntactically stricter than others. That means that within the program you are writing you have to be extremely careful with your punctuation and the way you refer to all of your data or it may not run correctly. Many languages will make you "declare" your datatype before you start using it. In other words, if you wanted to use an integer you may have to signal to the program that is your intention. One of the nice things about AutoIt is that it is a "looser" language. It does not require strict declarations that can trip up new programmers. Just be aware that if you transition to another programming language you may need to account for those types of differences.

In AutoIt there is only one datatype called a **Variant**. A variant can contain numeric or text data and decides how to use the data depending on the situation in which it is being used.[1] In other words, as we write AutoIt applications what we are doing will automatically determine the internal datatype. Since there is only one official datatype "variant" AutoIt does all the work for you and uses internal datatypes, i.e. those commonly used in the other languages behind the scenes (such as " integer"). That means that we are not required to declare them explicitly in our programs. AutoIt will take care of that for us. Another internal numerical datatype that is very common is called a "double." AutoIt's help file defines a double as "A precision floating point number." Said another way, this is a number that can use decimal places. Using doubles may be a better choice for the arithmetic operations of our hypothetical calculator application.

We also need to understand letters. Most programming languages refer to text as a "string" (an easy way to remember this is a "string of letters"). You can create a string in AutoIt simply by enclosing your text in quotation marks (single or double quotes). The following are all strings: "Hello World", "Dog", and "Cat". How about "100"? Is that a string or a number? The answer is string. If instead we were to simply type 100 without the quotation marks it would be treated as a numerical value. Once you wrap a number in quotes the program thinks of it as text.

NOTE: It is beyond the scope of this section but you can "cast" one data type to another. So if you have a number in a string "100" you could explicitly tell the program to treat it as a number by "casting" it to that datatype. However, it will default to text unless you do so.

A "Boolean" is another internal datatype that is very useful. It is a logical value that can be either True or False. It cannot be anything else. You can use Booleans in your applications to test certain logical conditions. We will talk more about that later.

---

[1] From the AutoIt help file "Language Reference – Datatypes"

REMINDER: We need data to write programs. Different types of data are referred to as datatypes. AutoIt, our programming language of choice, has only one datatype called "variant." The language automatically knows which datatype to apply internally based on the context of its usage. Integers and doubles are internal numerical datatypes used for whole numbers and precision numbers respectively. A string is an internal datatype used to refer to text. A Boolean is an internal datatype that can only be true or false.

Chapter 2: Variables

Your program will almost certainly need to temporarily store some information for one reason or another. The place this information is stored is called a variable.

Variables have whatever names you give them. That is why it is important to give them a name that is meaningful. Think of it this way, you will be writing programs that could have hundreds or even thousands of lines of code. They may contain many variables. If you give them names like "variable1", "variable2" and so forth you will start to forget what you are using them for as you get further into your application. Compare that approach to more descriptive storage names like "first name" and "age". It is hard to forget what you are storing in those variables because their names describe the information. You can store all sorts of information in a variable. You can store numbers like: 10, 2.14, -128.223. text such as "the quick brown fox jumped over the log", and much more.

This is all starting to sound very technical so let's use the very first line of code that appears in this text to visualize the concept of variables. To understand what we are about to look at you need one more piece of information. In AutoIt each variable name (remember we can call them anything) must start with a dollar sign "$". Here is our first variable: $message.

We can point our variable to information by using an equal sign. The information that follows is "assigned" to the variable.

$message = "your first variable"

The above code is creating a variable called "$message" and using it to store some text. We could have used any name we wanted but since we are storing a message "$message" is descriptive. The equal sign that follows the $message variable is used to assign the text "your first variable" (which is also referred to as a "string") to $message. Therefore, we have assigned the string "your first variable" to the variable "$message."

There are a few more things about variables that we will explore in subsequent chapters. The above explanation should be enough to get us started.

REMINDER: variables are places to store information.   We can use any name we want for a variable but should strive to make those names descriptive.   In AutoIt the name of our variables are always preceded by a dollar sign "$". Although variables can be used to store various datatypes such as, numbers, text, or Booleans (true / false) AutoIt only has one datatype: variant, and it figures out which internal datatype to apply for us automatically.

Chapter 3: "Hello. Operator. How may I assist you?"

We already know all about datatypes that describe numbers and letters along with variables where we store information.   Now we may want to take our data and compare it or perform arithmetic on it (i.e. do something with it). These types of actions are facilitated by operators.   You already know many of the operators even if you don't realize it. You learned them in grade school. The operators for addition, subtraction, division, and multiplication are as follows: +,-,/,*.

Let's combines some concepts here to see operators in action. In this example we will create two variables $firstNumber and $secondNumber to which we will assign values 3 and 5 respectively:

$firstNumber = 3, $secondNumber=5

Now those variables are storing our information and we can use them to make a little program. To illustrate this I will create a third variable called $answer. We will use it in our program to store the results by assigning the sum of the first two variables to it. We will use the operator "+" to add the first two variables together.

$answer = $firstNumber + $secondNumber

If we printed out the contents of the $answer variable after we ran this simple program we would see the numeric value of 8. This example allowed us to use datatypes, variables, and operators to produce the sum of two variables. Adding 3 + 5 is not an example of a particularly powerful program. You may never want to write that since you already know the answer but consider the use of bigger numbers. AutoIt even supports scientific notation. What if simply changed the values of $firstNumber and $secondNumber to the following values expressed using scientific notation 1.826395e7 and 23.34e3 and multiplied them rather than add them together? If you can do that in your head kudos! For those who can't the answer is: 426280593000. I only know that because I ran the program to get the answer. The point is that we can get our programs to easily perform tasks that we would find difficult or impossible – and they can do it at incredible speeds.

We won't cover every operator available in the AutoIt language. You can see the complete list at any time in the AutoIt help file that you will receive when we install the software (or you can view the online version on their website).[2] For now, let's cover a few of the more prominent ones:

>	Tests if the first value is greater than the second.
>=	Tests if the first value is greater than or equal to the second.
<	Tests if the first value is less than the second.
<=	Tests if the first value is less than or equal to the second.
=	Tests if two values are equal.
<>	Tests if two values are not equal.
&	Joins two strings.

Many of these operators also probably look familiar to you from your experience outside the programming world. The syntax you may have used in the past may be slightly different but hopefully the concepts look familiar.

I will demonstrate how they are used within a program. Let's imagine that you are writing a program that will decide whether or not somebody is eligible to vote.   We know the voting age is 18. Our program could ask the user to enter their age. We know that if a user's age is less than 18 or, using an operator <18, the program could determine they are not eligible. We also know that if the user's age is >18 they are eligible to vote.   The other two scenarios may be slightly less obvious. Let's start with greater than or equal to >=18. That means that if the user is equal to 18 _or_ older than 18 they are eligible to vote. That operator would suit us just fine as it seems to cover all the scenarios to determine if the user can vote (i.e. 18+). Using only > instead of >= would not allow us to correctly determine that somebody who is exactly 18 is eligible because it would only identify someone older than 18. Likewise, using <=18 to determine who is not eligible would lead to the false conclusion that someone whose age is equal to 18 is not eligible.

We may also want to compare values to test whether or not they are equal to each other. For this we would use = and <> respectively. To demonstrate this let's use a different example of a program that randomly selects from three different greetings: $greeting 1, $greeting 2, and $greeting 3. How might we tell the program what do to if it selects $greeting 2? One possibility is that we would use the = operator to say that if the random selection = $greeting 2 say "hello from greeting 2". You can also imagine how we might use not equal, <>, for similar tests and instructions within our program. For example, let's say it was important to your program to determine if the randomly selected greeting is anything other than $greeting 2.   To do that you would instruct the program to test if random selection <> $greeting 2. If that test is true then it must be something other than $greeting 2.

---

[2] http://www.autoitscript.com/site/

Let's cover one last operator used to join two strings together. We know from our discussion on dataypes that a string is information appearing between quotation marks that the program treats as text. We already assigned a string to a variable when we wrote: $message="your first variable". Now what if we wanted to add to that message? What if we wanted the entire message to say "your first variable. Wow! Look we are programming!". One approach could be to re-assign that entire message to the $message variable as follows: $message = "your first variable. Wow! Look we are programming!". However, thanks to our friend the ampersand operator, &, we don't have to do that. The code below demonstrates the use of the ampersand operator to concatenate (i.e. combine / join) two strings together. The code also uses semicolons to add comments that are ignored by the computer when it runs (see the note below the code explaining comments in more detail):

$message="your first variable" ; this is a comment explaining that we already wrote this earlier

$additionalMessage = " Wow! Look we are programming!" ; note the extra space before Wow.

$combinedMessage = $message & $additionalMessage ; will produce "your first variable. Wow! Look we are programming!"

Let's discuss what we just did above. We simply joined two strings, or text, together with the "&" operator. The result is that the messages are combined into a single block of text. It is worth mentioning that they are also still separate in the first two variables we used of $message and $additionalMessage. They are only joined in the $combinedMessage variable. Therefore, the comments below reflect what we would see if we where to print all three:

$message ; "your first variable"

$additionalMessage ; " Wow! Look we are programming!"

$combinedMessage ; "your first variable. Wow! Look we are programming!"

NOTE: Every language has a way to add notes to that program that are ignored when the program runs. We call these comments and they are useful to programmers to document what they have done as well as teach others reviewing their code. In AutoIt we use the ";" to comment on a line of code. Everything that follows the semicolon on that line will be ignored by the program when it runs. Think of it as a way to add your own sticky notes within the application.

REMINDER: operators help us perform operations with our data. Many of them are intuitive because we have seen them before outside of programming. Some examples of common operators include greater than ">", greater than or equal to ">=", less than "<", and less than or equal to "<=". We can also test to see whether or not things are equal with the = and <> operators. The ampersand operator "&" can join strings together. Finally, a complete list of AutoIt operators can be found in the help file.

Chapter 4: Let's Program Something Shall We?

NOTE: This chapter is broken into two sections: Installing AutoIt Development Tools and Our First Program. If you already have AutoIt installed you can skip the first section of this chapter.

The first three chapters provided us with an introduction to datatypes, variables, and operators. Those concepts help define the various types of data we will be working with, how to store them, and how to perform operations on them respectively. There are many additional nuances on all three subjects that we did not cover because some of it will be covered in subsequent chapters and some of it is beyond the scope of this text.   The information we already covered is enough to get started writing our first program. However, we will need to download and install some tools used to write the programs and test them. The following instructions demonstrate a step by step installation process to get you started:

Chapter 4: Section 1: Installing AutoIt

Before we can install AutoIt we have to download it from the official website at: http://www.autoitscript.com. Once on their site you will see a section for downloads. If they re-design their site it could move but as of the printing of this book you can find it here.
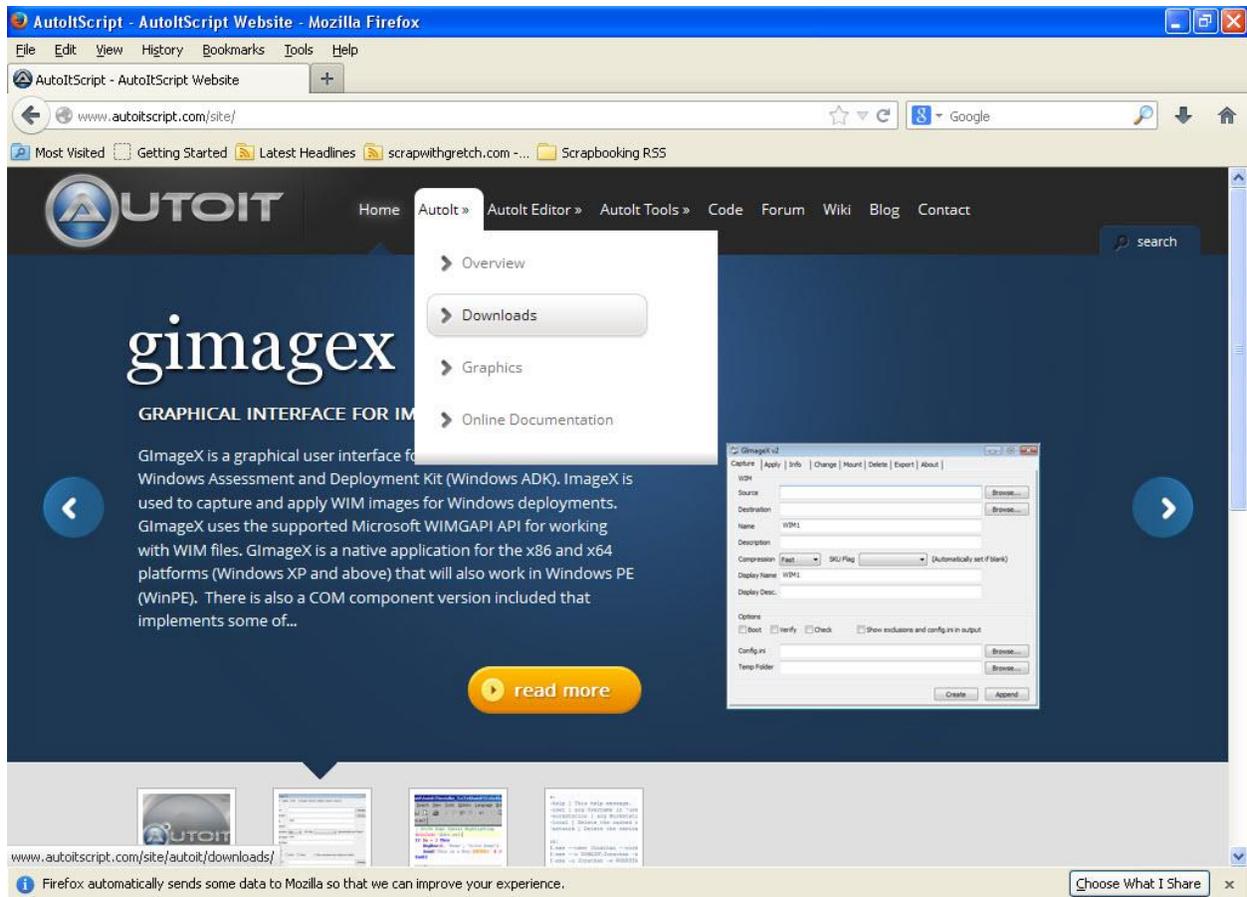
Once you click into the download section you will be presented with several options. I recommend selecting the "AutoIt Full Installation" package because it will add all the necessary components to your system with one easy installation.

Software	Download
AutoIt Full Installation. Includes x86 and x64 components, and: • **AutoIt** program files, documentation and examples. • **Aut2Exe** – Script to executable converter. Convert your scripts into standalone .exe files! • **AutoItX** – DLL/COM control. Add AutoIt features to your favorite programming and scripting languages! • **Editor** – A cut down version of the SciTE script editor package to get started. Download the package below for the full version!	Download AutoIt
AutoIt Script Editor.(Customised version of SciTE with lots of additional coding tools for AutoIt)	Download Editor
AutoIt- Self Extracting Archive (for those who don't like/want an installer)(includes x86 and x64 components and **Aut2Exe** and **AutoItX**)	Download ZIP

When you download, click the file to open it and you may see a warning that you are opening an executable (exe). Click "OK":
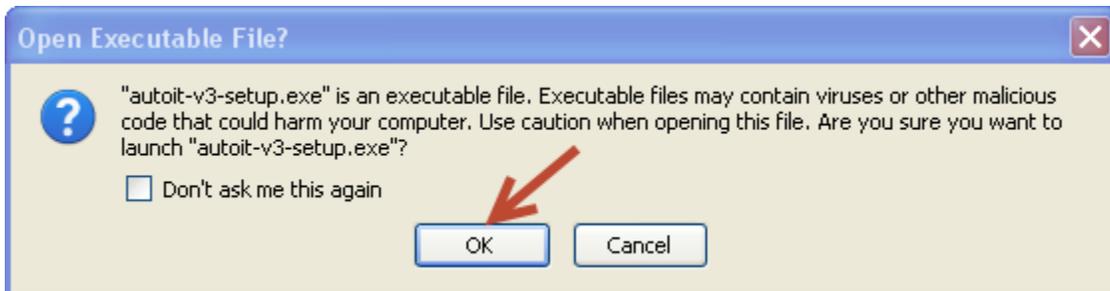
After that you will be presented with a window asking you if you want to run the application. Click "Run".
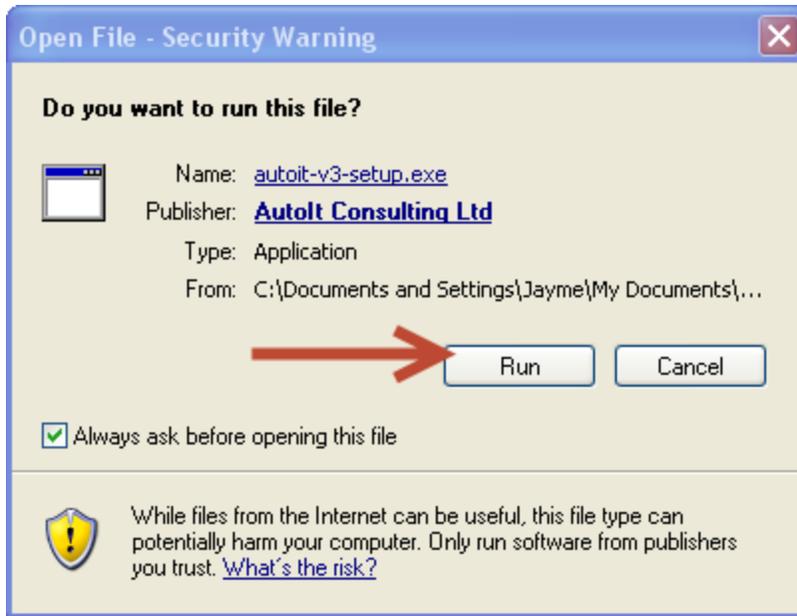
Next you will be in the installer that will walk you through the options to install the AutoIt application. Click "Next".   NOTE: the version numbers change with each release. They have been replaced with black boxes to avoid confusion between the version number in effect at the time this text was written and the latest version that you may download and install.
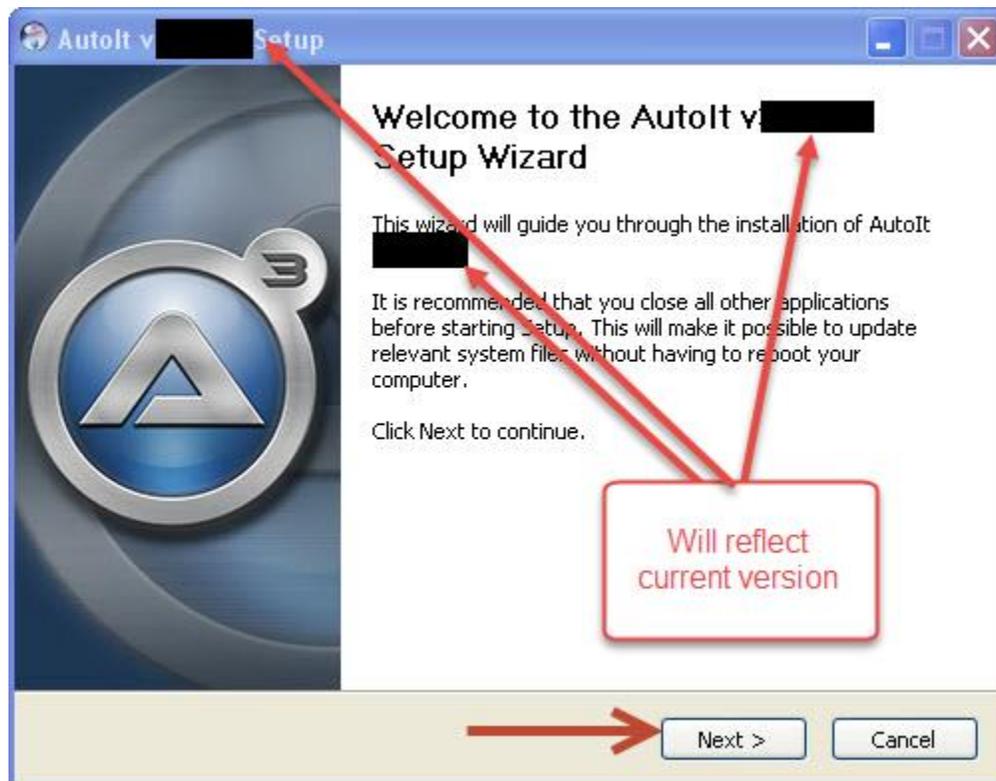
Then you will be presented with the license agreement. Click "I Agree":
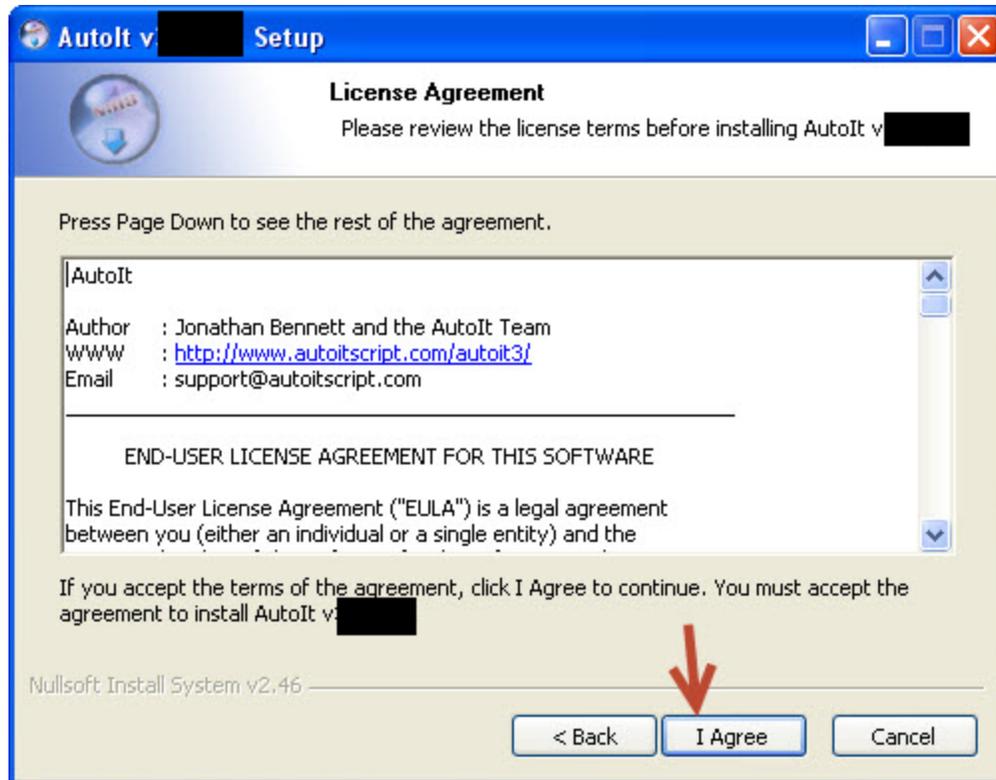
AutoIt saves files that you create with a *.au3 file extension. Those files are the source code for your programs. You can run them as a program and/or edit them. The next screen asks what you would like the default behavior to be when you click on an *.au3 file. It will either run by default or open up in an editor for you to edit. If you pick run you can still edit the file by opening it in the editor. If you pick edit you can still run the file. This is only your personal preference for what happens when you click the file. AutoIt is a "scripting" language so programs are often referred to as scripts.[3] The installer is defaulted to "Run the script" whereas my personal preference is to "Edit the script".   Pick the option that suits you best and click "Next".

REMINDER: AutoIt saves files that you create with a *.au3 file extension. Those files are the source code for your programs. You can run them as a program and/or edit them.

---

[3] Scripting languages are known for the automation of tasks which could alternatively be executed one-by-one by a human operator. http://en.wikipedia.org/wiki/Scripting_language.
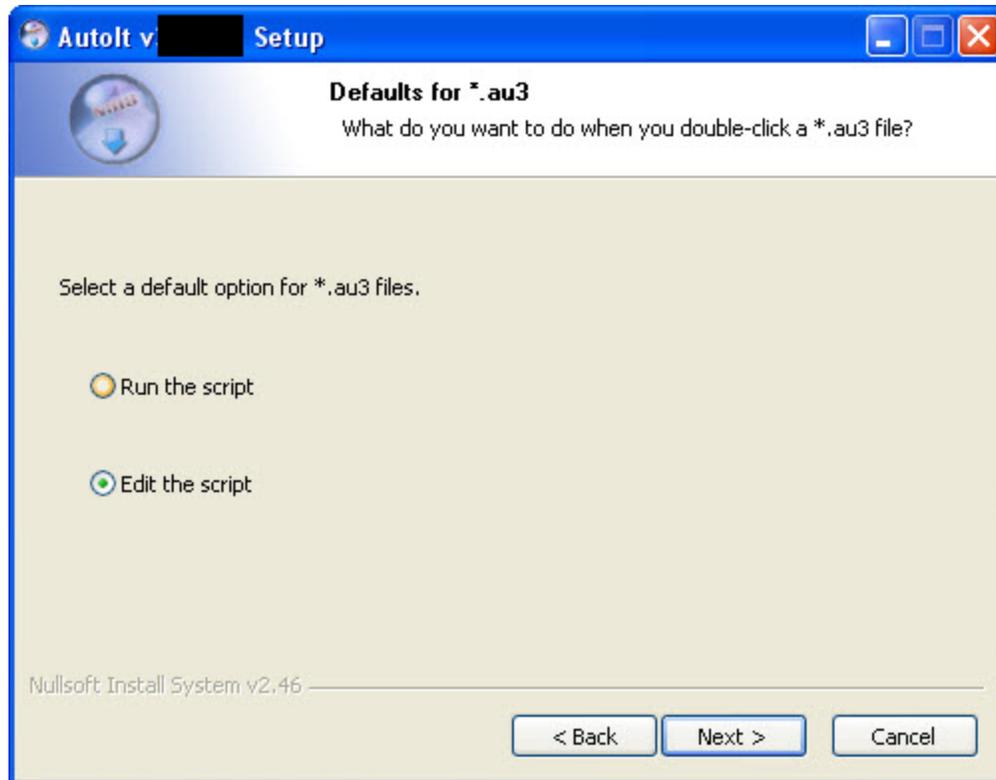
You will then see a screen prompting you to select the components for install. Make sure all the boxes are checked and then click "Next".
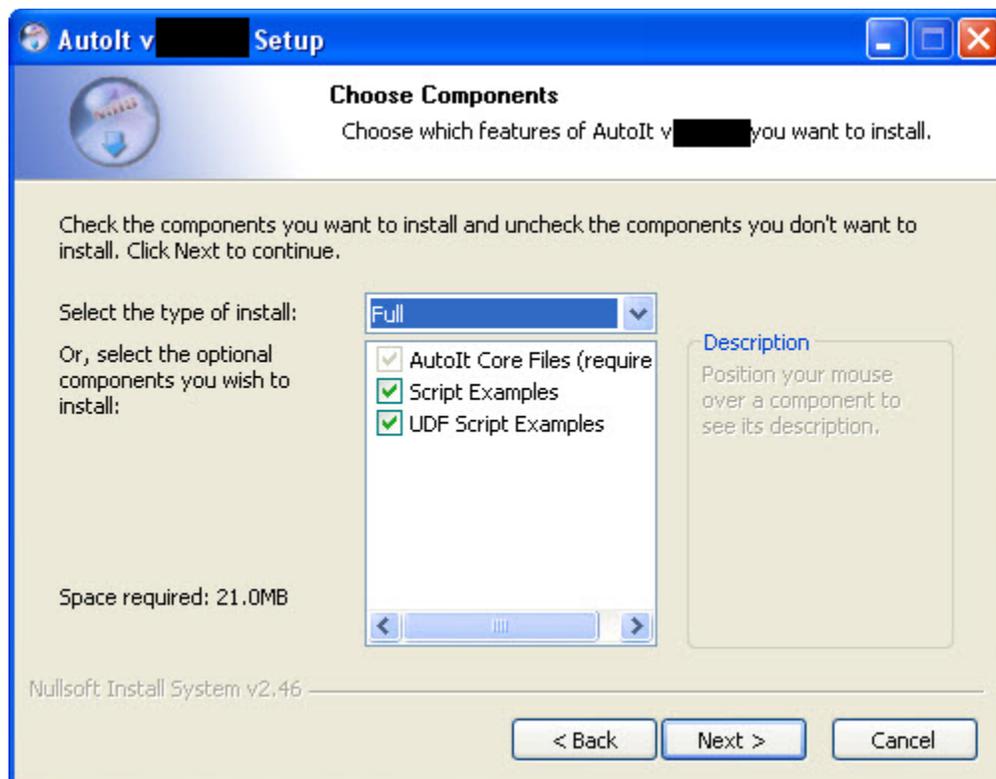
Next you need to select the location to install the AutoIt application. You can use the default location that appears in the window or change it using browse. I recommend using the default. Click "Install" when done.
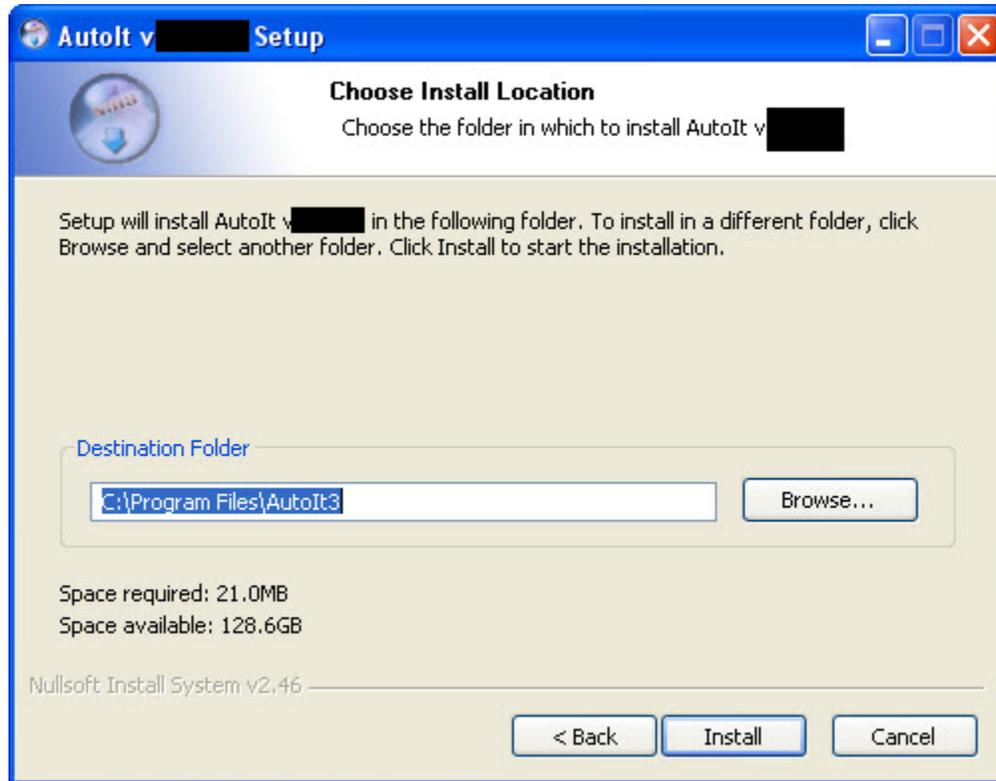
The application will then start to install:
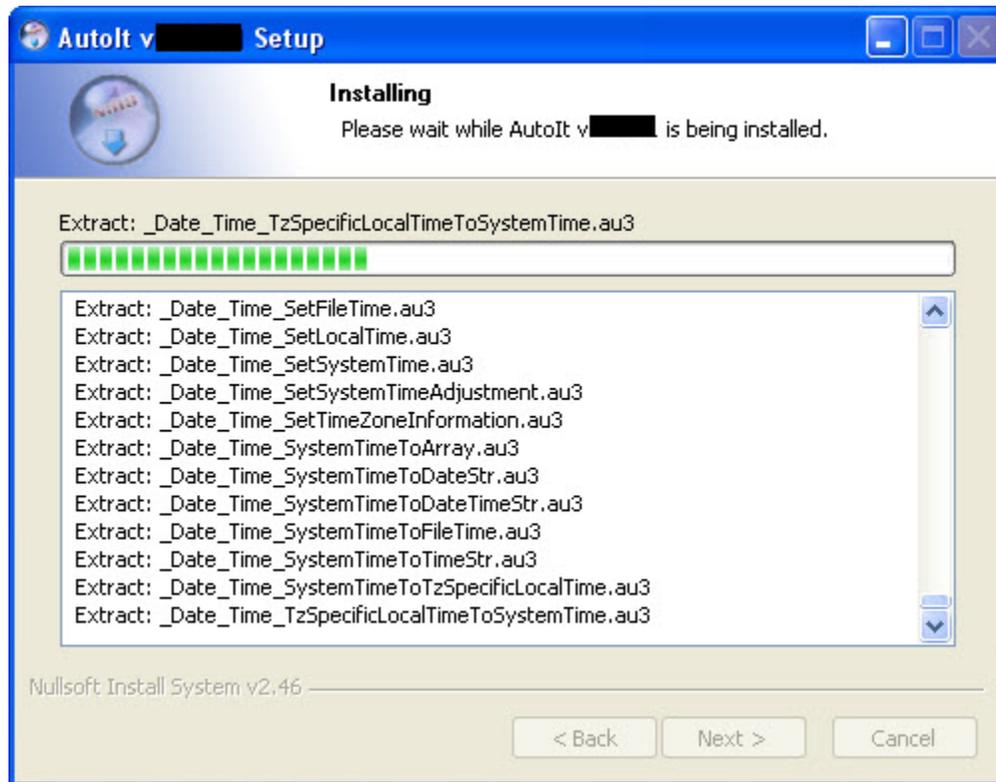
When it is done installing you will see a confirmation screen with a "Finish" button. Note that the default checkbox with the green arrow will show the release notes when done. Release notes can be important because they reflect all the changes to the AutoIt language that could potentially interfere with your applications if you write something and then upgrade AutoIt. However, for now, since this is your first program you can uncheck that box and click "Finish".

If you check your installed programs you should now see AutoIt listed.   Open the directory where you installed and you should see the following:
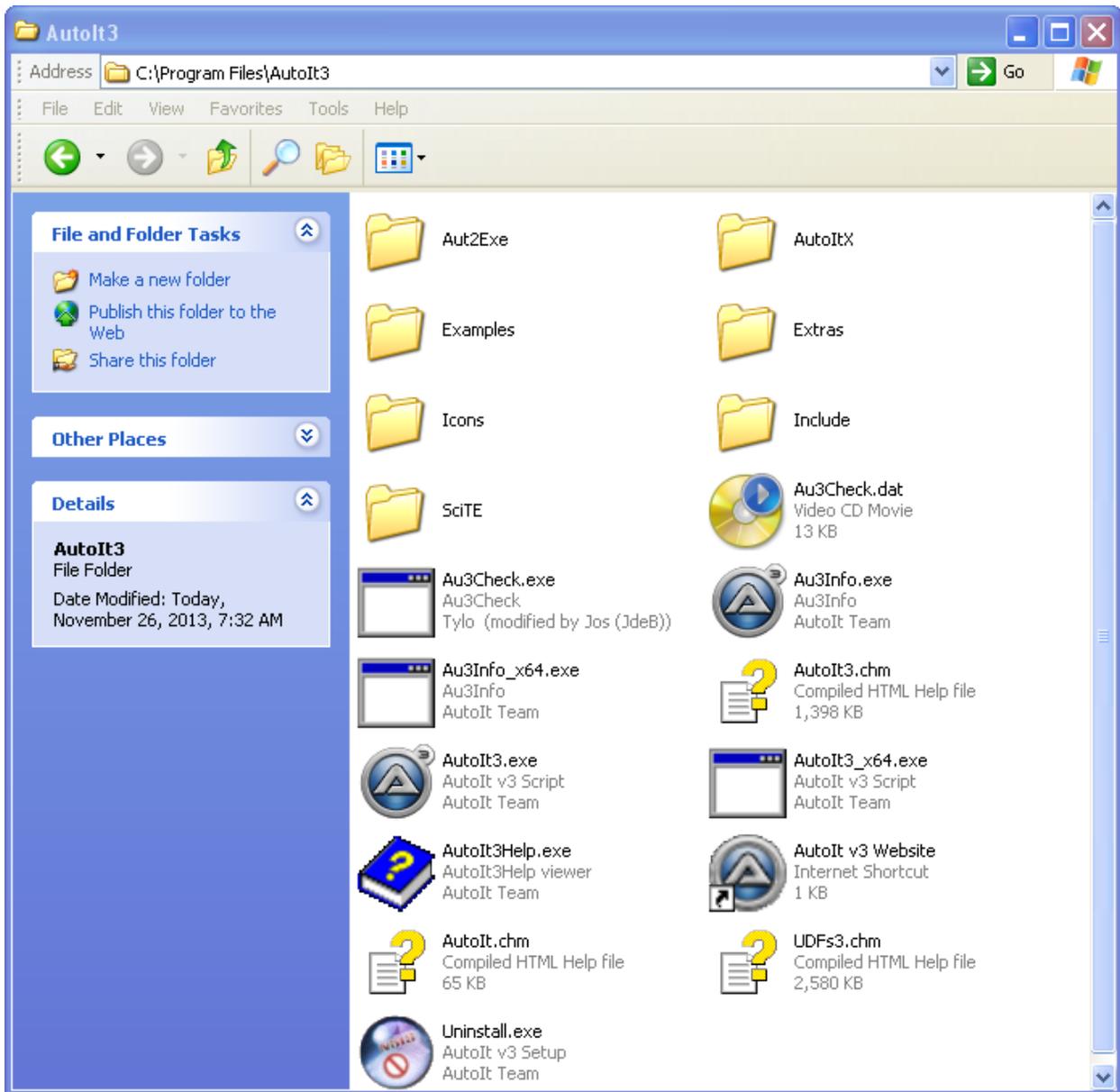
Two very important files are: AutoIt3.chm (note: use the one with the '3') and inside the SciTE directory (folder) there is a file called SciTE.exe. These files appear as follows:

Help file	
SciTE (we will use this to edit scripts)	

REMINDER: The AutoIt help file is a very valuable resource. It is very detailed, contains tutorials, examples, and is searchable. Reading the help file will assist you in gaining a better understanding of the language and programming in general.   SciTE is the tool we will use to create, edit, and run our programs. It has a built in syntax highlighter for AutoIt that will help make our code more readable.

NOTE: It is also worth noting that AutoIt has a Wiki. The wiki is located here: https://www.autoitscript.com/wiki and contains a lot of valuable information beyond that in the help file. It is yet another resource at your disposal to search for answers to common questions.

Chapter 4 Section 2: Our first program:

If you had any difficulty with any part of the install process you can go to the AutoIt website and post a question about installing to the forum (please search the forum for answers first before starting a new thread) and/or read the help file and FAQs. If all went according to plan you have everything you need to create your very first program (also known as a script).

SciTE is a programming editor, a tool that allows us to write programs, that comes pre-loaded with AutoIt keywords and functionality. It has the ability to highlight certain key words and areas of your code to make them easier to read, auto-complete commands, and more.

Find the SciTE.exe application that was installed in the SciTE directory where you installed AutoIt. It should look like the above picture and double click it to open the editor.

You should see something like this when SciTE opens:
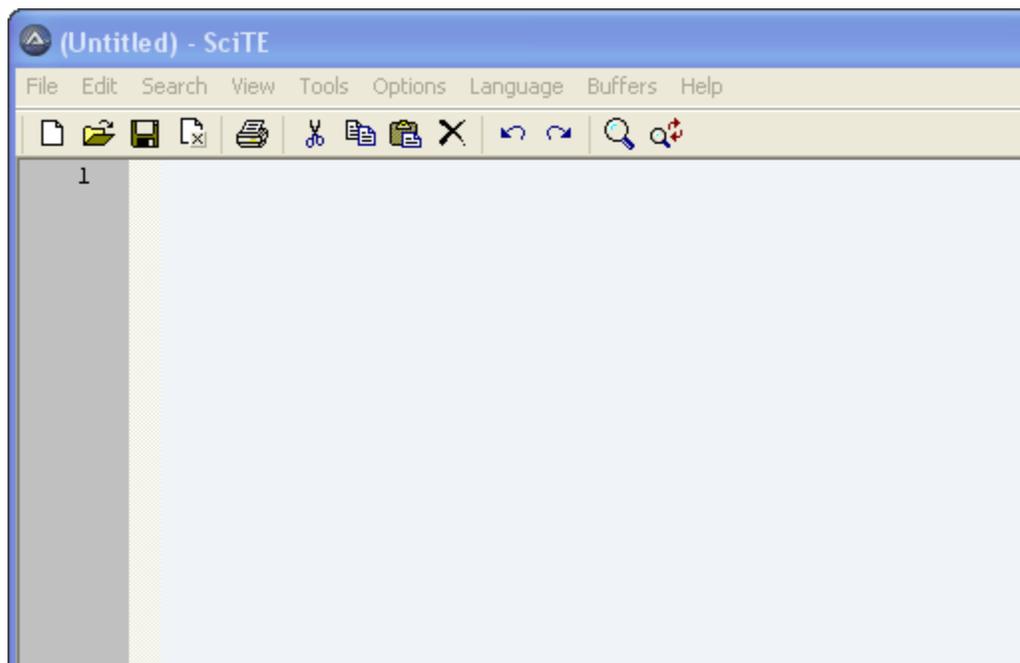
Note that SciTE is very similar to a word processor. You can see many icons that you are probably already familiar with for saving, printing, copying, pasting, etc. However, as a program editor (also referred to as a "script editor" or an IDE "Integrated Development Environment") it can also help you create, edit, and run your scripts (remember we will sometimes refer to our programs as scripts and vice versa).

The perennial favorite first program for all languages is "Hello World" where we write a simple program that displays the "Hello World" message back to us. I don't want to rob you of the experience of seeing those magic (and somewhat anti-climactic) words on your computer screen so I won't break from tradition – that will be our first program. From prior chapters we know that we are going to need to create a string because the information we want to display is a message surrounded by quotation marks. This signals to our program that we are dealing with text. AutoIt can figure out that the internal datatype is a string because of the syntax (use of quotation marks). However, we have not discussed how we are going to make the computer display the string back to us. To do that, we will use a function.

Every programming language has built in functions that perform useful tasks. They are pre-built collections of commands created for programmers in an effort to reduce the amount of coding you have to do (you can also create your own). We will cover many of the functions in the help file in subsequent chapters. For now, we only need to understand that there is a pre-built function that can create a message box that pops up on the screen and prompts us to close it with an "OK" button. The image below reflects the section of the AutoIt help file that speaks to the creation of a message box. We will study the command in detail and then use it in our first program.

MsgBox

Displays a simple message box with optional timeout.

```
MsgBox ( flag, "title", "text" [, timeout [, hwnd]] )
```

Parameters

flag	The flag indicates the type of message box and the possible button combinations. See remarks.
title	The title of the message box.
text	The text of the message box.
timeout	**[optional]** Timeout in seconds. After the timeout has elapsed the message box will be automatically closed. The default is 0, which is no timeout.
hwnd	**[optional]** The window handle to use as the parent for this dialog.

MsgBox function screenshot above from AutoIt help file

In order for the message box to display the appropriate title and message we have to feed certain information to it. These slots where we put this information are often referred to as parameters and the data we supply are called the arguments. The help file screenshot for the MsgBox function shows the command required to create the message box which is in yellow (i.e. typing "MsgBox" followed by parenthesis that will contain the parameters which are separated by commas). Directly below the command we see an explanation of the parameters.   They are flag, title, text, timeout, and hwnd respectively. All of these parameters are responsible for different facets of the message box. For example, "flag" allows us to manipulate the appearance and behaviors of the message box we are creating. There is additional information on the message box function in the help file that you can review to understand all the options and what you would need to supply as a value for that parameter to create your desired behavior/appearance.   The parameters title and text will set the title of the message box and the text contained within it respectively.

A quick review of the parameters shows that some appear within quotation marks while others do not (i.e. title and text are in quotes).   That is because the function expects us to input strings (text within quotation marks) for those parameters. Said another way, the ***arguments*** we supply should be strings

for those two parameters. Conversely, the parameter flag is not in quotes. That parameter is expecting a numeric value. If we put in quotes the program will think it is text and get confused.

We can also observe the last two parameters of timeout and hwnd are preceded by open brackets. That means they are optional (as you can see in the comments from the help file that explain them). We could optionally supply a timeout parameter so that the message box closes automatically after so many seconds or we could leave it out entirely and it will use the default behavior that requires a user to click "OK" to close it. That is different from the first three parameters. Those don't have brackets. Therefore, they are mandatory. We must supply all three arguments (one for each parameter of flag, title, and text) or the program will not run. Instead, it will produce an error telling us that we did not supply the required parameters to the function. However, we may be able to supply blank arguments – which still satisfy the requirement of supplying the mandatory items. Passing an empty string (i.e. double quotation marks with nothing in them: "") as an argument will count. An example of this would be if we wanted to create a message box with a title at the top. Then we could use double quotation marks ("") for the title parameter.   The hwnd optional parameter is a bit advanced for the purpose of this discussion.   Handles are basically ways to refer to resources that we want to interact with. We will cover that more in subsequent chapters.

Now let's use this function and supply some parameters to create our own message box. Follow these steps, which we will review after we run the program:

NOTE: This book comes with a companion set of files containing the code examples. Code examples that appear in the file are organized by chapter. They are also called to your attention in this text with the following tag:

Code

In this case, the example below will be found in the Chapter 4 directory and labeled as Example 1

Code Chapter 4 Example 1

First, type the following into SciTE:

```
File  Edit  Search  View  Tools  Options  Language  Buffers  Help

   1    #include <MsgBoxConstants.au3>
   2    MsgBox($MB_OK,"My first program","Hello World"); this is a comment - good job
   3
```

Next, save the file with the name "hello world.au3." You can save it anywhere but I recommend creating a folder that will hold all of your programs.
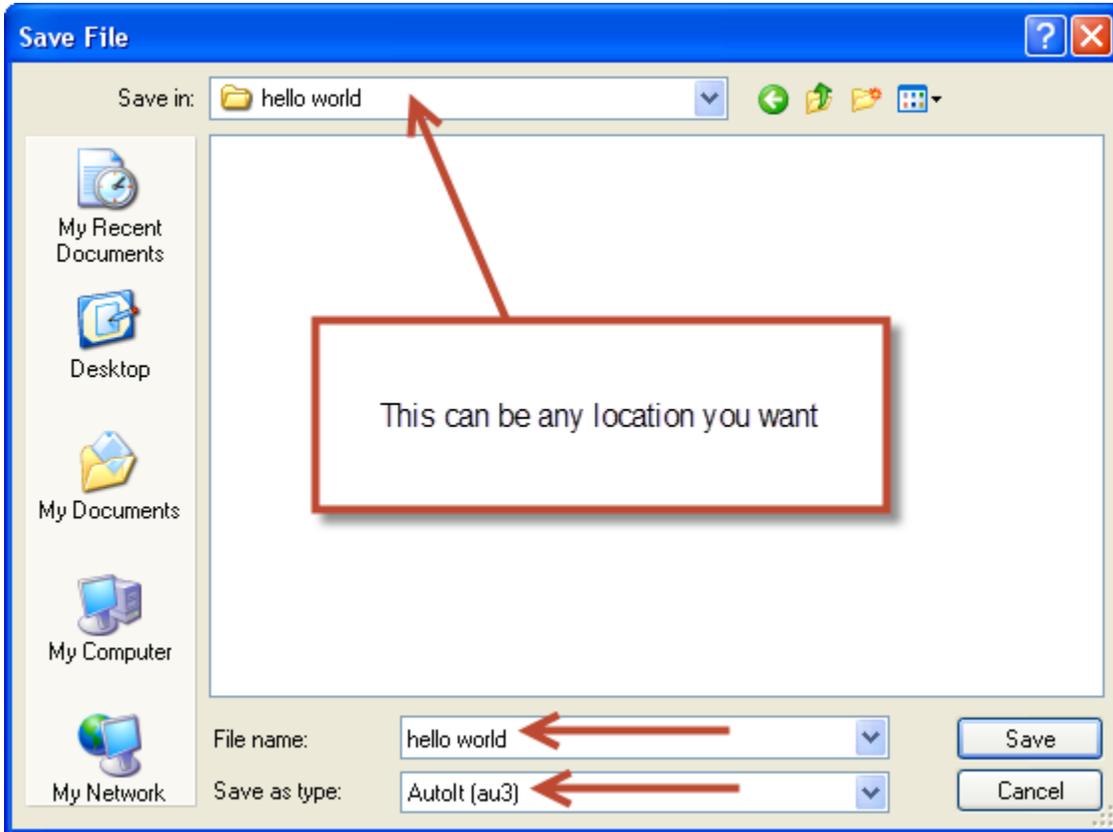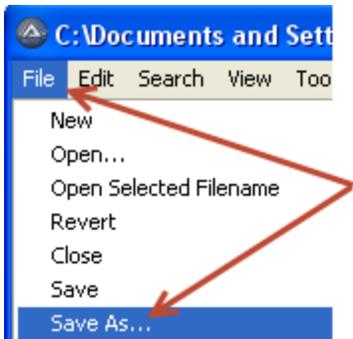
Now go back to SciTE and select "Tools" from the top menu, then "Go" (this is how we run the script):
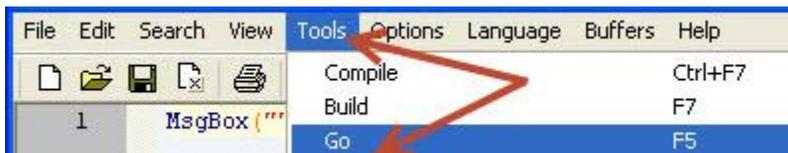
You should see the following:

Click "OK" and the box will disappear. Congrats! You just wrote your first program and ran it. This was a simple one-line program but there is a lot of useful information if we dissect it a bit. The following diagram breaks the program into five numbered pieces so we can easily refer to them:
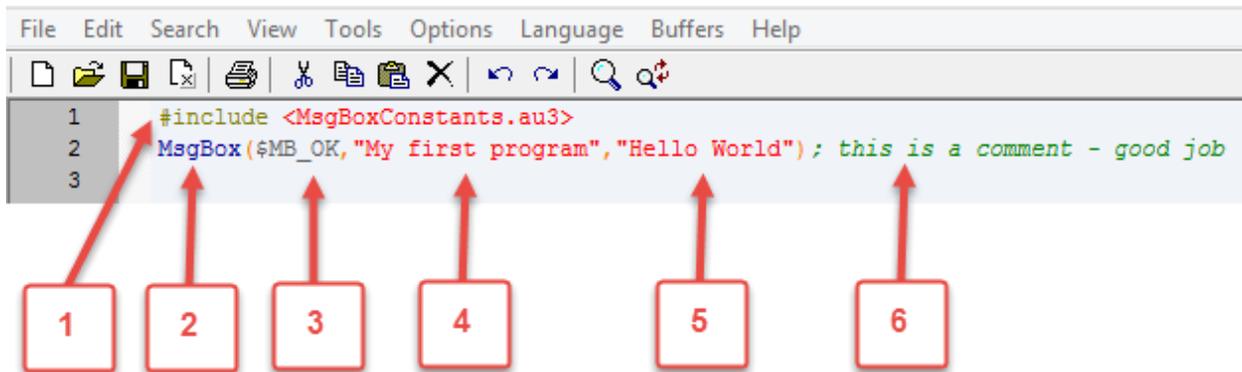
#1) Here we are using a keyword that we have not previously covered called "#include". This simply references another named script to bring in or "include" within your script. It is handy for including functions created by others, making your code easier to read by breaking it into sections with separate scripts, etc. See #3 below for an explanation of what we are including.

#2) We know from the help file that the term "MsgBox" is a pre-built function that created the message box we observed when we ran the program. For now, we need to understand that in order for the message box to display the appropriate title and message we have to feed certain information to it. These slots where we put this information are often referred to as parameters and the data we supply are called the arguments. Our arguments appear within the parenthesis and are separated by commas.

#3) The flag parameter: The message box is customizable with different options. The first parameter of "flag" is a mandatory parameter. We must supply an argument for it in order for the message box to appear. The help file tells us that we could use this parameter to change the appearance and behavior of the box when we supply an argument. There are many different flags available and referenced in the help file. A partial screen shot of the help file is shown here:

Constant names coming from the included file.

Numeric flag that you could use instead of the constant.

The types of different buttons you will see based upon the flag you use.

Within the MsgBoxConstants.au3 file this would be: $MB_OK=0

Constant Name	Decimal flag	Button-related result
$MB_OK	0	OK button
$MB_OKCANCEL	1	OK and Cancel
$MB_ABORTRETRYIGNORE	2	Abort, Retry, and Ignore
$MB_YESNOCANCEL	3	Yes, No, and Cancel
$MB_YESNO	4	Yes and No
$MB_RETRYCANCEL	5	Retry and Cancel

The MsBoxConstants.au3 file contains a set of variables that do not change. They are referred to as "constants". The constants in the file give human readable names to the numeric flags used to customize the message box with various options. For example, a flag with the value of zero (i.e. 0) will create message box with an "OK" button. The file we included contains a constant of "$MB_OK" and assigns 0 to it as follow $MB_OK= 0. Now, instead of using a number that does not make much sense to us when we read our code (i.e. how are you supposed to know 0="OK"?) we can use $MB_OK which clearly conveys that we are creating a message box (MB) that has an okay button (OK).

NOTE: The use of #include <MsgBoxConstants.au3> is optional. It merely gives you access to the human readable constants. If instead, you simply wanted to use 0 to show a message box with an OK button you could do the following:

```
MsgBox(0,"some title","some text")
```

That will produce the same result as:

```
#include <MsgBoxConstants.au3>
MsgBox($MB_OK,"some title","some text")
```

#4) The title parameter: The second parameter for the message box controls the title to the window that pops up. You will note that when you ran the program the window had a title of "My first program" that appeared in the blue title bar at the top of the popup window. If you were to leave this blank the popup would not have a title. Alternatively, you could change this to anything within the quotation marks and create a new and/or different title.

#5) The text parameter: The third parameter that we supplied dictates the message that will appear within the body of the message box window. We supplied "Hello World" and that is what we saw when we ran the program.

#6) The human-readable comment that does not impact the program: This was not required. I added it so that we could study how to make notes in our program using comments that have no impact on the actual script / program we are running.

It is also worth mentioning that we did not supply any arguments for the optional parameters of timeout and hwnd. This is different than when we passed a blank argument to the optional parameter of flag. In this case we did not supply any arguments – blank or otherwise. The program is fine with this because they were optional.

REMINDER: Autoit comes with pre-built functions that can help reduce the amount of coding required to get your program up and running. Many of these functions have parameters that impact the outcome of the function when it is used. We can pass in data, known as arguments, to produce the desired result when using functions. MsgBox is one such function. It has three mandatory parameters that must be used to make the function operate correctly as well as two optional parameters that do not have to be used. When the MsgBox function is run within an application the default behavior is that it will pop up a small window (i.e. message box) displaying a message that requires the user to click "OK" to close.

NOTE: Functions can also be created without any parameters. The arguments in this case are "void". An example would be somefunction().   In this case there are parenthesis that don't contain anything. The function would be called simply by referencing it: somefunction(). No additional information is required.

NOTE: Though this was not discussed the message box will actually stop the rest of your script from executing until it is either closed manually or with the timeout parameter. Therefore, be careful when using it so you don't stop an application from running through completion when a user is not there to close the box. One way to avoid that is by having it close itself with the timeout parameter.

Chapter 4 Section 3: Using our first program to incorporate concepts from prior chapters

Your confidence is no doubt swelling after you wrote you first program and ran it to completion. Now let's build on the first effort to incorporate variables and operators from chapters 2 and 3 respectively.

We already know what happens when we run the following line of code:

```
File  Edit  Search  View  Tools  Options  Language  Buffers  Help
1    #include <MsgBoxConstants.au3>
2    MsgBox($MB_OK,"My first program","Hello World"); this is a comment - good job
3
```

Now what if we were to create two variables: $firstMessage and $secondMessage and store the following information as follows?

```
File  Edit  Search  View  Tools  Options  Language  Buf
1 hello world.au3    2 hello world with variables.au3
1    $firstMessage = "Hello "
2    $secondMessage = "World"
```

How might we now combine those two strings to form our complete message and display it in a message box? If you guessed with an operator, specifically the ampersand that is used to join strings, you are correct! Here is what the modified program might look like with variables and operators:

Code Chapter 4 Example 2

```
File  Edit  Search  View  Tools  Options  Language  Buffers  Help
1 Example1.au3    2 Example2.au3
1    #include <MsgBoxConstants.au3>
2    $firstMessage = "Hello "
3    $secondMessage = "World"
4    MsgBox($MB_OK,"My second program",$firstMessage&$secondMessage)
```

Try entering that text and saving it as a separate file called "helloworld2.au3". If you run that program you will get the identical message to the first program: "Hello World". Note: the title of the window changed because we changed the second argument to "My second program". So why is it important to know this other approach that takes more coding to produce the same result? Remember that we can store anything in our variables. One possibility is that our program may have logic that changes the information of one of our messages so that when the message box appears it says something completely different (this would require additional code). The storage location is called a "variable" which suggests that it can change.   We will cover examples of how this occurs later on when we tackle conditions that impact the flow of our programs.

REMINDER: We can use variables instead of typing out the data that we intend to use later on in our programs. We can have variables that store strings (text) and join them with the ampersand operator.

Chapter 5: Conditional Statements

Conditional statements can be used to change the flow of your program. You can test to see if a condition exists and then depending on the answer follow a certain path. Let's think back to our voting example. We have already decided that if a person is greater than or equal to >= 18 then they can vote. We covered the operator >= but did not discuss how to apply the test with code. The answer is in the sentence we used to describe the test. "**If** a person is >=18 **then** they can vote". "If … Then … Else" is one of the conditional statements available in AutoIt. So how does it work? Let's take a look at the following example:

Code Chapter 5 Example 1

```
File   Edit   Search   View   Tools   Options   Language   Buffers   Help

1      #include <MsgBoxConstants.au3>
2      $age = 17
3      if $age>=18 Then
4          MsgBox($MB_OK,"Voting Answer","The person can vote")
5      Else
6          MsgBox($MB_OK,"Voting Answer","The person cannot vote")
7      EndIf
```

If you want to type out this code and save it as vote.au3 you can. Then you can run it. When you do you will see that if $age=17 you will get the second message box stating they cannot vote. When you run the program above you should see the following output:

Voting Answer

The person cannot vote

OK

As you can see, the program selected the appropriate response based on the person's age. However, if you set the age to 18 or higher you will get the first message box stating they can vote. Try it out. Now let's cover *why* the program behaved that way. The "If… Then … Else" statement allowed to create a conditional statement that tested for whether or not our age variable was greater than or equal to 18.

The statement allowed for two different paths that would trigger two different messages depending on the answer

.

1.) On this line we have created a variable called $age that we are using to store the user's age which we have set to 17.
2.) On this line we have created the start of our If … Then … Else conditional statement to evaluate whether or not $age is >= 18.
3.) This line is a catchall test. It is saying "for everything else, do this". In our case, everything else means that age is <18 since if it were >=18 it would have triggered the first test.
4.) We have to close our If … Then … Else statements by telling the program it is the end of our test with EndIf.

Using the above program we can ensure the person is eligible to vote based on their age. What if we also wanted to test to make sure that they were a U.S. citizen? In other words, our requirement is that they are at least 18 years of age AND a U.S. citizen. We could express that in code using our old friend the operator. We did not cover the "AND" operator earlier but it is a logical test that we can use to ensure multiple conditions are satisfied. In this case we can test to see if $age>=18 AND $nationality="US". The code would be altered as follows:

Code Chapter 5 Example 2

```autoit
#include <MsgBoxConstants.au3>
$age = 18
$nationality = "Canadian"

if $age>= 18 and $nationality = "US" then
        MsgBox($MB_OK,"Voting Answer","The person can vote")
Else
        MsgBox($MB_OK,"Voting Answer","The person cannot vote")
EndIf
```

If you run this example you will see that the person is not eligible to vote based on their nationality even though they are 18 years of age. Now try changing the nationality from "Canadian" to "US" and you will the output showing that they can vote.

Another way to test for multiple conditions using is a conditional statement is by "nesting" them. That means you can place one conditional statement inside another. Let's consider our voting example. What if we changed the logic to look like this?

Code Chapter 5 Example 3

```autoit
#include <MsgBoxConstants.au3>
$age = 18
$nationality = "Canadian"

if $nationality = "US" then
    if $age>=18 then
        MsgBox($MB_OK,"Voting Answer","The person can vote")
    Else
        MsgBox($MB_OK,"Voting Answer","The person cannot vote")
    EndIf
Else
    MsgBox($MB_OK,"","Sorry. You are not a US citizen.")
EndIf
```

This produces a similar outcome but it gets to the answer in a different way. Note that there are actually two independent tests. The first is testing for nationality. If the person is not a U.S. citizen it will skip over the age test and output the last message: "Sorry. You are not a US citizen". However, if you change the citizenship to "US" then and only then will it check the age requirement.

When it checks for the person's age it will use the second If … Then … Else statement that is "nested" within the US citizenship test.

NOTE: in the nested example there are two complete sets of statements starting with If .. Then and the closing with EndIf. If you nest statements you will need to close the nested statement(s) and the outer statements in which they are nested. Also, you can continue nesting multiple times so that you have several more nested conditions. Alternatively, you may want to use the Select … Case or Switch … Case conditional statements. They are detailed within the AutoIt help file.

NOTE: Certain words are considered "keywords" because they are reserved by the AutoIt programming language. They are used to perform various tasks. "If, Then, Else, EndIF" are all examples. There is a complete keyword reference in the help file.

With that understanding of the If… Then… Else in hand it's time to "***Switch***" things up a bit. That's a bit of programming humor for you because **Switch** is a keyword also used for conditional statements.

The AutoIt help file shows the syntax and parameters for the switch:

Switch...Case...EndSwitch

Conditionally run statements.

```
Switch <expression>
    Case <value> [To <value>] [,<value> [To <value>] ...]
    statement1
    ...
    [Case <value> [To <value>] [,<value> [To <value>] ...]
    statement2
    ...]
    [Case Else
    statementN
    ...]
EndSwitch
```

Parameters

<expression>	An expression that returns a value. The value from the expression is then compared against the values of each case until a match is found. This expression is always evaluated exactly once each time through the structure.
<value> To <value>	The case is executed if the expression is between the two values.
<value>	The case is executed if the expression matches the value.

The switch will test an expression to see if it matches any of the cases. If it does, it will run the code contained in the statement. Here is a small example:

Code Chapter 5 Example 4

```
File   Edit   Search   View   Tools   Options   Language   Buffers   Help

 1      #include <MsgBoxConstants.au3>
 2      $expression = 5
 3
 4      Switch $expression
 5          Case 1
 6              $sMsg = "The value is 1"
 7          Case 2
 8              $sMsg = "The value is 2"
 9          Case 3
10              $sMsg = "The value is 3"
11          Case Else
12              $sMsg = "The value is something other than 1,2,or 3"
13      EndSwitch
14
15      MsgBox($MB_OK, "", $sMsg)
```

In this example we are testing the expression contained in our variable with the same name. The script will set a message to different values for different cases. The cases we have setup can be either 1, 2, 3 or 'else'. That means we will see a different message if the value of $expression is 1, 2, 3, or something else. In this case, we have set the variable $expression to the number 5 so we would see the following:

The value is something other than 1,2,or 3

OK
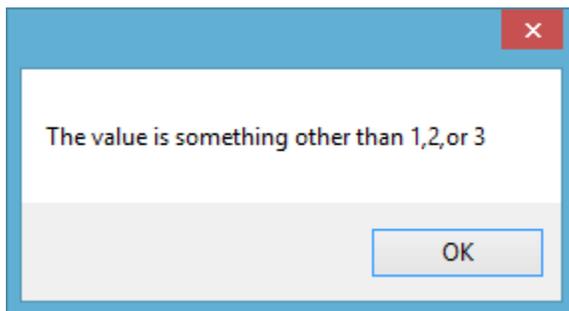
REMINDER: Conditional statements can be used to change the flow of your program. You can test for certain conditions and have your program respond different ways depending on the outcome. The If … Then … Else is a staple of conditional statements. It's user friendly syntax allows you to test to see "**If**" a condition is true and "**Then**" take the appropriate action "**Else**" do something different. We can combine our use of operators with conditional statements to assist with creating the condition we are testing. We can also If … Then … Else statements within each other.

Chapter 6: Do That Again. Understanding Loops

As you start to write more programs you will invariably need to do something repeatedly. Consider the conditional statements from the previous chapter. In that example we test to see if someone was a U.S. citizen and at least 18 years of age to determine if they were eligible to vote. However, we did it only for a single pair of values for $age and $nationality that we coded into our program. What if instead, we had a list of 1,000 values and we wanted to test them all? Writing out a program 1,000 times wouldn't be any faster than looking at the list row by row (in fact it would likely be much slower).   For this type of task we have loops. Loops are statements that repeat for a certain number of times or until a specific condition is satisfied.

To properly demonstrate a loop I am going to introduce you to another pre-built AutoIt function: "ConsoleWrite". ConsoleWrite will show information in the bottom of our SciTE window (the console) when our application runs. It has only one parameter called "data" which is whatever you want to send to the console. Therefore, the function looks like this: ConsoleWrite("data"). Let' start of by sending a simple piece of text to the console and then we can introduce some loops.

Code Chapter 6 Example 1
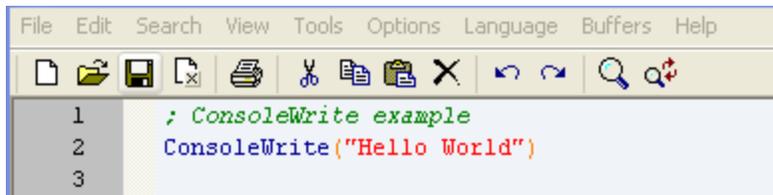
```
1    ; ConsoleWrite example
2    ConsoleWrite("Hello World")
3
```

If you save this code snippet as "consolewrite.au3" you will see the following appear at the bottom of the SciTE window:

```
File  Edit  Search  View  Tools  Options  Language  Buffers  Help

1        ; ConsoleWrite example
2        ConsoleWrite("Hello World")
3        |

>"C:\Program Files\AutoIt3\SciTE\AutoIt3Wrapper\AutoIt3Wrapper.exe" /run
+>07:20:18 Starting AutoIt3Wrapper v.2.1.3.0 SciTE v.3.3.6.0 ;  Keyboard:
>Running AU3Check (1.54.22.0)  from:C:\Program Files\AutoIt3
+>07:20:18 AU3Check ended.rc:0
>Running:(3.3.8.1):C:\Program Files\AutoIt3\autoit3.exe "C:\Documents and
--> Press Ctrl+Alt+F5 to Restart or Ctrl+Break to Stop
Hello World+>
>Exit code: 0    Time: 1.188

li=3 co=1 INS (CR+LF)
```

Note: The output has a bunch of information about how long it took to run your program, syntax checks that were performed, etc. However, you can also see the output from our program next to the arrow. It is somewhat hard to read because it is not on its own line. To rectify this we could add a carriage return and make it easier to read by using a ***macro***.   Macros are predefined read-only variables (so you can't change their values) that offer easy access to various system resources. Think of them as shortcuts. The macro shortcut for a hard carriage return is @CRLF.   The first two letters are for "carriage return" and the last two letters are for "line feed". Together, they create a "hard return" at the end of your text. Let's modify our program to include the @CRLF macro and see what happens to our output. Type the following into SciTE and save it somewhere:

Code Chapter 6 Example 2


```
File  Edit  Search  View  Tools  Options  Language  Buffers  Help

1        ; ConsoleWrite example
2        ConsoleWrite("Hello World"&@CRLF)
3        |
```

NOTE: we are adding the ampersand operator to join our text with the hard return. The script would not run properly without the ampersand because "Hello World" is its own string (text within quotations) and the @CRLF macro would be treated separately if we did not join them together. The function expects only one argument for the single parameter. As such, the program would not understand additional text without us signaling that the two should be combined into a single string with the ampersand operator.

If you run the modified script your output should appear as follows:

You can see that the carriage return made our text appear on its own line with no text after it.

Now we will take the program one step further and use a *loop* to repeat the process. Let's start with the "for" loop. The AutoIt help file explains the syntax (i.e. format) for how to formulate a "for" loop:

For...To...Step...Next

Loop based on an expression.

```
For <variable> = <start> To <stop> [Step <stepval>]
    statements
    ...
Next
```

Parameters

variable	The variable used for the count.
start	The initial numeric value of the variable.
stop	The final numeric value of the variable.
stepval	[optional] The numeric value (possibly fractional) that the count is increased by each loop. Default is 1.

The loop will repeat everything after the "For To Step" line until it reaches the end count. After the loop runs the designated number of times the script will continue to whatever follows "Next". The keyword "Next" is used to close the loop (just like EndIf in our conditional statements).

We covered the concept of keywords in the last chapter on conditional statements. "For, To, Step, and Next" are all keywords. To create a For loop we use the keyword "For" to signal that we are starting a loop. Then we assign a value to the start parameter to start our count for how many times we want to perform the desired actions within the loop.   Then specify a stopping point with the stop parameter that follows the keyword "To". We can optionally use a stepval parameter to state whether or not we are counting by some number other than 1 which is the default (i.e. 2 to count by twos, or -1 to count backwards). The value of our variable will increase (or decrease) by the interval each time the loop runs. If we don't specify anything our variable will increase by 1 each time (that is the default as shown in the help file).

Let's modify our code and test it out:

Code   Chapter 6 Example 3
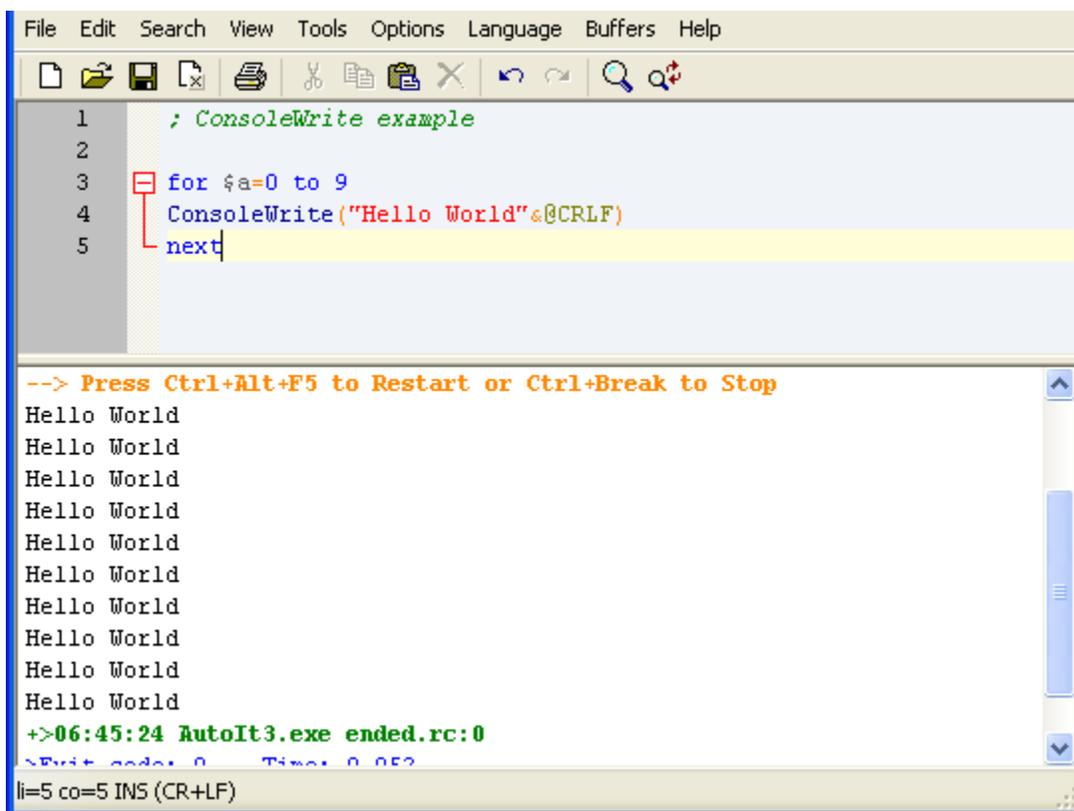
```
File  Edit  Search  View  Tools  Options  Language  Buffers  Help

1        ; ConsoleWrite example
2
3    ┌  for $a=0 to 9
4    │     ConsoleWrite("Hello World"&@CRLF)
5    └  next

--> Press Ctrl+Alt+F5 to Restart or Ctrl+Break to Stop
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
+>06:45:24 AutoIt3.exe ended.rc:0
>Exit code: 0     Time: 0.053

li=5 co=5 INS (CR+LF)
```

Note, in the above example we started with a variable called $a and assigned it a value of zero. We then told it to repeat the loop until $a was equal to 9. The $a variable automatically increases by 1 by default. That means each time the operation within the loop is performed the value of $a

increases until it reaches the stop number (in this case 9). How many times did we print out "Hello World"? The answer is 10 … not 9. This is because we started at 0 and counted by ones until we got to 9. That made the loop execute 10 times.

NOTE: Many programmers make the mistake of incorrectly counting the number of times their scripts will run a loop because they forget that they used a zero base number from which to start the count. You can avoid that by being careful or by assigning the value 1 to the variable in the counter. For reasons we will cover later when we discuss arrays it is better to become familiar with the zero number as the starting point.

We can see the value assigned to $a increasing if we change our code a bit. This time, instead of printing out the same message over and over again we can write a value that is different each time. To do this we will write to the console a short text message (string) joined with an ampersand operator to our variable $a joined with another ampersand operator to a carriage return macro. The output of the modified program appears in the console below the code. Now we have joined

the variable used in our loop to the data displayed in the console. By doing this we can see it increasing each time the loop is run. We then joined the carriage return to the end of the data to make sure each console message appeared on its own line.

Code Chapter 6 Example 4

```
File  Edit  Search  View  Tools  Options  Language  Buffers  Help

1        ; ConsoleWrite example
2
3      for $a=0 to 9
4      ConsoleWrite("This is a: " & $a &@CRLF)
5      next


This is a: 0
This is a: 1
This is a: 2
This is a: 3
This is a: 4
This is a: 5
This is a: 6
This is a: 7
This is a: 8
This is a: 9
+>06:58:09 AutoIt3.exe ended.rc:0
>Exit code: 0    Time: 0.504
li=4 co=39 INS (CR+LF)
```

What would we do if we wanted to countdown from 9? To do that we could modify the code as follows:

NOTE: In this modified example we are starting the value of $a higher than the stop value and using the optional interval parameter to instruct the script to subtract 1 from the value assigned to $a each time it runs the loop.   Note the difference in the output.

```
File  Edit  Search  View  Tools  Options  Language  Buffers  Help

1        ; ConsoleWrite example
2
3        for $a=9 to 0 step -1
4          ConsoleWrite("This is a: " & $a &@CRLF)
5        next


This is a: 9
This is a: 8
This is a: 7
This is a: 6
This is a: 5
This is a: 4
This is a: 3
This is a: 2
This is a: 1
This is a: 0
+>07:02:49 AutoIt3.exe ended.rc:0
>Exit code: 0    Time: 0.514

li=3 co=22 INS (CR+LF)
```

Another really important tidbit about For loops is that just like IF … Then … Else conditional statements they can be nested. That means you can put one or more loops inside your outermost loop. This is very powerful but we warned – it can also quickly create confusion as you need to remember what each loop does and how many times it does it. For that reason, I like to comment the "For" line of each loop to remind myself what each loop is doing. Also, we want to make sure we use different variable names for each variable in each separate loop. If the first one was $a we may want to make the second one $b. Alternatively, we may want to use a more meaningful name as is the best practice when dealing with variables in general.   Let's explore a simple example of a nested For loop:

Code
Chapter 6 Example 5

Note: the inner loop is indented to make it easier to read. This loop will run everything in it twice: once for 0 and once for 1.

Will run outer loop twice

```
1    ; ConsoleWrite example
2
3    $loops = 1
4    for $a=0 to 1
5        for $b = 0 to 9
6            ConsoleWrite("This is a: " & $b &@CRLF)
7        next
8        ConsoleWrite("Inner loop:"&$loops& " set finsished"&@CRLF)
9        $loops+=1
10   next
```

This will occur 10 times just like in our first script but it will be run to completion the number of times specified by the outer-loop. In this case twice.

```
This is a: 0
This is a: 1
This is a: 2
This is a: 3
This is a: 4
This is a: 5
This is a: 6
This is a: 7
This is a: 8
This is a: 9
Inner loop:1 set finsished
This is a: 0
This is a: 1
This is a: 2
This is a: 3
This is a: 4
This is a: 5
This is a: 6
This is a: 7
This is a: 8
This is a: 9
Inner loop:2 set finsished
```
li=8 co=30 INS (CR+LF)

When the inner loop completes it goes to whatever follows Next before the outer loop kicks in again. In this case we have a special message telling us the inner loop is done.

In this example several things are happening: We have introduced a new variable called $loops to which we assigned a value of 1. We will use it to keep track of the number of times we execute the outermost loop. Then we created an outer loop that will run twice (For $a=0 to 1). We then nested the loop we were using earlier counting from 0 to 9 inside the outer loop (but we changed the starting count variable from $a to $b so as to avoid confusion with the outer loop). Finally, we added two lines of code:

ConsoleWrite("Inner loop:"&$loops& " set finsished"&@CRLF), and

$loops+=1

The first line of code is creating a new message that we will write to the console every time the inner loop completes. We are using the $loops variable we created to tell us how many times the inner loop has run.   The next line of code uses a special operator "+=" which adds the number following the "=" to the $loops variable. Therefore, since $loops started with a value of 1 when we say $loops+=1 the value of loops becomes 2.

NOTE: Loops can introduce a lot of power into your scripts – but they can also introduce complexity. If you start to get overwhelmed try breaking your script down into pieces and testing each piece separately before combining them (that goes for any code – not just loops).

The next type of a loop is another powerful way to repeat instructions. However, it can be very dangerous if used incorrectly. It is called the "while" loop. It tests for a certain condition and repeats the loop "while" the condition is true. It is dangerous because if loop never has a condition that sets it to false it will keep running in an "infinite loop". That means that the script will get stuck running the same lines of code over and over again forever (or until you forcefully close it). On the other hand, sometimes this will be your intention. You may have need to create a loop that is open as long as you program is running. A common example is when you show a user interface (i.e. an application with controls like buttons, text boxes, menus, etc) which we will over later. You would not want to flash the controls for a split second only to have them disappear before a user could interact with them. In that case, "while" can be employed to keep the application open. More on that later when we cover interfaces.

The help file explains the syntax for this loop as follows:

While...WEnd

Loop based on an expression.

```
While <expression>
    statements
    ...
WEnd
```

Let's test out a loop with a simple example building on some of the concepts we have already used. In this example we will create a variable called $a and assign it 0 ($a=0). Then we will create a while loop that writes a message to the console as long as $a is less than 10. We must also remember to increase the value of $a with every pass or we will be stuck in an infinite loop. The code and the output for this example appear below:
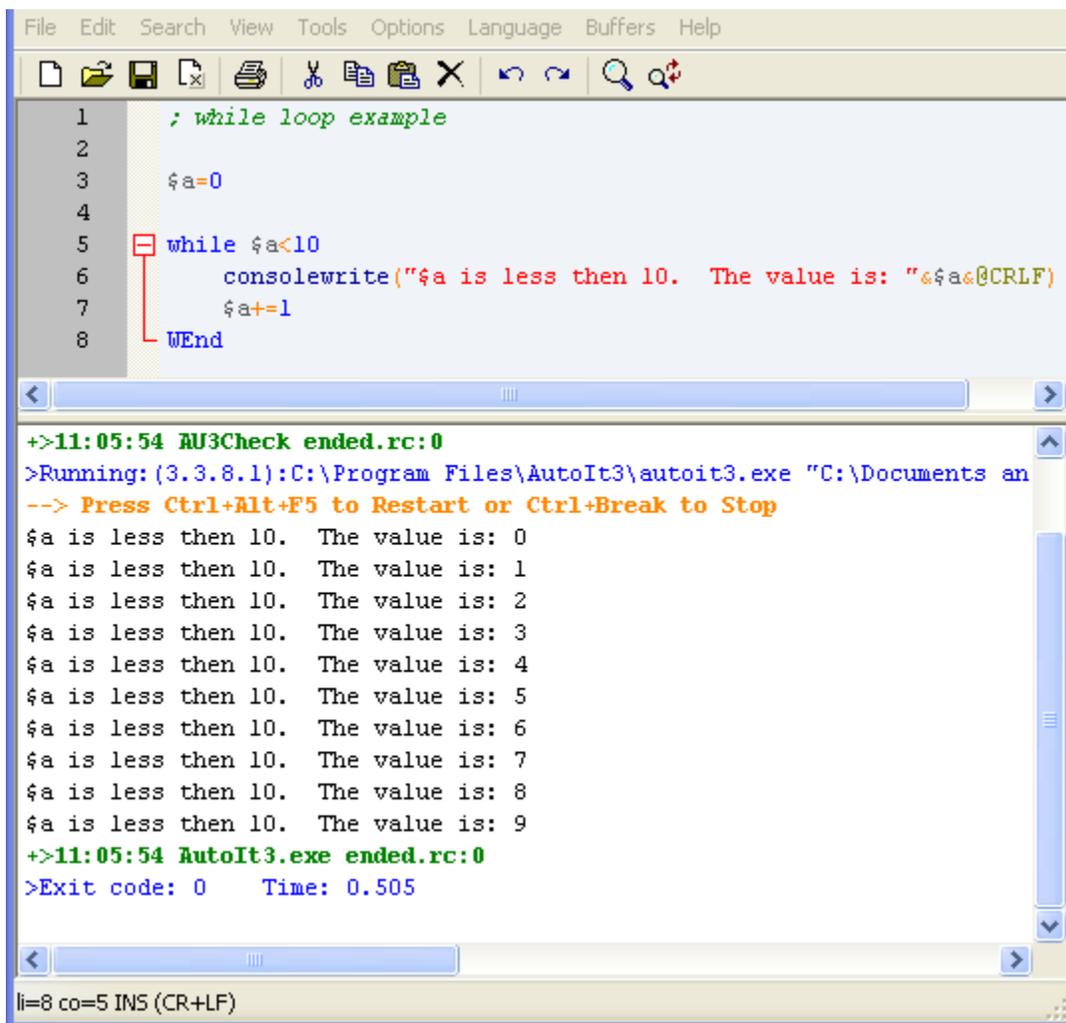
Code Chapter 6 Example 6

```autoit
; while loop example

$a=0

while $a<10
    consolewrite("$a is less then 10.  The value is: "&$a&@CRLF)
    $a+=1
WEnd
```

```
+>11:05:54 AU3Check ended.rc:0
>Running:(3.3.8.1):C:\Program Files\AutoIt3\autoit3.exe "C:\Documents an
--> Press Ctrl+Alt+F5 to Restart or Ctrl+Break to Stop
$a is less then 10.  The value is: 0
$a is less then 10.  The value is: 1
$a is less then 10.  The value is: 2
$a is less then 10.  The value is: 3
$a is less then 10.  The value is: 4
$a is less then 10.  The value is: 5
$a is less then 10.  The value is: 6
$a is less then 10.  The value is: 7
$a is less then 10.  The value is: 8
$a is less then 10.  The value is: 9
+>11:05:54 AutoIt3.exe ended.rc:0
>Exit code: 0    Time: 0.505
```

li=8 co=5 INS (CR+LF)

NOTE : The expression is tested **before** the loop is executed. That is why we see it stop at 9. On the 11th pass $a would have been 10 would no longer be < 10. Therefore, the test would be false and the loop would end. Also, it is important to note that you can also nest While loops.

You know what I am going to Do? I am going to "*Do*" loops "*Until*" we finish the topic. That's right, Do … Until is another type of loop statement. The AutoIt help file explains the syntax as follows:

Do...Until

Loop based on an expression.

```
Do
    statements
    ...
Until <expression>
```

Did you know that almost every topic in the help file comes complete with its own code example? Not only that, but you can usually open the code up with the push of a button and test it out. Let's take a look at the code example for Do … Until:

Example

```
Local $i = 0
Do
    MsgBox(0, "Value of $i is:", $i)
    $i = $i + 1
Until $i = 10
```

Open this Script ← Lets you pop open the example script and test it out.

The script should open up in SciTE when you click the button:

```
File  Edit  Search  View  Tools  Options  Language  Buffers  Help

1      Local $i = 0
2   ⊟ Do
3          MsgBox(0, "Value of $i is:", $i)
4          $i = $i + 1
5      Until $i = 10
6
li=1 co=1 INS (CR+LF) R
```
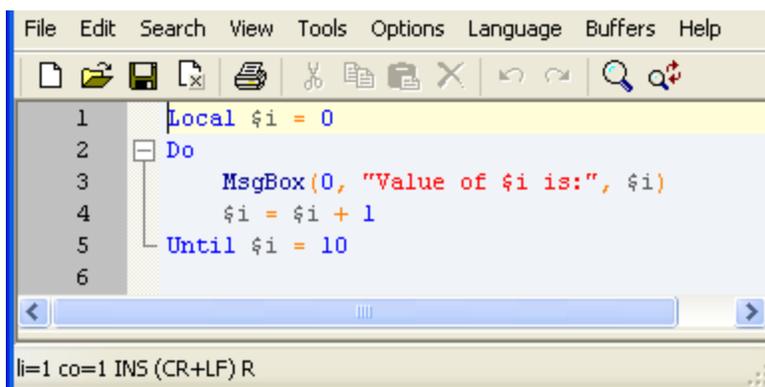
From there, you can run it to see what happens. You will note that in this case we will see message boxes appear that must be closed until the value of $i is equal to 10. What is the last value that you see in the message box as you click through them? It should be 9. Why not 10? The answer is that it stops once the condition on the "until" line has been satisfied. In this case line 3 creates a

message box displaying the current value of $i while line 4 adds 1 to the value of $i and then adds it back to itself (another way to do this is $i+=1 as we saw in the previous section). Therefore, the value of $i increases **after** each message box. As a result when the message box display 9 as the value of $i the very next thing to happen is that $i is increased to 10 at which point the loop stops because it only goes **until** $i = 10.

It is important to note that the condition the Do … Until loop is evaluating is tested after the loop is run so the loop will be executed at least one time. This is different from the while loop which tests before the condition is evaluated and, therefore, is run zero or more times.

REMINDER: Loops can be used to help us repeat actions for a certain number of times or until a condition is satisfied. The For loop uses a variable to which we assign a starting value for the loop count. Then we specify "To" what number we would like to count to (up or down). Finally, if we desire to count by something other than +1 we can use the optional interval parameter to express the value by which our starting variable will change (i.e. 2, 5, 10, -1, -3 etc). The Next keyword signals that we are done supplying commands that we would like to run inside our For loop. While loops test for a condition before executing the loop. As long as the condition is true the loop will continue. Be careful not to create "infinite loops" (unless that is your intention) by ensuring you will eventually get a false result and stop the loop. While loops test for the condition before the loop runs so they will run zero or more times. "Do … Until" performs a loop until a condition is met. It tests for the condition after it runs so it runs at least one time.

Chapter 7: Custom Functions

We have already seen some examples of functions that are built-in to AuotIt.   In Chapter 4 we were introduced to the MsgBox function that we used in our first program to display "Hello World" and in Chapter 6 we used ConsoleWrite to output the results of our loops to the screen within our SciTE editor. Those functions are ready for use without any additional coding from us. Think of them as pre-packaged lines of code that can be called or used from within our script simply by referring to them by name and supplying the necessary arguments if required (information required by the function – though recall some don't require any arguments).   Most (if not all) programming languages also let you write your own functions. AutoIt uses the keywords "Func" and "EndFunc" to create a user defined function as follows:

```
1    ; An example of a user function
2
3    Func _myFunction()
4    ;Our code will go here
5
6    EndFunc
```

In this example we are using the keyword Func to signal that we are creating a user defined function. Then we are naming it "_myFunction()".

NOTE : It is not required but rather recommended that as a "best practice" you start a user defined function name with the underscore "_" to differentiate it from prebuilt functions. Also, note the way the name starts off with a lower case "m" and the second word is capitalized. It is a common practice to make the names of your variables and functions easier to read my starting with lower case and separating each word with the use of capital letters. The parenthesis that follow the name of our function is where we would place any parameters that we wanted to use within the function. This function is void of parameters so it is just an empty parenthesis. That means no additional information is required to run the function (i.e. no parameters as in the case of the pre-built MsgBox and ConsoleWrite functions we used earlier.

The code above won't actually do anything for two reasons. First, we have only created the user defined function – we have not called upon it to do anything. Second, there is no code within the function so even if we were to call it nothing would happen. Let's change the example slightly by adding some code and calling the function:

Code Chapter 7 Example 1

In the above example we have added a line of code within the body of our function to create another Hello World message box. Then we added a line of code after the function to run the function. Now every time that we want to create the message box with that message all we have to do is call our user-defined function "_myFunction()" instead of typing out msgbox($MB_OK,"User Function","Hello World!"). That is not a huge savings of time but it can be when you consider that you can place as many lines of code as you want within your function. Let's try another example …

Code Chapter 7 Example 2

```
1       ; An example of a user function
2       #include <MsgBoxConstants.au3>
3
4       Func _mathWithAnswers($firstNum,$secondNum,$operator)
5
6       Switch $operator
7           case $operator = "+"
8               $answer=$firstNum+$secondNum
9           case $operator = "-"
10              $answer=$firstNum-$secondNum
11          case $operator = "*"
12              $answer=$firstNum*$secondNum
13          case $operator = "/"
14      EndSwitch
15      MsgBox($MB_OK,"Answer",$answer)
16      EndFunc
17
18      _mathWithAnswers(5,3,"+")
```

The above example builds upon several things we have already learned. It may look somewhat complex at first blush – but if we review what is happening you will see it is pretty straightforward. This user defined function acts as a calculator that can add, subtract, multiply or divide any two numbers. The function takes in three parameters: $firstNum, $secondNum, and $operator. These are for the first number, second number, and mathematical operators respectively. Inside the function we have added a conditional statement in the form of a switch that will change the flow of our script based on the operator that we pass as the third parameter. In other words, if we say we want to add we would need to pass "+" whereas if we want to multiply we would pass the "*". The operator parameter is not really the actual operator but rather a string (text) representation of it (because it is surrounded by quotes). The switch then tests the value of operator argument to see which mathematical operation should be used with the real operator (not in quotes). The $answer variable stores the results for us and the message box displays those results. Finally, when we call the function we need to supply all three parameters. In this case we have used (5,3,"+") which means we want to add 5+3. Not surprisingly, the script will produce the following output:

Answer

8

OK
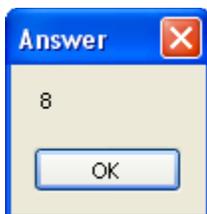
This function could come in handy for doing some simple math and displaying the results in a message box. However, there is at least one problem with it. The function will do whatever we tell it to – nothing more and nothing less. If we allow it do things that are inadvisable or even impossible it could crash the script or generate other unforeseen bugs. What if we called the function as follows: _mathWithAnswers(5,0,"/")? What would that do? We would be trying to divide 5 by 0. That is a no-no. In this case, the consequence would be trivial. We will see an error in the message box. However, what if we were relying on the answer for another part of our script? You get the idea. To guard against that possibility we can use a conditional operator and modify our program to block our undesirable scenarios (that we can foresee ahead of time). In this case, we might try something like this:

Code Chapter 7 Example 3

```
File  Edit  Search  View  Tools  Options  Language  Buffers  Help

1         ; An example of a user function
2         #include <MsgBoxConstants.au3>
3
4         Func _mathWithAnswers($firstNum,$secondNum,$operator)
5
6         Switch $operator
7             case $operator = "+"
8                 $answer=$firstNum+$secondNum
9             case $operator = "-"
10                $answer=$firstNum-$secondNum
11            case $operator = "*"
12                $answer=$firstNum*$secondNum
13            case $operator = "/"
14                if $secondNum = 0 Then
15                    MsgBox($MB_OK,"Error","You cannot divide by zero")
16                else
17                    $answer=$firstNum/$secondNum
18                EndIf
19        EndSwitch
20        MsgBox($MB_OK,"Answer",$answer)
21        EndFunc
22
23        _mathWithAnswers(5,0,"/")
```

> If the second number is zero tell them they can't do that.

> Otherwise, let them divide the second number.

The modifications in the above code "trap" the error that we were concerned about and take it down another path. Specifically, if the second number is zero then instead of doing the division it will explain to the user that they can't divide by zero … otherwise, it will proceed. You can see we used our old friend If … Then … Else to accomplish this new error trapping functionality.

Chapter 7 Section 2: Variable Scope

When we first started talking about variables in Chapter 2 I intentionally skipped some information about their usage. Specifically, variables also have an attribute called scope. This dictates where within the script they can be used and recognized. The scope can be either local or global. I did not cover it earlier because the function is the area that creates the greatest opportunity for differentiation – and

we just covered functions. This is because a global variable will be recognized by the entire script (inside and outside of functions).   By contrast, a local variable can only be seen within the function that uses it unless: (a) we force it to be global in scope by using the global keyword or (b) a global variable of the same name exists (in that case we can still force it be local using the keyword local). On the other hand, a variable that is declared outside of the function is global in scope.   The AutoIt help file summarizes these situations as follows:

Scope

A variable's scope is controlled by when and how you declare the variable. If you declare a variable at the start of your script and outside any functions it exists in the **Global** scope and can be read or changed from anywhere in the script.

If you declare a variable *inside* a [function](#) it is in **Local** scope and can only be used *within that same function.* Variables created inside functions are automatically destroyed when the function ends.

By default when variables are declared using [Dim](#) or assigned in a function they have **Local** scope **unless** there is a global variable of the same name (in which case the global variable is reused). This can be altered by using the [Local](#) and [Global](#) keywords to declare variables and **force** the scope you want.

The help file section above that addresses variable scope also references the keyword **Dim**. The use of this keyword is still supported to declare variables. However, best practice is to use either the local or global scopes instead.   Side note: Dim is a funny key word. So far many of the keywords we have encountered have been somewhat descriptive of their usage. The origin of Dim morphed from the longer term "dimension" which was originally used to declare the dimension of an array of values (i.e. an "Array") which we will cover in the next chapter. It was shortened to Dim and applied to all variables – and arrays.

REMINDER: AutoIt has many pre-built functions that can quickly add power to your scripts. They need to be called with the appropriate number of parameters as specified in the help file. However, you can also create your own functions with the keywords Func and EndFunc. You can also name your function whatever you want although best practices dictate that it start with an underscore ("_") to differentiate it from the pre-built functions. Once your function is written you can call it from within your script as many times as you like which, just like the pre-built functions, can reduce the amount of code you have to write. It is also a great way to organize your code for testing and debugging one section at a time. For that reason, try to avoid enormously long functions and break them into smaller pieces when possible. Variables can be local to a function or globally accessed from anywhere in the script. Declaring a variable outside a function makes it global by default. Declaring it within a function

makes it private unless you use the keyword "global" or there is already a global variable of the same name outside of the function. You can also force the variable to be local with the keyword local. Finally, while you don't have to declare variables you can optionally do so with using either local or global to define their scope.

Chapter 8: Arrays

You will see reference to the term "array" in the AutoIt help file. So what is an array? Think of them as variables that hold more than one piece of information.   More precisely, an array is a data structure that allows you to hold numerous data elements that you can access by referencing an index to their position. For example, you may want to store three names in your array. To do so you could say:

dim $namesArray[3]=["John","Steve","Bob"]

In this case, we are holding three items so we might say the upper boundary is 3. That value must be specified within brackets following the name of our array. Brackets must also be used to hold the items we are storing in the array and those items must be separated by commas. The Array starts with a zero index so although there are three values the indices are: 0, 1, and 2 for those three items. Therefore, if you want to reference Steve you would say $namesArray[1]. Bob would be $namesArray[2]. If you try to reference an index that is not part of the specified range you will get an error. This is what it might look like in practice when we reference the second item of the above array:

Code Chapter 8 Example 1

```
1    #include <MsgBoxConstants.au3>
2    global $namesArray[3]=["John","Steve","Bob"]
3    MsgBox($MB_OK,"Who is second in the array?",  "Answer: "&$namesArray[1])
4
```

Who is second in the array? ×

Answer: Steve

OK
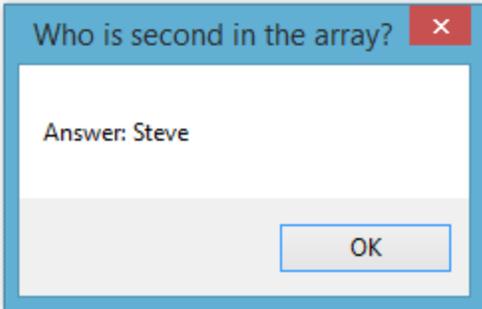
You can see that by referencing the index position of 1 we see "Steve" even though Steve is the second entry. This is a consequence of the zero based index.

NOTE: Although the boundary of the array is specified when it was created i.e. the "3" in the $namesArray[3] - the upper most index in the array is 2. This is because we have zero based index. It is

advised to keep all the elements of the array the same datatype. As you may recall from Chapter 1 AutoIt only has one official datatype called variant. However, from the variables usage it AutoIt will assign an internal datatype of string, integer, double, etc. In our example all three values are of the internal datatype string (meaning text within quotes).

Arrays can also be multi-dimensional. That means that you can have an array that looks like this:

dim $people[3][2] = [["John",34],["Steve",23],["Bob",41]]

Now we have three entries with 2 pieces of information per entry: name and age. To get Steve's name and age we would do something that resembles the following:

Code Chapter 8 Example 2

```
1    #include <MsgBoxConstants.au3>
2    dim $people[3][2] = [["John",34],["Steve",23],["Bob",41]]
3    MsgBox($MB_OK,"",$people[1][0]&" is "&$people[1][1]&" years of age")
4
```

Steve is 23 years of age

OK
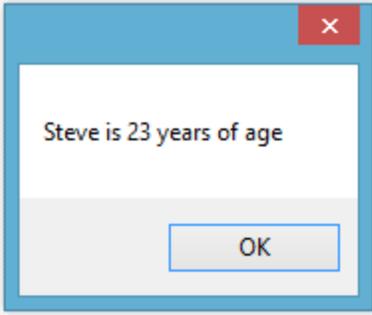
Notice the way we declare our array and access the information within has changed slightly because we now have a two dimensional array. The boundaries of each dimension are 3 and 2 respectively; three items with two pieces of information per item. The brackets used to open and close the array [] encapsulate the entire array as well as each pairing. Then when we reference the data we must use both dimensions. We already know that to reference a specific person our choices are 0, 1, or 2 because there are three entries with a zero based index. That has not changed. However, since there are two pieces of information concerning the person we are referencing we need to specify which one we are after: name or age. Since there are only two values in our example (name and age) we will always only ever reference 0 or 1. We will use 0 for name and 1 for age. Therefore, $people[1][0] is referring to the name in the second pair while $people[1][1] is referring to the age in the second pair. How about if we wanted to find Bob's name and age in the array? What would that look like? It would be $people[2][0] for Bob's name and $people[2][1] for his age (John would be name: $people[0][0] and age: $people[0][1]).

I realize that concept of creating a list of values and referencing them by an index number can seem confusing. Why would we do that? Your script may use a lot of complex data. The ability to store it, augment it, and access it quickly with a small amount of code is very powerful. Also, consider the loops

that we learned about earlier.   Think about how the "For … Next" loop uses a number that increases with each pass the loop executes.   Combining a loop with an array is a natural fit. Take our example we just covered and let's loop it:

Code Chapter 8 Example 3

```
1    #include <MsgBoxConstants.au3>
2    dim $people[3][2] = [["John",34],["Steve",23],["Bob",41]]
3
4    for $a=0 to UBound($people)-1
5        MsgBox($MB_OK,"",$people[$a][0]&" is "&$people[$a][1]&" years of age")
6    Next
```

$a is pulling double duty. It is increasing the loop and being used to reference the index of the array.

In the above example you will note that we are creating a loop that starts at zero (for $a=0). When does the loop end? When we covered loops earlier we specified a value to stop at 10, 17, 100, etc. or dictated that the loop stop when a condition was satisfied. Now we are using a pre-built function called UBound to tell us the stopping point which will be the upper boundary of the $people array less 1. The upper boundary is 3. Why do we then subtract 1 from that value to create the number of loops we want to perform? Because of the zero based array. The code will loop three times starting at 0: 0, 1, and 2. If we used the upper boundary it would go four times: 0, 1, 2, and 3; that would be a problem for us as we only have three pieces of data. If we let the script look for a fourth piece of data when there are only three we will get an error.

Now that we know the script will loop exactly three times let's take a look at what will happen inside the loop. As you can see, it is the exact same thing that was happening before with one change: now we don't have to reference which person we want to read about in our message box. The loop will show them all to us. It can do this because the value of $a is increasing by 1 with each pass. Therefore, we can use it to reference it when we create our message box to point to the appropriate index in the array like so:

MsgBox($MB_OK,"",$people[$a][0]&" is "&$people[$a][1]&" years of age")

The above line of code that uses the $a variable within the first dimension of the array to increase from 0 – 2 with each pass of the loop.   Since there can only be two values in the second dimension: name and age, we simply write 0 and 1 after the $a for the second dimension. When you run the code in this example you will see three message boxes showing John, Steve, and Bob along with their ages in the order they appear in the array.   The loop has the same impact as writing out the following three lines of code:

This is the equivalent of the image below with each pass of the loop:

```
1    MsgBox($MB_OK,"",$people[0][0]&" is "&$people[0][1]&" years of age"); pass 1
2    MsgBox($MB_OK,"",$people[1][0]&" is "&$people[1][1]&" years of age"); pass 2
3    MsgBox($MB_OK,"",$people[2][0]&" is "&$people[2][1]&" years of age"); pass 3
```

REMINDER: Arrays can hold numerous data elements and allow you to reference them by their index value.   Most arrays have an index that starts with zero unless otherwise specified so the index will not match the upper boundary of the array. Therefore, when looping through an array you should make sure you loop to the upper boundary less one. This can be done with the UBound function followed by the array of interest as an argument less 1: UBound($myArray)-1. Arrays can also be multi-dimensional. You can reference any value but be sure to provide all dimensions when doing so or you may get an error.

Chapter 9: Graphical User Interfaces (GUIs)

Up to this point we have learned about data, operators, conditions, and loops. Those are the basic building blocks for almost any application. In fact, with those concepts in hand you could go off and start to write your own scripts. However, sometimes you may want users to interact with your scripts. One of the most popular ways to accomplish this is through the use of a Graphical User Interface or GUI. GUIs contain various controls like buttons, areas to input text, dropdown menus etc. An example GUI from the AutoIt help file is shown below.
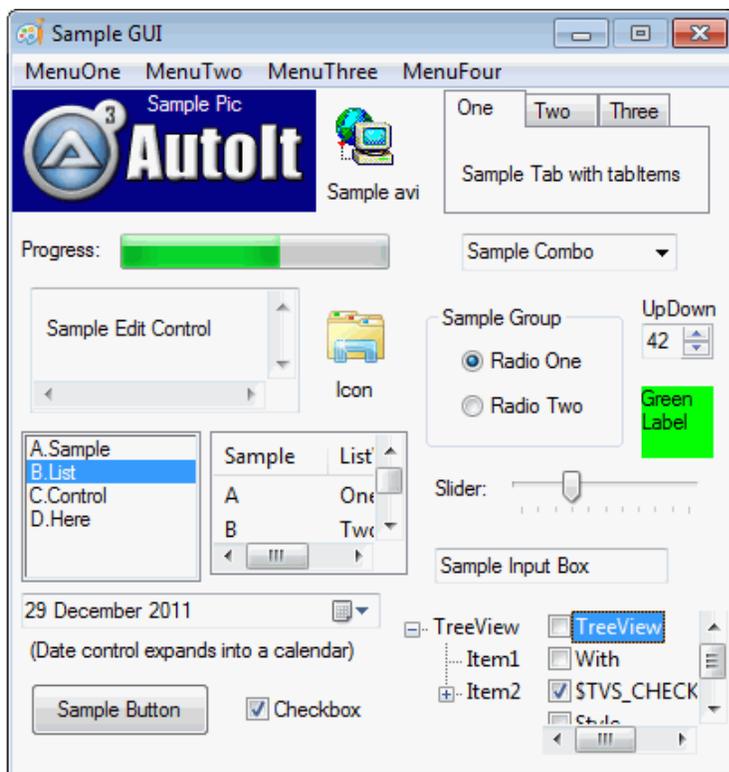
From the AutoIt help file under "GUI Concepts"

GUI / Control Layout

Before we get too far into GUI creation we should understand the coordinate system used to place GUI controls. In other words, how do we tell our program where precisely to place our button, input field, etc.? The answer is by using an X,Y coordinate that is very similar to basic graph plotting that you may have done in grade school. The following image illustrates this with a 300 x 300 GUI that does not yet contain any controls. Knowing the size of our GUI we could quickly determine that the center point is X=150, y=150 and so on. We can use any X,Y point within the 300x300 to place our controls.
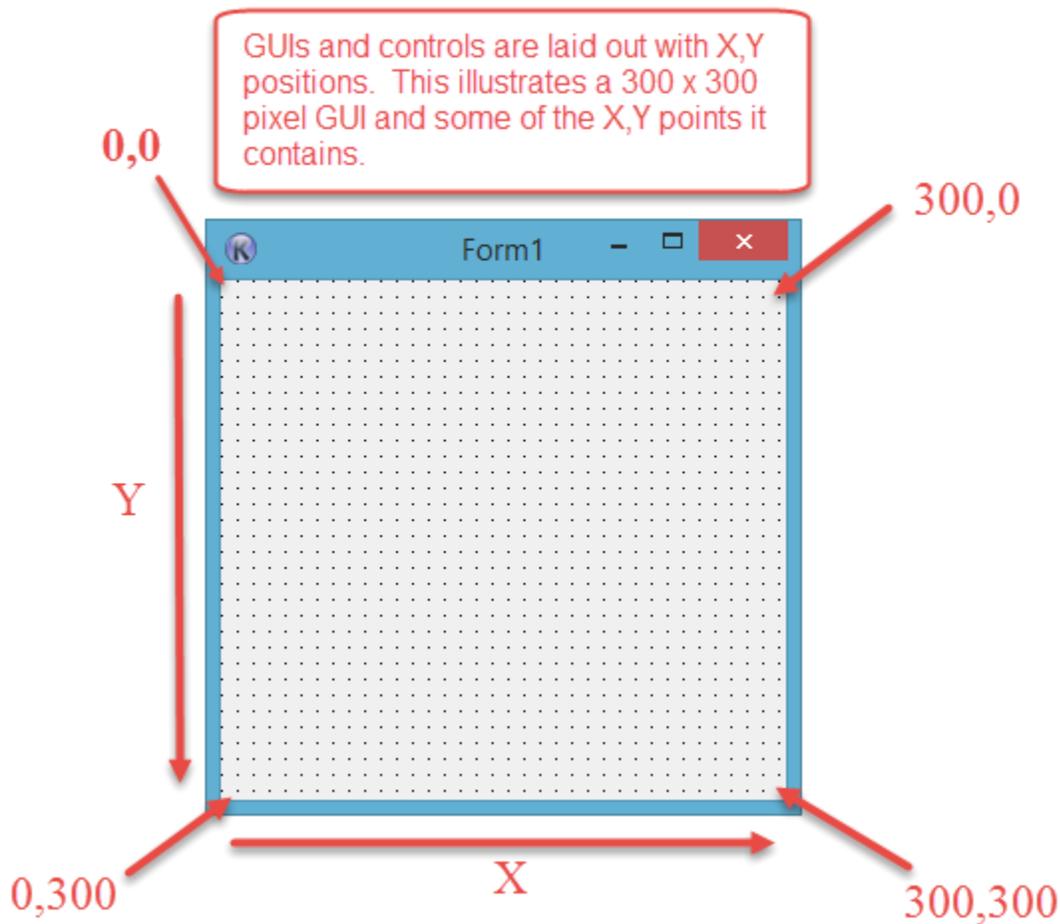
NOTE: you can refer back to this image as we progress through some examples. We will do some basic calculations to determine where to place the controls we create in our examples.

GUI Creation

GUIs can be created with a series of functions starting with the aptly named GUIcreate. Let's take a closer look at that function and its parameters.

GUICreate

Create a GUI window.

```
GUICreate ( "title" [, width [, height [, left = -1 [, top = -1 [,
style = -1 [, exStyle = -1 [, parent = 0]]]]]]] )
```

Parameters

title	The title of the dialog box.
width	**[optional]** The width of the window.
height	**[optional]** The height of the window.
left	**[optional]** The left side of the dialog box. By default (-1), the window is centered. If defined, top must also be defined.
top	**[optional]** The top of the dialog box. Default (-1) is centered
style	**[optional]** defines the style of the window. See GUI Control Styles Appendix. Use -1 for the default style which includes a combination of $WS_MINIMIZEBOX, $WS_CAPTION, $WS_POPUP, $WS_SYSMENU styles. Some styles are always included: $WS_CLIPSIBLINGS, and $WS_SYSMENU if $WS_MAXIMIZEBOX or $WS_SIZEBOX is specified.
exStyle	**[optional]** defines the extended style of the window. See the Extended Style Table below. Use -1 for the default, which is no extended styles. Forced styles: $WS_EX_WINDOWEDGE
parent	**[optional]** The handle of another previously created window - this new window then becomes a child of that window.

The AutoIt help file section on GUIcreate.

We can see that there is only one required parameter: title. The remaining optional parameters are used to dictate the GUIs size, placement, style, and behavior.   If you supply a title the script will create a GUI with some default dimensions but it won't do much. It may not even stick around very long. Why? Because the script is reading the instructions and executing them. When it runs the GUIcreate function the script will end. So what would happen?   We would see it flash up on the screen for a moment and then disappear. How might we remedy this?   Do you remember what I said in Chapter 6 about While loops? Here is a reminder:

"You may have need to create a loop that is open as long as you program is running. A common example is when you show a user interface (i.e. an application with controls like buttons, text boxes, menus, etc) which we will over later."

If we combine the while loop with the creation of a GUI we can get it to stay around as long as we want.

Let's take this in steps. First, let's try to just call the function with the required parameter of title and see what happens. To do that, type this into SciTE and run it: GUIcreate("Test GUI"). Did you see anything? Probably not. That is "When windows are created they are initially hidden so you must use this function to display them (@SW_SHOW)" (from the help file). In order to that we need to add a line of code so our script would now look like this:

GUIcreate("Test GUI")

GUISetState(@SW_SHOW)

If you run those two lines of code you will see the GUI flash up on the screen and disappear. We know the cause of this. Now we need a solution that will stop it using a while loop. Don't do the following, let's just take a look at it to understand what it does and does not do:

```
1    GUIcreate("Test GUI")
2    GUISetState(@SW_SHOW)
3    while 1
4
5    WEnd
```

The above code will create a GUI and show it forever. Why? Because there is nothing in the while loop to ever set the condition being tested (1) to anything other than 1. Isn't that what we wanted? We did want the GUI to stay around for more than a flash. However, forever is a bit longer than we probably wanted. Therefore, we need a way to close the GUI. The following code snippet details an approach we could take to accomplish this:

Code Chapter 9 Example 1

```
1    #include <GUIConstantsEx.au3>
2    #include <WindowsConstants.au3>
3    GUIcreate("Test GUI")
4    GUISetState(@SW_SHOW)
5    While 1
6        $nMsg = GUIGetMsg()
7        Switch $nMsg
             Case $GUI_EVENT_CLOSE
                 Exit
10
11       EndSwitch
12   WEnd
13
```

1

2

3

4

5

6

#1) In this case, we are including AutoIt GUI libraries that help with styles and behaviors. The Windowsconstants.au3 file is referenced in the notes to the GUIcreate function. You can refer to that section of the help file for more information.

#2) This is what we started with. We use GUIcreate and GUISetState to create and show the GUI respectively. The problem was that it flashed up on the screen and disappeared.

#3) This is where we setup our While loop. It still says "While 1" which would theoretically run forever. However, now there is some additional code below that will save us from that fate.

#4) Here again we are using a switch which as we recall from Chapter 5 is a conditional statement. In this case, the switch is triggered by the value of $nMsg – a variable that was included within the from the #include files at the top. We can think of this in terms of the GUI "listening" for a message and performing different behaviors based on the message contents.

#5) This is the only case within our switch. In other words, the only thing that the switch could do is to listen for a message that is $GUI_EVENT_CLOSE. As the name would suggest, that event is triggered when a user attempts to close the GUI.

#6) The case $GUI_EVENT_CLOSE close would not do anything if a user tried to close the GUI without some additional code under it. Therefore, we have added a new keyword called "Exit" that will terminate the script (i.e. end the program).

Putting all these pieces together we have created a GUI that we are showing a user which will stay open until the user closes the GUI (with the red arrow in the upper right hand corner):
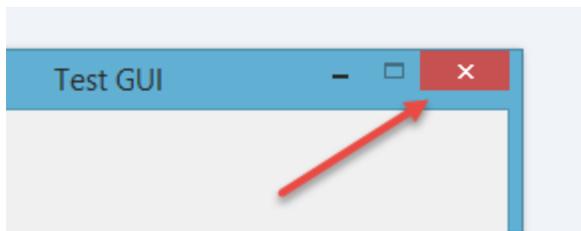
If you are underwhelmed at what we have built and think that it was a lot of work to do nothing – don't worry. It will get easier from here. That was just the basic framework to demonstrate how the GUI works. Also, after we get a few more concepts down we will explore how to design a GUI without writing any code using some free tools that come with AutoIt.

The Button

The following code will add a button called "Push ME" to our GUI:

 Chapter 9 Example 2

```
1   #include <GUIConstantsEx.au3>
2   #include <WindowsConstants.au3>
3   GUIcreate("Test GUI")
4   GUISetState(@SW_SHOW)
5   GUICtrlCreateButton ( "Push ME", 175,200,75,50)
6   While 1
7       $nMsg = GUIGetMsg()
8       Switch $nMsg
9           Case $GUI_EVENT_CLOSE
10              Exit
11
12      EndSwitch
13  WEnd
14
```

The parameters (found in the help file) are title, left, top, width, and height. The title is whatever we want the button to display. The left and top parameters dictate where the button will be positioned within our GUI. If you set the dimensions of a GUI using the optional parameters as you create it you will have a much easier time placing controls. In the present case, I played around with the position because we used the default dimensions (more on this later). When you run the code you should see the following:
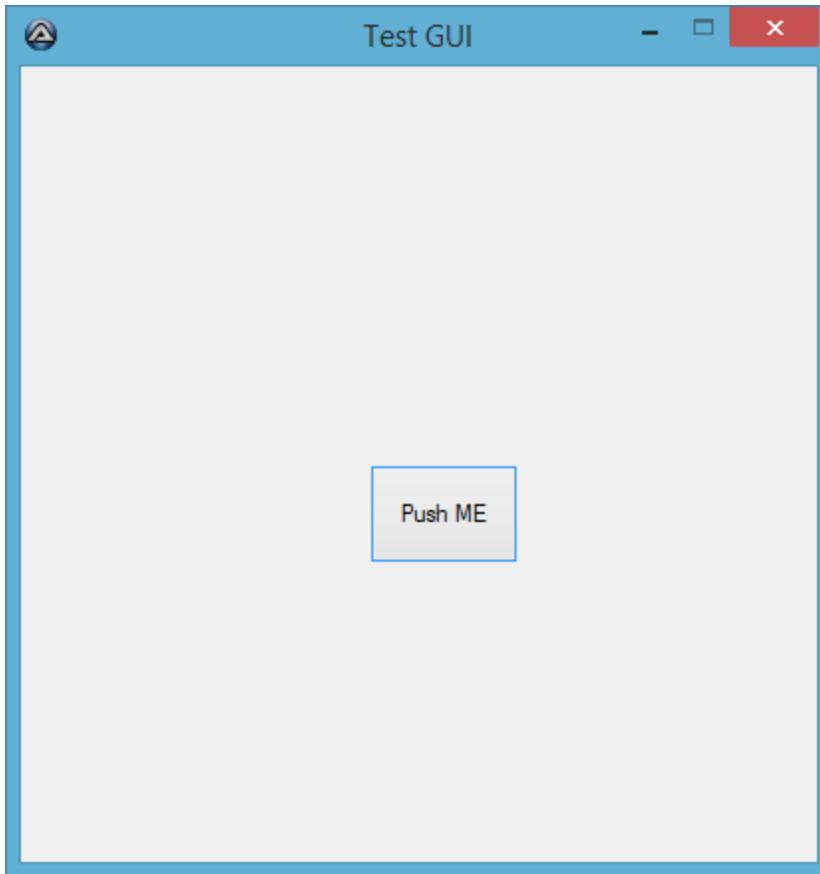
Try pushing the button.   Nothing happens when you do that. That is because we have not told the GUI to listen for the button push. That is a new event that we need to introduce as a case. To that, we could add the following to our script:

Code Chapter 9 Example 3

```
1    #include <MsgBoxConstants.au3>
2    #include <GUIConstantsEx.au3>
3    #include <WindowsConstants.au3>
4    GUIcreate("Test GUI")
5    GUISetState(@SW_SHOW)
6    $button=GUICtrlCreateButton( "Push ME", 175,200,75,50)
7    While 1
8        $nMsg = GUIGetMsg()
9        Switch $nMsg
10           case $button
11               MsgBox($MB_OK,"Button Push","You pushed me!")
              Case $GUI_EVENT_CLOSE
                  Exit
15           EndSwitch
16    WEnd
```

#1) When you successfully create a GUI control AutoIt "returns" a control ID to you. This ID can be used to interact with the control if you store it in a variable. In this case, $button is storing the control ID of the newly created button control.

#2) In this step we are adding $button as a case to our switch within the while loop that is listening for GUI messages. In other words, if our button control (the ID of which is stored in that variable) is pressed then the program will run whatever code we have in the case for the button.

#3) This is the code that we want to run when the button is pushed.

NOTE: You don't have to put all your code in the switch under the case. Remember, you can create large functions elsewhere in your script and simply reference the name(s) of those functions in the switch that is tied to the GUI control. This make it much easier to read your code.

When we tie it all together and push the button, this is what we will see:
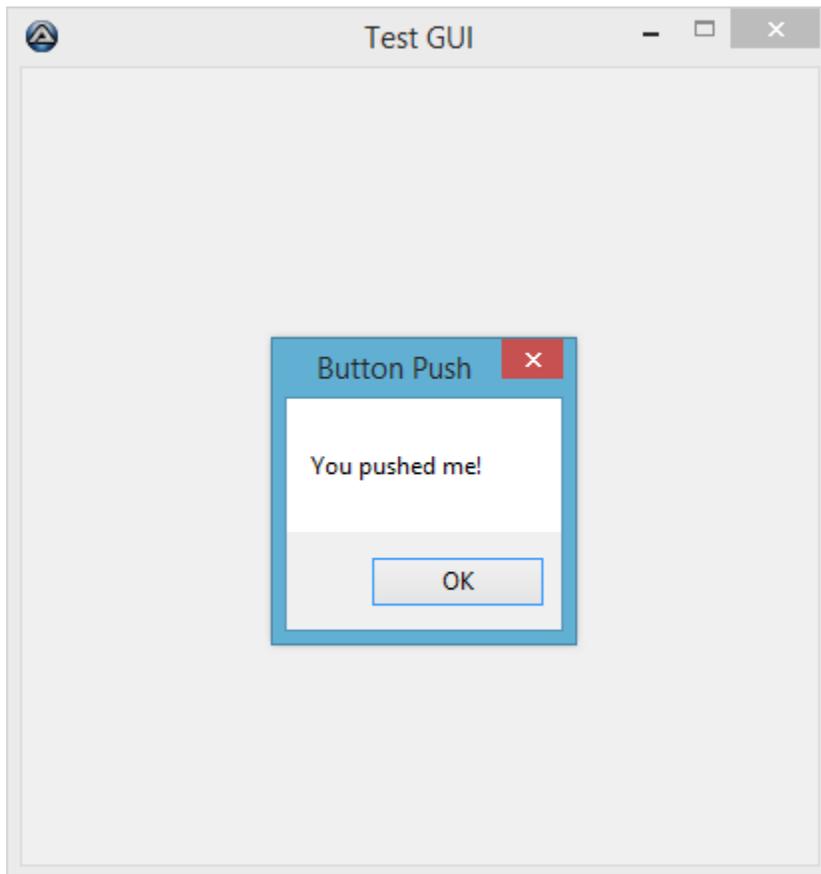
NOTE : To interact with a control in the GUI we can create a case within the switch located in our while loop (used to keep our program running). The case will listen for the controlID returned to us when we successfully created the control. We store that ID in a variable and then "listen" for interaction with it. When the case is triggered the script will run the code contained in the case.

The Input Box

It was great making a button and having it do something when we pushed it. However, we often need to collect information from a user in order to make our program more useful. One way to do that is with an input field. The function for this is described below in the AutoIt help file:

GUICtrlCreateInput

Creates an Input control for the GUI.

```
GUICtrlCreateInput ( "text", left, top [, width [, height [, style =
-1 [, exStyle = -1]]]] )
```

Parameters

text	The text of the control.
left	The left side of the control. If -1 is used then left will be computed according to GUICoordMode.
top	The top of the control. If -1 is used then top will be computed according to GUICoordMode.
width	[optional] The width of the control (default is the previously used width).
height	[optional] The height of the control (default is the previously used height).
style	[optional] Defines the style of the control. See GUI Control Styles Appendix. default (-1) : $ES_LEFT, $ES_AUTOHSCROLL forced styles : $WS_TABSTOP only if no $ES_READONLY. $ES_MULTILINE is always reset.
exStyle	[optional] Defines the extended style of the control. See Extended Style Table. default (-1) : $WS_EX_CLIENTEDGE

As you can see, the parameters to create an input are very similar to those used to create GUI. Let's incorporate an input field into our program. As we do this, we will also change the GUI a tiny bit by adding our own width and height parameters rather than using the default size. This will make it easier to place our controls. The following code creates variables that are used in the script to set the width and height of the GUI.   We can the reference the variables as the width and height parameters instead of using numbers.

$guiWidth=300 ; the width of our GUI

$guiHeight=300 ; the height of our GUI

Next, we will create similar width and height variables for our button and the input field:

$guiBtnWidth=75 ; the width of our button

$guiBtnHeight=50 ; the heighr of our button

$inputWidth=150 ;the width of our input field

$inputHeight=30; the height of our input field

Positioning GUI Controls

REMINDER: the layout chart we saw at the beginning of the chapter that explained X,Y coordinates? Now may be a good time to refer back to it as we are about to do some basic mathematical operations to calculate the placement of our button and input field's position (i.e. their "left" and "top" parameters).

In order to center the button and input in the middle of the GUI we need to use a small bit of math. We need to take the width of the GUI and divide by 2. This is the middle of the GUI. However, the button and input also have their own widths. Therefore, we need to subtract the width of the button and input after they are divided by 2 to center them (this takes into account their width as part of centering process). Then we need to do the same thing for their height.

; math to center the controls

$buttonTop=($guiHeight/2)-($guiBtnHeight/2)

$buttonLeft=($guiWidth/2)-($guiBtnWidth/2)

$inputLeft=($guiWidth/2)-($inputWidth/2)

$inputTop=(($guiHeight/2)-($guiBtnHeight/2))-($inputHeight+10)

Now let's walk through some of the math in more detail as we take a closer look at how we placed the input control. First, we positioned its left most coordinate. This was easy. We did the same thing as our button code above except now we are using the input width:

($guiWidth/2)-($inputWidth/2)

Next we had to position it's height relative to the top of the GUI. This was a bit more complicated because if we used the same exact calculation as the button the input field would be placed directly on top of the button. That would have obscured the button so instead we had to take what we did for the button, factor in the height of the input field, and then add a 10 pixel buffer so that the input field appeared above the button. Had we not done this the two controls would have been laid out on top of one another and we could not interact with the properly:

(($guiHeight/2)-($guiBtnHeight/2))-($inputHeight+10)

Note the use of the parenthesis that dictate the order in which the operators are evaluated. The first part the equation is the same our calculation to find the top parameter for the button. Then we subtracted the height of the input control plus the aforementioned 10 additional pixels. This resulted in placing the input control above the button by 10 pixels.

When we run our GUI it will look like this:

The code appears in full below:

Code Chapter 9 Example 4

```autoit
1    #include <MsgBoxConstants.au3>
2    #include <GUIConstantsEx.au3>
3    #include <WindowsConstants.au3>
4
5    $guiWidth=300 ; the width of our GUI
6    $guiHeight=300 ; the height of our GUI
7
8    $guiBtnWidth=75 ; the width of our button
9    $guiBtnHeight=50 ; the heighr of our button
10
11   $inputWidth=150
12   $inputHeight=30
13
14   $buttonTop=($guiHeight/2)-($guiBtnHeight/2)
15   $buttonLeft=($guiWidth/2)-($guiBtnWidth/2)
16   $inputLeft=($guiWidth/2)-($inputWidth/2)
17   $inputTop=(($guiHeight/2)-($guiBtnHeight/2))-($inputHeight+10)
18
19
20   GUIcreate("Test GUI",$guiWidth,$guiHeight) ; using our variables to set height & width
21   GUISetState(@SW_SHOW)
22   ;on the following line we are using some math and some variables to center the button by taking
23   ; width and height of the GUI and button
24   $button = GUICtrlCreateButton ( "Push ME",$buttonLeft,$buttonTop,$guiBtnWidth,$guiBtnHeight)
25   $input=GUICtrlCreateInput("Enter text here",$inputLeft,$inputTop,$inputWidth,$inputHeight)
26
27   While 1
28       $nMsg = GUIGetMsg()
29       Switch $nMsg
30           case $button
31               MsgBox($MB_OK,"Button Push","You pushed me!")
32           Case $GUI_EVENT_CLOSE
33               Exit
34
35       EndSwitch
36   WEnd
```

Why do this instead of just entering the positions we want as numeric values? One reason would be that it allows us to place the button dead center in the GUI by using code. Another is that we may change our minds a bit as we add controls. What if we want to make the GUI bigger and keep the button in the center? What would we have to do? If we did not use variables we would have to manually calculate the position relative to the GUI and then update the left and top numeric parameters accordingly. Using the above approach it is automatic. When we make changes to the $guiWidth and $guiHeight variables. The code does the work for us.

NOTE : using variables to position GUI controls can save you time if you decide to make changes to your GUI. The controls can be setup to reposition themselves so that any change you make impacts all other GUI elements without any additional coding.

Listening for control messages

Now we are getting somewhere. We have a GUI, button, and input box. We can also make changes to the layout with relative ease and everything will reposition itself. Of course, when we push the button all that will happen is that we will see the message box that we wired to the $button earlier. Let's use that same message box to present whatever text we have typed into the input field. To do that, we will need to read the input control and store the information in a variable. Fortunately, there is a function for this called GUICtrlRead.

GUICtrlRead

It has one mandatory parameter of controlID. Remember that when we created the button we stored the controlID in the button variable? We are doing the same thing for the input control when we placed the variable $input= in front of its creation. Therefore, to read the what is in the input control all we need to do is GUICtrlRead($input). However, we will want to store that message in a variable for later use so let's do this: $inputContents= GUICtrlRead($input). Then we only need to change the text parameter in the message box to reference our $inputContents variable instead of our hard coded text. This is what it would look like. Try it out yourself.

```
While 1
    $nMsg = GUIGetMsg()
    Switch $nMsg
        case $button
            $inputContents=GUICtrlRead($input)
            MsgBox($MB_OK,"Button Push",$inputContents)
        Case $GUI_EVENT_CLOSE
            Exit

    EndSwitch
WEnd
```

> Read the input control text here

> Have the MsgBox display the text from the input control
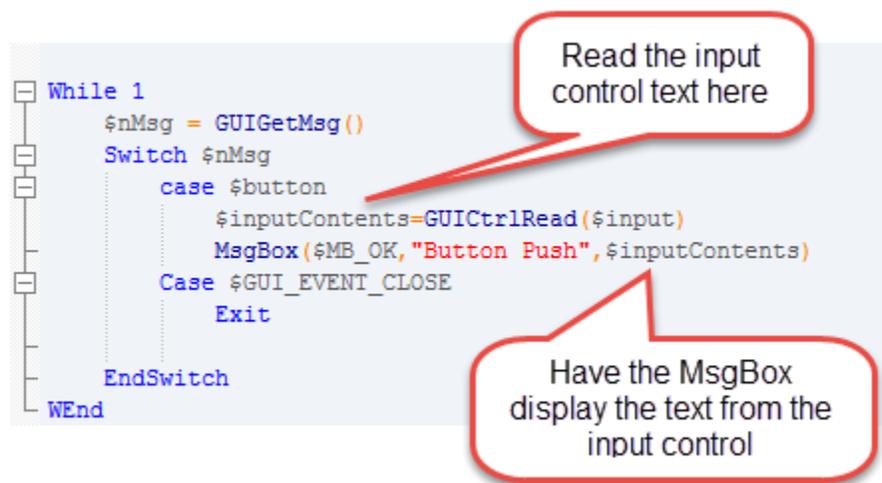
Alternatively, we could have done it all in one line of code as follows:

```
26
27  While 1
28      $nMsg = GUIGetMsg()
29      Switch $nMsg
30          case $button
31              MsgBox($MB_OK,"Button Push",GUICtrlRead($input))
32          Case $GUI_EVENT_CLOSE
33              Exit
34
35      EndSwitch
36  WEnd
```

> The text displayed in the MsgBox is being read from the input control

Simple GUI Calculator

A great extension of this example would be to use the custom math function we created in Chapter 7. Remember it? It allowed us to take any two numbers and perform a mathematical operation on them. At the time, we were limited to manually entering the values for the math operations every time we wanted to do a calculation (i.e. _mathWithAnswers(5,3,"+"). Now we can add a few controls to our GUI and we will have a dynamic calculator. We already have one input control. That could be for the first number. However, we will need to reposition it slightly by moving it up to allow enough room for another input field. To do this we could change our top position calculation as follows:

```
$inputTop= (($guiHeight/2)-($guiBtnHeight/2))-(($inputHeight*2)+10)
```

Note: we are using 2 x the height of the control giving us some additional room for the new input control

Now we need to add a second input control for the second number. We already have all the math for how the first two controls layout. No we can simply add a new line creating $input2. The only parameter we need to change is "top". To do this so that the new input appears 10 pixels above the old one we could define $input2Top as:

```
$input2Top=$inputTop+($inputHeight+10)
```

Here we are adding to the top position of the first input so this control will appear beneath it with a small 10 pixel cushion

The complete line of code would look like this:

```
$input2=GUICtrlCreateInput("Enter text here",$inputLeft,$input2Top,$inputWidth,$inputHeight)
```

Note we used identical parameters to $input with the exception of top. Now the GUI looks like this:

One problem is that we will always have to overwrite the existing text in our input boxes. To fix the problem we could set the title parameter to "" (double quotes with no space). This is an empty string so the input control will be blank at the start.

That is all well and good but how about our mathematical operator? What could we use for that? If we use an input box we run the risk of a user typing in things that aren't operators. We know that the operator can only be one of four choices: +,-,/, and *. Therefore, it may be best to give those and only those choices to a user. One way to accomplish that could be with the use of a combo box control. A combo box is a dropdown menu that will contain whatever values we feed it as choices for the user. The help file shows the function required to create a combo box:

GUICtrlCreateCombo

GUICtrlCreateCombo

Creates a ComboBox control for the GUI.

```
GUICtrlCreateCombo ( "text", left, top [, width [, height [, style =
-1 [, exStyle = -1]]]] )
```

Parameters

text	The text which will appear in the combo control.
left	The left side of the control. If -1 is used then left will be computed according to GUICoordMode.
top	The top of the control. If -1 is used then top will be computed according to GUICoordMode.
width	[optional] The width of the control (default is the previously used width).
height	[optional] The height of the control (default is the previously used height).

style	[optional] Defines the style of the control. See GUI Control Styles Appendix.
	    default (-1) : $CBS_DROPDOWN, $CBS_AUTOHSCROLL, $WS_VSCROLL
	    forced style : $WS_TABSTOP

exStyle	[optional] Defines the extended style of the control. See Extended Style Table. default (-1) : $WS_EX_CLIENTEDGE

Our biggest problem will be where to place it. I would propose to the left of the two input boxes just between them on the height so it looks like a math equation. To do this we would create the following variables:

```
$comboWidth=50
$comboHeigh=30
$comboLeft=$inputLeft-($comboWidth+10)
$comboTop=(($inputTop+$input2Top)/2)
```

The first two variables set the width and height of the combo control. The second two variables $comboLeft and$comboTop are storing the positions we calculated to place the combo control in our GUI. The $comboLeft calculation starts with the left position of the input controls. Then it subtracts the width of the combo control + another 10 pixels. This means it will appear 10 pixels to the left of the input controls. Next we calculate the topmost position of the control. We want it to appear between the two input controls so we add the top positions of those controls together and divide by two to find the middle.

The line of code we would add to our GUI section would look like this:

```
$combo=GUICtrlCreateCombo("Oper",$comboLeft,$comboTop,$comboWidth,$comboHeigh)
```

With that control added out GUI now looks like this:
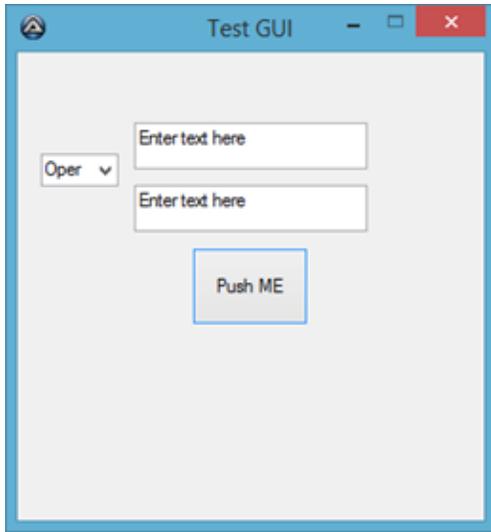
The complete code is getting a bit long but it is as follows:

Code Chapter 9 Example 5

```autoit
#include <MsgBoxConstants.au3>
#include <GUIConstantsEx.au3>
#include <WindowsConstants.au3>

$guiWidth=300 ; the width of our GUI
$guiHeight=300 ; the height of our GUI

$guiBtnWidth=75 ; the width of our button
$guiBtnHeight=50 ; the height of our button

$inputWidth=150
$inputHeight=30

$buttonTop=($guiHeight/2)-($guiBtnHeight/2)
$buttonLeft=($guiWidth/2)-($guiBtnWidth/2)
$inputLeft=($guiWidth/2)-($inputWidth/2)
$inputTop= (($guiHeight/2)-($guiBtnHeight/2))-(($inputHeight*2)+10)
$input2Top=$inputTop+($inputHeight+10)


$comboWidth=50
$comboHeigh=30
$comboLeft=$inputLeft-($comboWidth+10)
$comboTop=(($inputTop+$input2Top)/2)

GUIcreate("Test GUI",$guiWidth,$guiHeight) ; using our variables to set height & width
GUISetState(@SW_SHOW)
; on the following line we are using some math and some variavles to center the button by taking
; width and height of the GUI and the button
$button = GUICtrlCreateButton ( "Push ME",$buttonLeft,$buttonTop,$guiBtnWidth,$guiBtnHeight)
$input=GUICtrlCreateInput("Enter text here",$inputLeft,$inputTop,$inputWidth,$inputHeight)
$input2=GUICtrlCreateInput("Enter text here",$inputLeft,$input2Top,$inputWidth,$inputHeight)
$combo=GUICtrlCreateCombo("Oper",$comboLeft,$comboTop,$comboWidth,$comboHeigh)
While 1
    $nMsg = GUIGetMsg()
    Switch $nMsg
        case $button
            $inputcontents=GUICtrlRead($input)
            MsgBox($MB_OK,"","Text from input ",$inputcontents)
        Case $GUI_EVENT_CLOSE
            Exit

    EndSwitch
WEnd
```

The problem is that our combo dropdown has no values. If we add the following line to beneath the line where we created the combo control all of our mathematical operators will be available in the dropdown:

```autoit
GUICtrlSetData(-1, "+|-|/|*", "+")
```

The -1 is simply a reference to the preceding control. Because this code was placed under the combo control it is referring to that control. What follows is a pipe delimited string of values for the combo

control (the pipe looks like a straight vertical line and is usually above the enter key on your keyboard). The final value after the initial string and last comma is the optional default. In this case we are defaulting to "+".

Now our GUI has those options available:
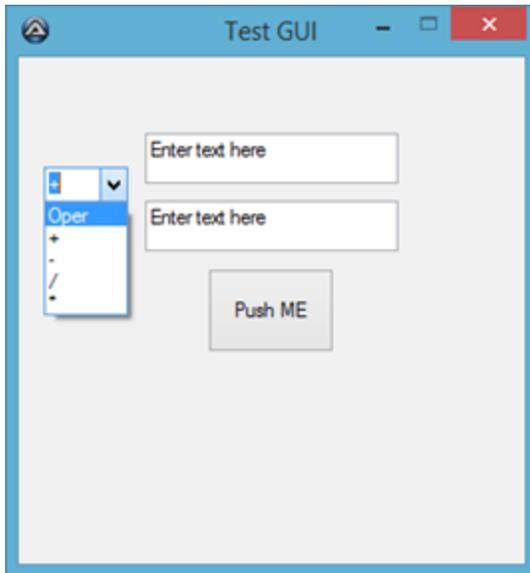
To use our calculator function from Chapter 7 we need to add it to this script. We could use the keyword #include for that. Refer back to Chapter 7 for that code. When we include it – we need to make sure that we remove the last line that triggered the function – all we want is the function. If we were to include that line it would calculate 5+3 and show us the answer before we could use our calculator. By eliminating that last line (19), we just get the math function ready to go for whatever we call upon it to do.

```
File  Edit  Search  View  Tools  Options  Language  Buffers  Help

  1        ; An example of a user function
  2        #include <MsgBoxConstants.au3>
  3
  4     ⊟  Func _mathWithAnswers($firstNum,$secondNum,$operator)
  5
  6     ⊟    Switch $operator
  7     ⊟        case $operator = "+"
  8               $answer=$firstNum+$secondNum
  9     ⊟        case $operator = "-"
 10               $answer=$firstNum-$secondNum
 11     ⊟        case $operator = "*"
 12               $answer=$firstNum*$secondNum
 13            case $operator = "/"
 14        EndSwitch
 15        MsgBox($MB_OK,"Answer",$answer)
 16        EndFunc
 17
 18        _mathWithAnswers(5,3,"+")
```

> Remove this line and save as "calc.au3" in the same directory as your

You should save that script from Chapter 7 with the math function in the same directory as your GUI and name it "calc.au3". To include it in our GUI script we would reference it at the top with the other includes:

```
#include <GUIConstantsEx.au3>
#include <WindowsConstants.au3>
#include <calc.au3>
```

Now the math function we wrote in Chapter 7 is now available to us. Remember that we are passing that function three arguments: first number, second number, and mathematical operator. We can get the first and second number from our two input controls and the operator from our dropdown. This would look like:

$firstNum=GUICtrlRead($input)

$secondNum= GUICtrlRead($input2)

$operator= GUICtrlRead($combo)

To make this work with our GUI we might alter what happens when we press the button once again as follows:

```
case $button
    $firstNum=GUICtrlRead($input)
    $secondNum=GUICtrlRead($input2)
    $operator=GUICtrlRead($combo)
    _mathWithAnswers($firstNum,$secondNum,$operator)
```

The revised script appears as follows:

Code   Chapter 9 Example 6

```
1    #include <GUIConstantsEx.au3>
2    #include <WindowsConstants.au3>
3    #include <calc.au3>
4
5    $guiWidth=300 ; the width of our GUI
6    $guiHeight=300 ; the height of our GUI
7
8    $guiBtnWidth=75 ; the width of our button
9    $guiBtnHeight=50 ; the heighr of our button
10
11   $inputWidth=150
12   $inputHeight=30
13
14   $buttonTop=($guiHeight/2)-($guiBtnHeight/2)
15   $buttonLeft=($guiWidth/2)-($guiBtnWidth/2)
16   $inputLeft=($guiWidth/2)-($inputWidth/2)
17   $inputTop= (($guiHeight/2)-($guiBtnHeight/2))-(($inputHeight*2)+10)
18   $input2Top=$inputTop+($inputHeight+10)
19
20
21   $comboWidth=50
22   $comboHeigh=30
23   $comboLeft=$inputLeft-($comboWidth+10)
24   $comboTop=(($inputTop+$input2Top)/2)
25
26   GUIcreate("Test GUI",$guiWidth,$guiHeight) ; using our variables to set height & width
27   GUISetState(@SW_SHOW)
28   ; on the following line we are using some math and some variavles to center the button by taking into account the
29   ; width and height of the GUI and the button
30   $button = GUICtrlCreateButton ( "Push ME",$buttonLeft,$buttonTop,$guiBtnWidth,$guiBtnHeight)
31   $input=GUICtrlCreateInput("",$inputLeft,$inputTop,$inputWidth,$inputHeight)
32   $input2=GUICtrlCreateInput("",$inputLeft,$input2Top,$inputWidth,$inputHeight)
33   $combo=GUICtrlCreateCombo("Oper",$comboLeft,$comboTop,$comboWidth,$comboHeigh)
34   GUICtrlSetData(-1, "+|-|/|*", "+")
35
36   ;GUICtrlCreateInput(
37   While 1
38       $nMsg = GUIGetMsg()
39       Switch $nMsg
40           case $button
41               $firstNum=GUICtrlRead($input)
42               $secondNum=GUICtrlRead($input2)
43               $operator=GUICtrlRead($combo)
44               _mathWithAnswers($firstNum,$secondNum,$operator)
45           Case $GUI_EVENT_CLOSE
46               Exit
47
48       EndSwitch
49   WEnd
```

The function in our calc.au3 file from Chapter 7 already includes a message box that will display the answer. Therefore, we can eliminate the message box we were previously using in this area of the code. When we run the script we see that we have built a crude calculator:
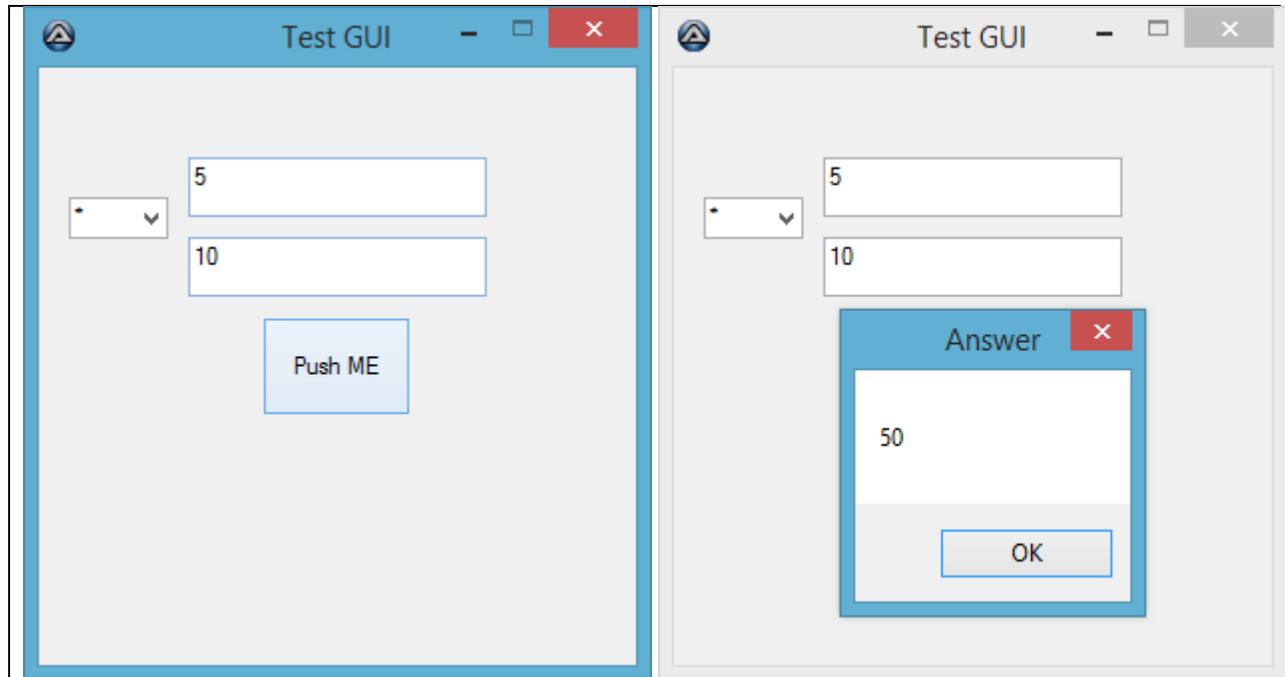
The complete code for this example is as follows:

There are many more GUI controls available. They are all detailed in the help file under GUI Control Creation. By following the example in this chapter you should be able to add them to your scripts with relative ease.

 REMINDER: Graphical User Interfaces (GUIs sometimes pronounced gooey) allow users to interact with your programs. AutoIt supports a wide variety of controls that can be placed on a GUI such as inputs, combo dropdowns, buttons, and more. The parameters used when creating a control dictate the size and position of the control relative to the GUI. When coding GUIs from hand a good practice is to use variables for your measurements so that you can make significant changes to position of the controls without recoding all their parameters.

Chapter 10: Introducing KODA – Drag and Drop Graphical User Interface Tool

In Chapter 9 we explored the creation of GUIs using AutoIt functions. We tackled some of the challenges with layouts (i.e. the relative position of controls we were using to the GUI and each other). We used math functions to be able to have any desired changes ripple through the GUI and automatically adjust everything. That foundation will prove useful to AutoIt script creation and for other programming endeavors. In many programming languages it is the only way to create a user interface. Fortunately, in AutoIt there is another way – Koda. Koda is a "forms creation tool" (forms are sometimes how we refer to user interfaces / GUIs) that allows users to quickly create an interface by

dragging and dropping desired elements wherever they want them to be when the script runs. Koda generates the code for this that you can incorporate into your script. It should install into your SciTE directory as part of your AutoIt install. However, if you do not find it there you download Koda from http://koda.darkhost.ru/page.php?id=download. Download and install the tool:

Step 1:
Click on the zip file containing the latest release from the site:
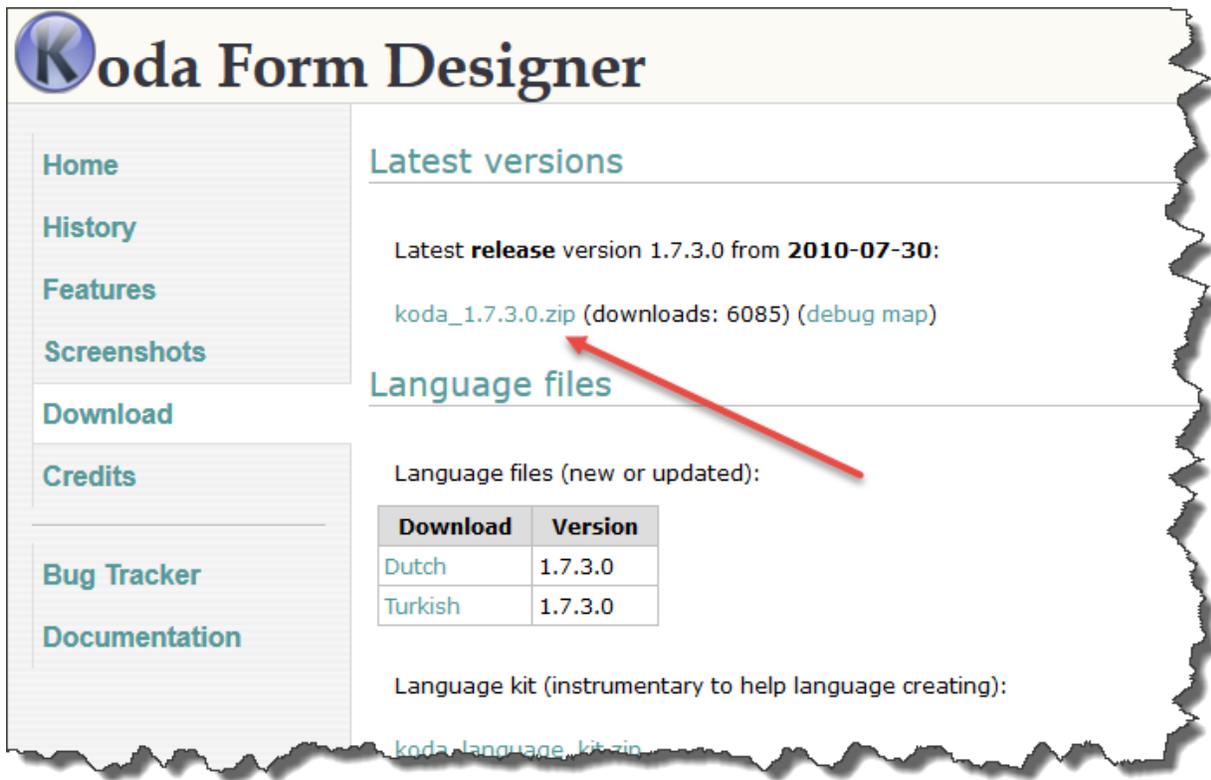
Step 2:
Unzip the file in the directory you wish to contain Koda (preferably the same one that contains SciTE).
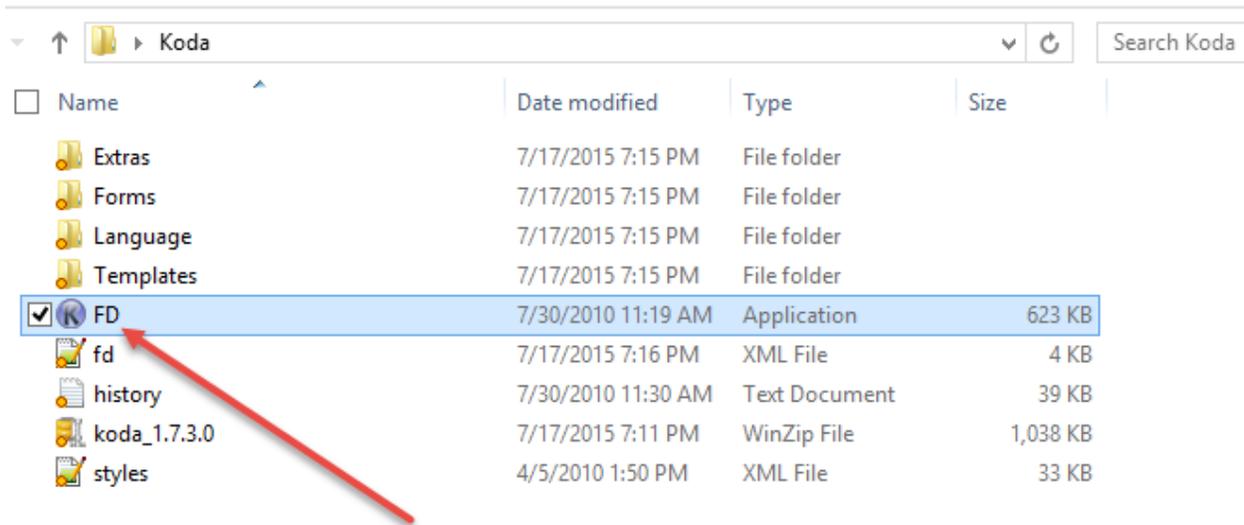
Step 3:

Double click the FD.exe application to launch Koda. You should see the following:
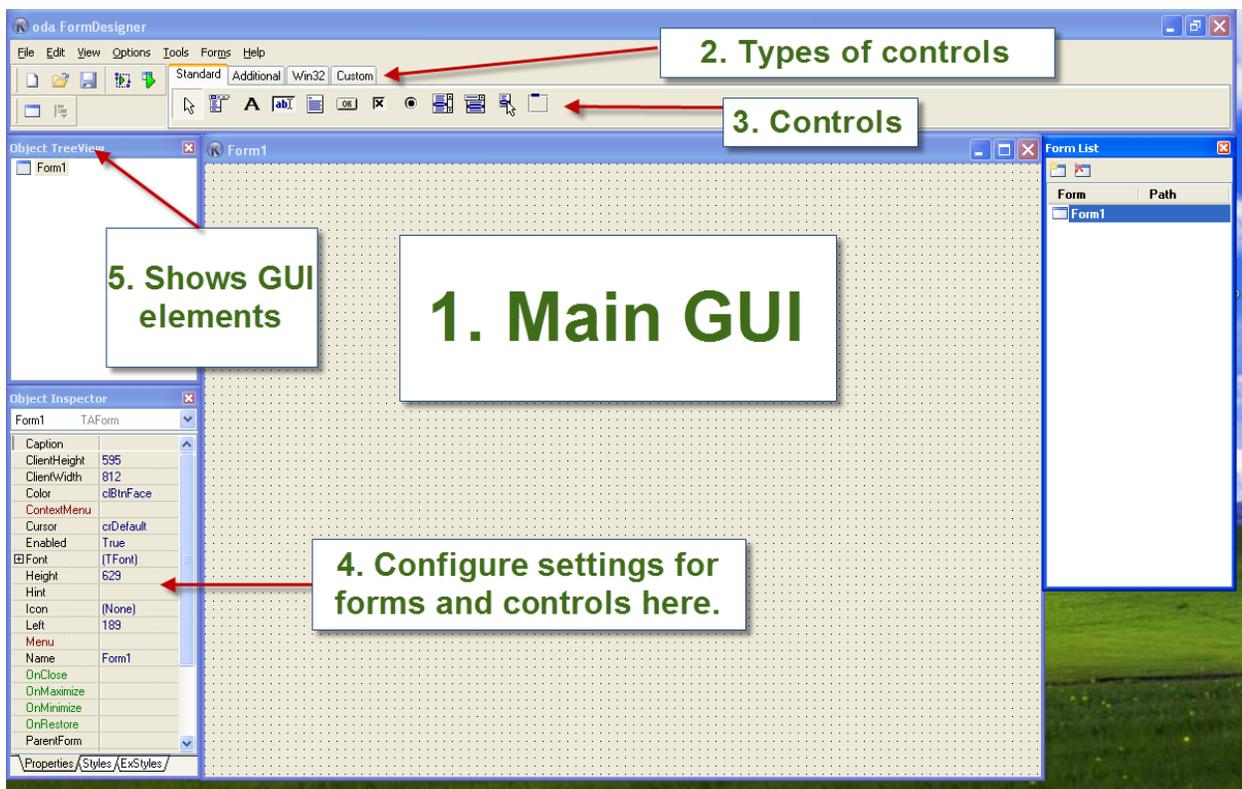
1. The main screen has several different menus and panels. We will focus on the five main areas where we will do most of the work.
2. The main GUI area. This is the GUI to which you can add controls. It is resizable. To resize the GUI simply use your mouse on any edge and hold down the left mouse key while dragging.

Contrast this approach with what you would have to do when using the GUICreate function we covered earlier (i.e. supply the exact dimensions and then use code to update them).

3. At the top of the screen are several tabs that group different types of GUI controls. You can click through the different tabs to explore the groupings.

4. Beneath the tabs that group the controls are the controls themselves. You can hover over them with your mouse to see their names. You should observe that they are the same types of controls available from the standard functions within AutoIt. Adding a control is as simple as clicking on it and "drawing" it on the GUI by holding down your left mouse button and moving the mouse to get the desired size and shape. You can always resize after the initial control creation in a similar fashion to resizing the GUI.

5. The fourth area is the object inspector. It is where you provide additional details that impact the appearance and behavior of the GUI and controls. For example, you may have observed that when Koda opened the default name of the GUI that appeared at the top in the blue bar was "Form1". What if you wanted it to say something more descriptive like "My First Application v1.0"? Koda makes this task simple. You would simply set the caption property of the Object Inspector to your desired text and watch it appear on your form as you enter it.
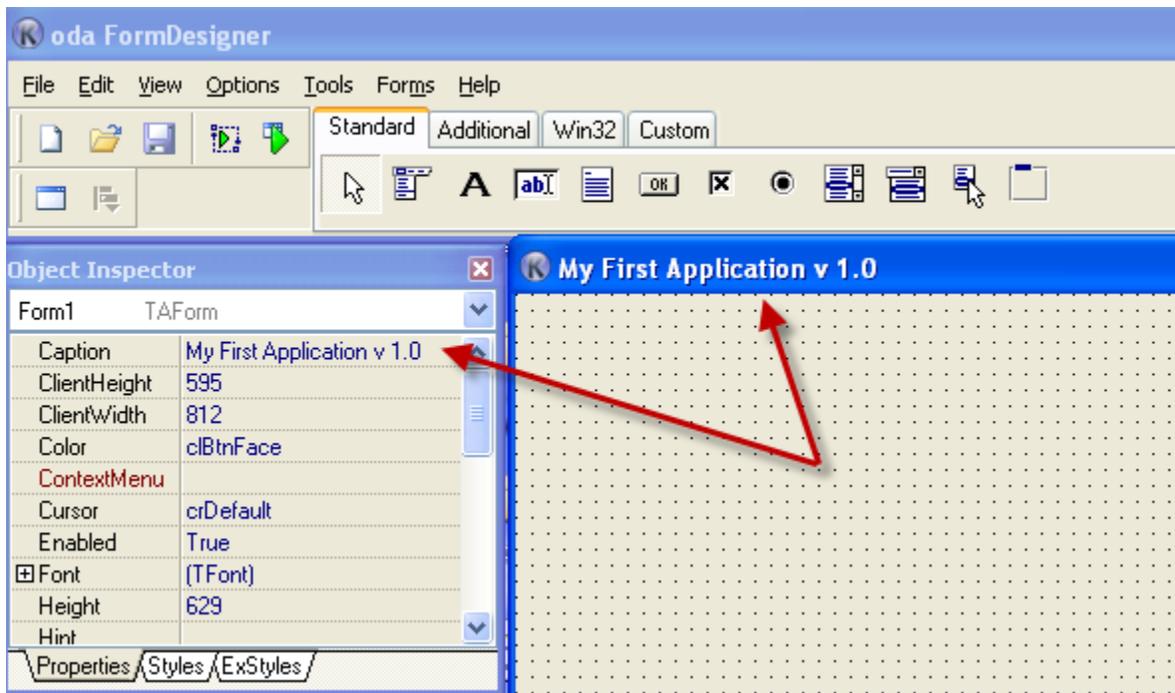
Now that we understand the basic areas of Koda let's recreate our calculator GUI from Chapter 9 and compare the effort.

Step 1: Fire up Koda and resize the default form to 300 width x 300 height. This can also be done with the object inspector:
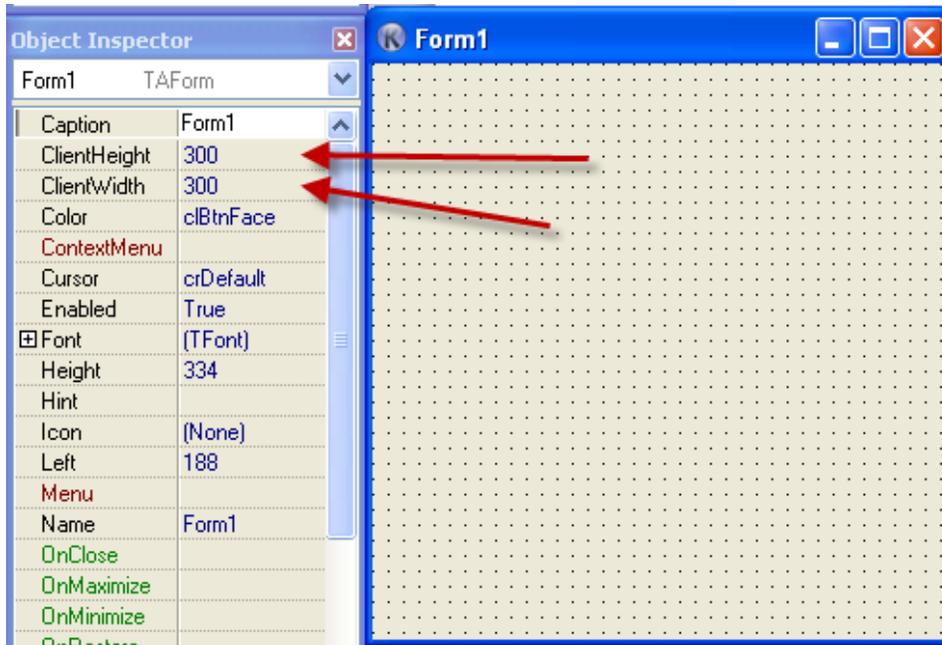
Step 2:

Let's create our first control – the button that was centered vertically and horizontally in our calculator. To do this we make sure the standard control tab is activated and select the input control from that group. Then we need to select the button icon.

Then we drag our mouse with the left hand button pressed to draw it onto the GUI. Don't worry about its dimensions or placement. We can correct all that with the object inspector. We want the button to be centered vertically and horizontally just like the one we created in Chapter 9. Do you remember how we did that? We manually calculated the left and top positions relative to the height and width of the GUI. To do that we had to also factor in the height and width of the button itself:

```
$buttonTop=($guiHeight/2)-($guiBtnHeight/2)
$buttonLeft=($guiWidth/2)-($guiBtnWidth/2)
```

Then we had to find the horizontal and vertical centers of the GUI position the button in the middle by dividing the height and width of the button in half and subtracting it from the GUI center points.

```
$buttonTop=($guiHeight/2)-($guiBtnHeight/2)
$buttonLeft=($guiWidth/2)-($guiBtnWidth/2)
```

Whew, that was a lot of work! In Koda it is much easier. Before we center It we need to set the button's width, height, name, and "caption" (i.e. the text it will display):
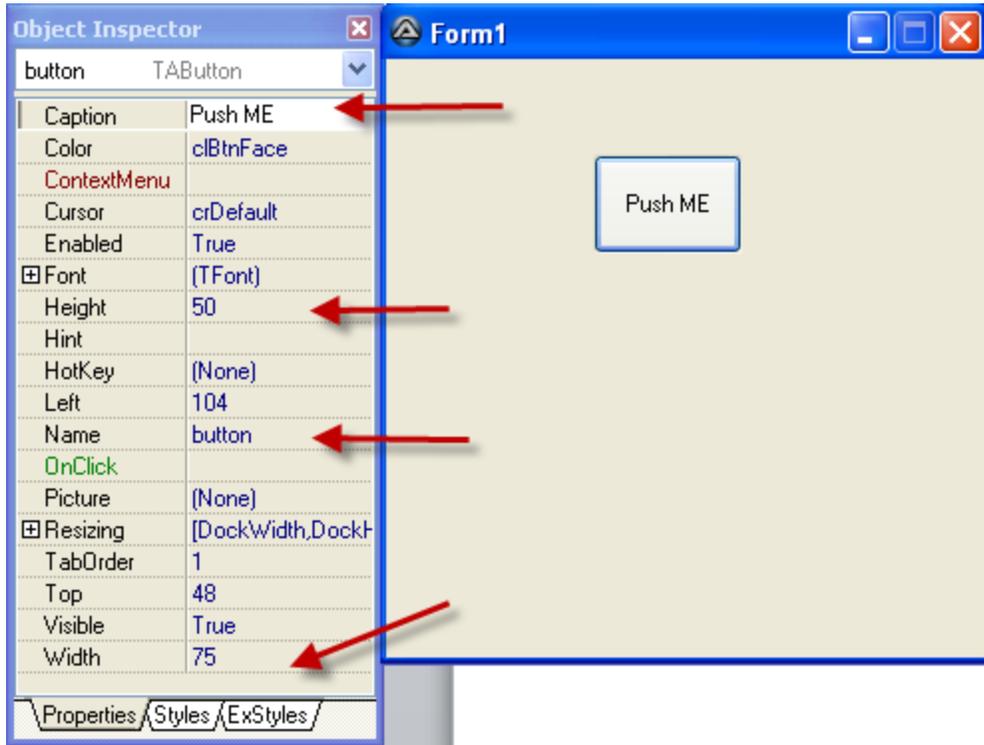
Now that we have the correct properties for our button we need to center it. To do that simply right click on the button control we will see an option called "align". The submenu choices from align are horizontal and vertical.   We will do both:
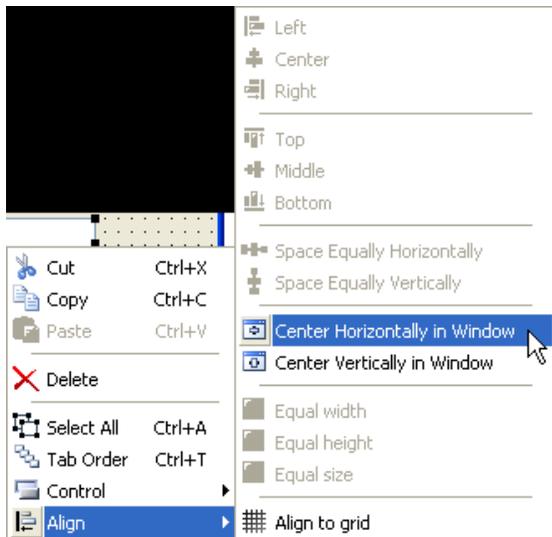
The result is perfect placement of our button control with just a few mouse clicks:

Jayme Fishman, Copyright 2015 all rights reserved

Step 3:

With our "anchor" control in place (i.e. the control that will dictate the position of our other controls because we coded them relative to the button's position in Chapter 9 and for no other reason – we don't have to do it this way) we can start to add the input and combo controls.   Once again we will navigate to the standard tab grouping in Koda. However, this time we will select the input control:

Once we select the control we will create it on the GUI by using a dragging motion on the GUI with left mouse button depressed. Again, the exact size and shape does not matter because we will handle that through the object explorer. Go to the object explorer and set the input box to the same width and height it was in our calculator example. Then make sure the name of the control is the same (i.e. input … you don't need the "$" in the object explorer – Koda will add that for you automatically. Next, make sure the input box is blank by deleting the text in the "text" section. Finally, use the approach to center the control by right clicking on it and selecting the horizontal center.

A few clicks voila!

The next step is to position the first input control vertically to match our example in Chapter 9. You may recall that we previously set the vertical position of this control with code:

```
$inputTop= (($guiHeight/2)-($guiBtnHeight/2))-(($inputHeight*2)+10)
```

The code set the top most position of the input control to that of the button less 2 x 30 (the height of the input control) + 10 for a total of 70. If we click on the button control in Koda we can see that the top value is set to 125. 125 – 70 = 55. Therefore, the topmost position of our first input control should be 55 which we will set in the object explorer.

Step 4:

Now let's add the second input control. We need to follow the exact same steps we performed in Step 3 with two exceptions: (1) the name of the this input control will be "input2" and the topmost position will be 95. How did we get 95? In Chapter 9 we used the following code:

```
$input2Top=$inputTop+($inputHeight+10)
```

That code sets the top of the second input control the top of the first input control + 40 (30 for the height and another 10 for separation between the two).

Step 5:

We are almost done. The final step before we wire this GUI to our program is to create our combo box. Once again we go to the general tab grouping for controls but this time we click on the combobox icon:
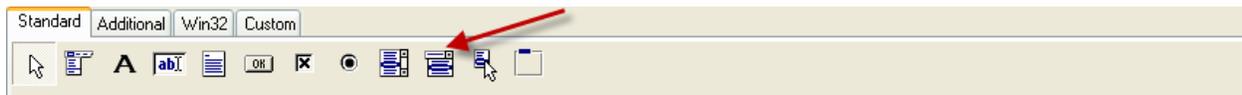
Next we bring our mouse pointer over to the GUI and create the control by dragging its approximate shape and position. Then we make sure the combo box control is selected and we use the object explorer to sets its shape, size, and position. In this case, we will also add the values we want to appear in the drop down when a user interacts with the control. A quick review of the code we used in Chapter 9 gives us all the information we need to configure this last control:

```
$comboWidth=50
$comboHeigh=30
$comboLeft=$inputLeft-($comboWidth+10)
$comboTop=(($inputTop+$input2Top)/2)
```

The left position of the combo control is the left position of the input control less 60 (the combo control width + 10). If we click on one of the input controls we can see the left position is 77 in the Koda object explorer. Therefore, the left position of the combo control will be 77-60=17. The top position of the control is sum of the top positions for our two input controls divided by two. We know from above those two values were 55 and 95 respectively. Those add up to 150. Half of 150 is 75.
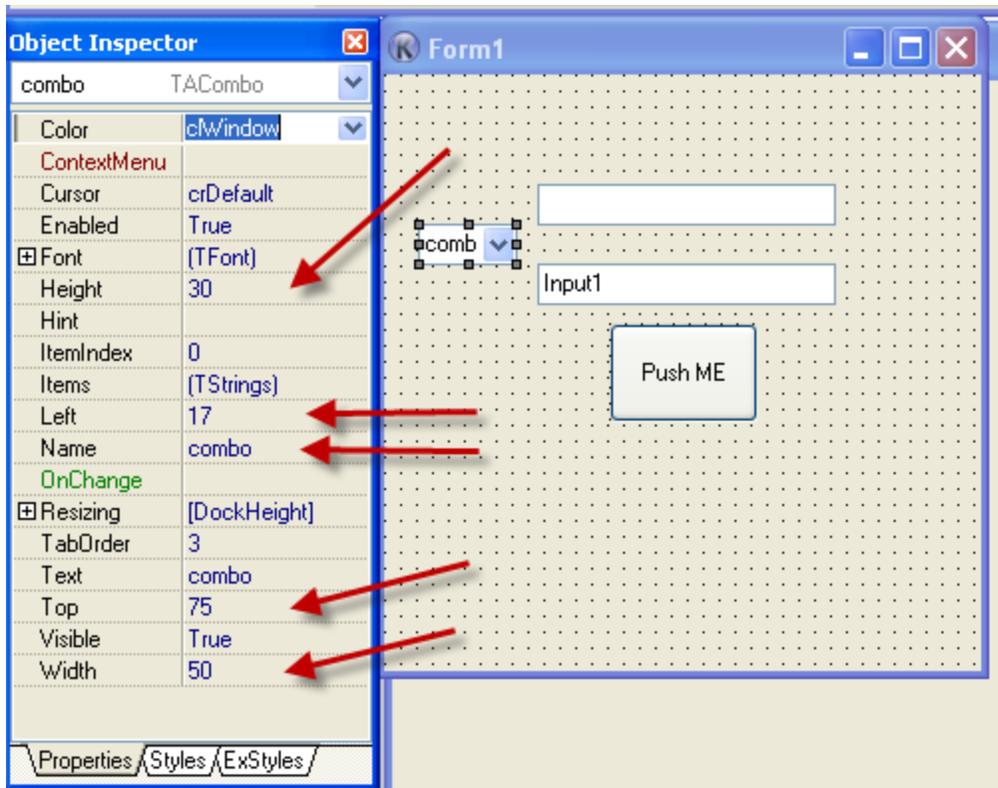
Now that we have configured the size, shape, and placement of the combo control we need to set its values. To do this we need to use a feature in the object explorer that we have not encountered to this point … "Items". If you click on the value next to items in the object explorer you should see a window pop up with title of String List Editor that lets you add a series of strings. These are the values a user will see in the drop down when they activate the control.
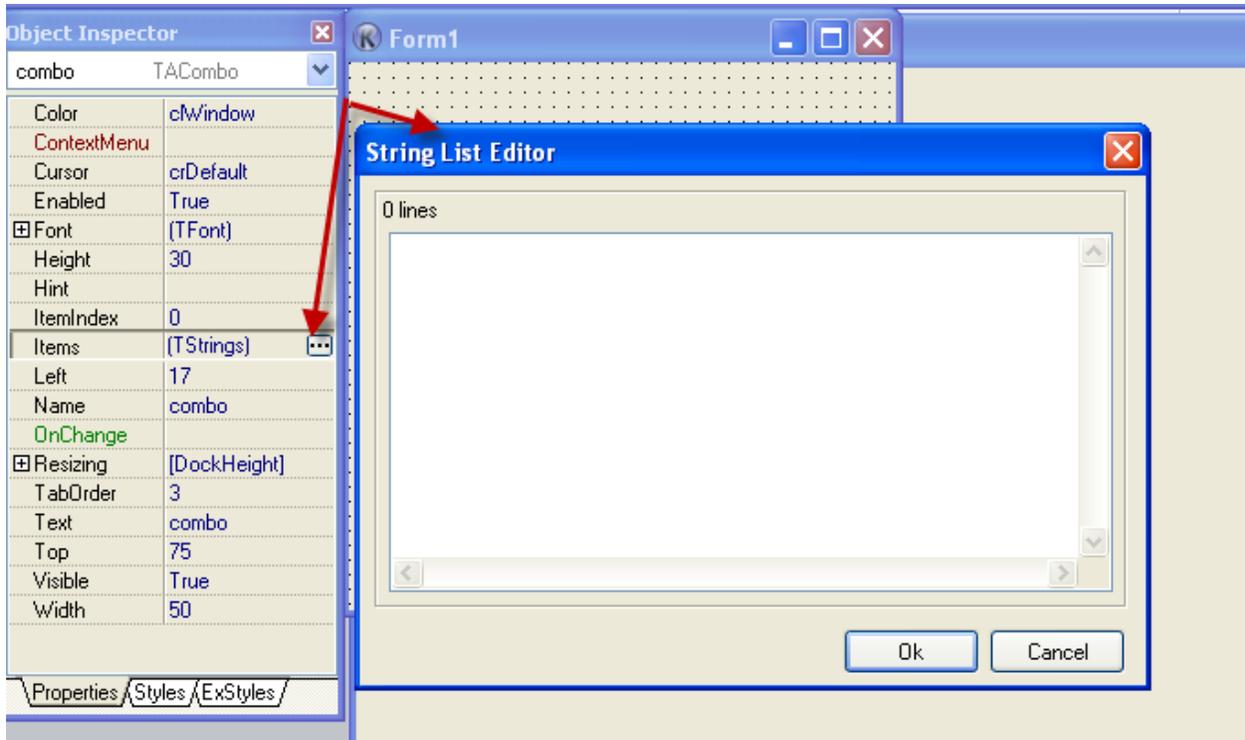
Simply type in the arithmetic operators we used in Chapter followed by a hard return and you see the following.
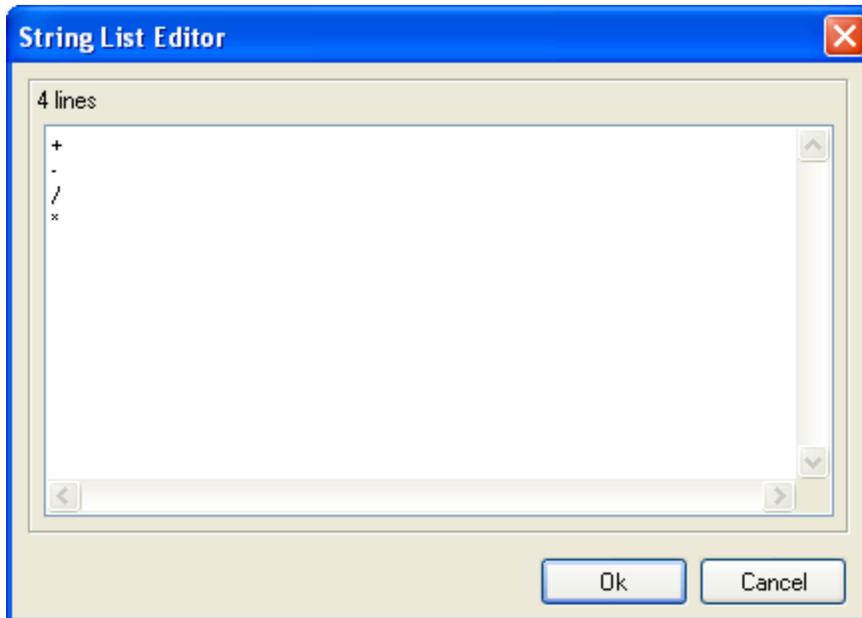
If you want to see how this will look when you run it you can always select the run command from the Koda menu.
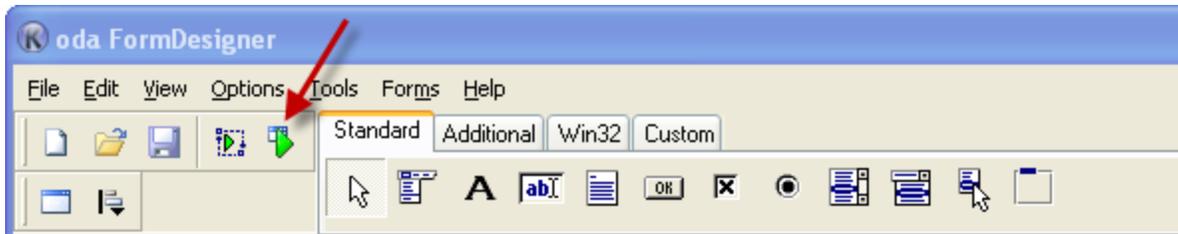
When you do that you should see running version of the GUI. It won't do anything yet – but will look like what a user would see when they run your program. You may notice that my GUI does not say the name of the control in the box. That is because I took the extra step of deleting the text value for the combo box control in the object explorer. Try deleting it yourself and then run the GUI afterward to see if yours looks the same.

Code

Chapter 10 KodaCalculatorGUI.kxf (This is the GUI file created by Koda that you can use to see if yours came out the same.
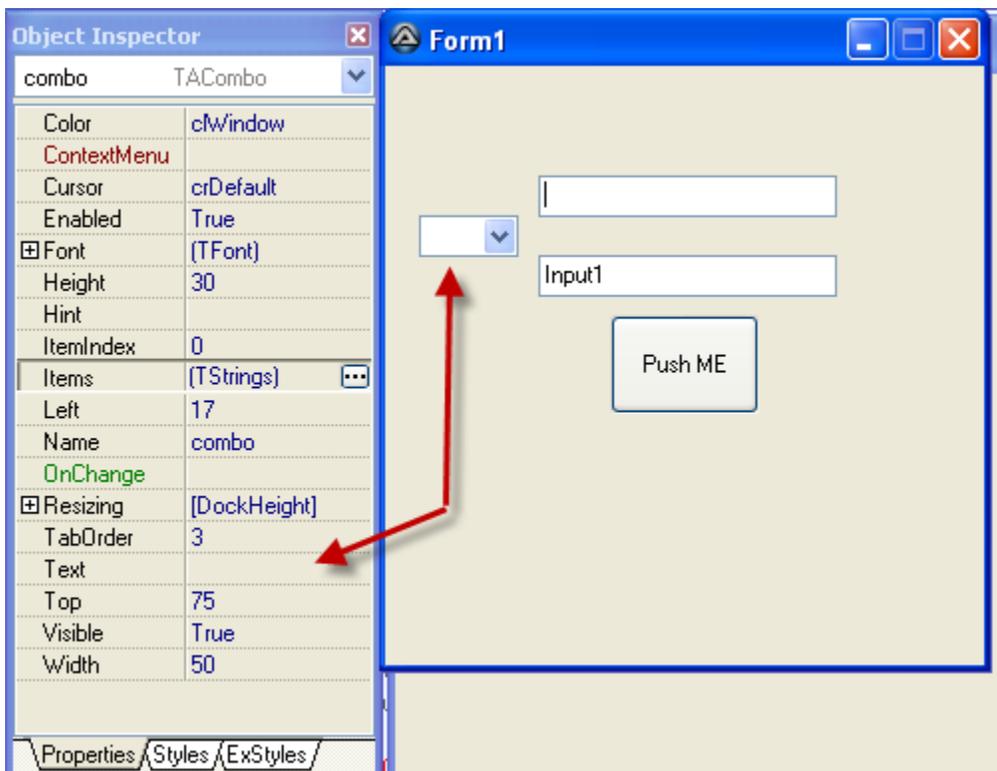
Step 6:

Wiring the Koda GUI to our application: so how do we get what we create in Koda into a script that we can edit and run as part of our program? Koda has a button that we can use to generate the code which we can then cut and paste in SciTE or our favorite text editor. If you press the button you will see a window popup that allows you to select all of the Koda code or just portions for copying and pasting.

📝 NOTE : You may want to select only a portion of the code if you were toggling back and forth making minor adjustments rather than selecting the entire Koda generated text. This is especially true if you have made any modifications to what Koda generated in your script because any such modifications would be overwritten if you perform a wholesale copy and paste.

Press the code button. You will see a window open with the title "Code".
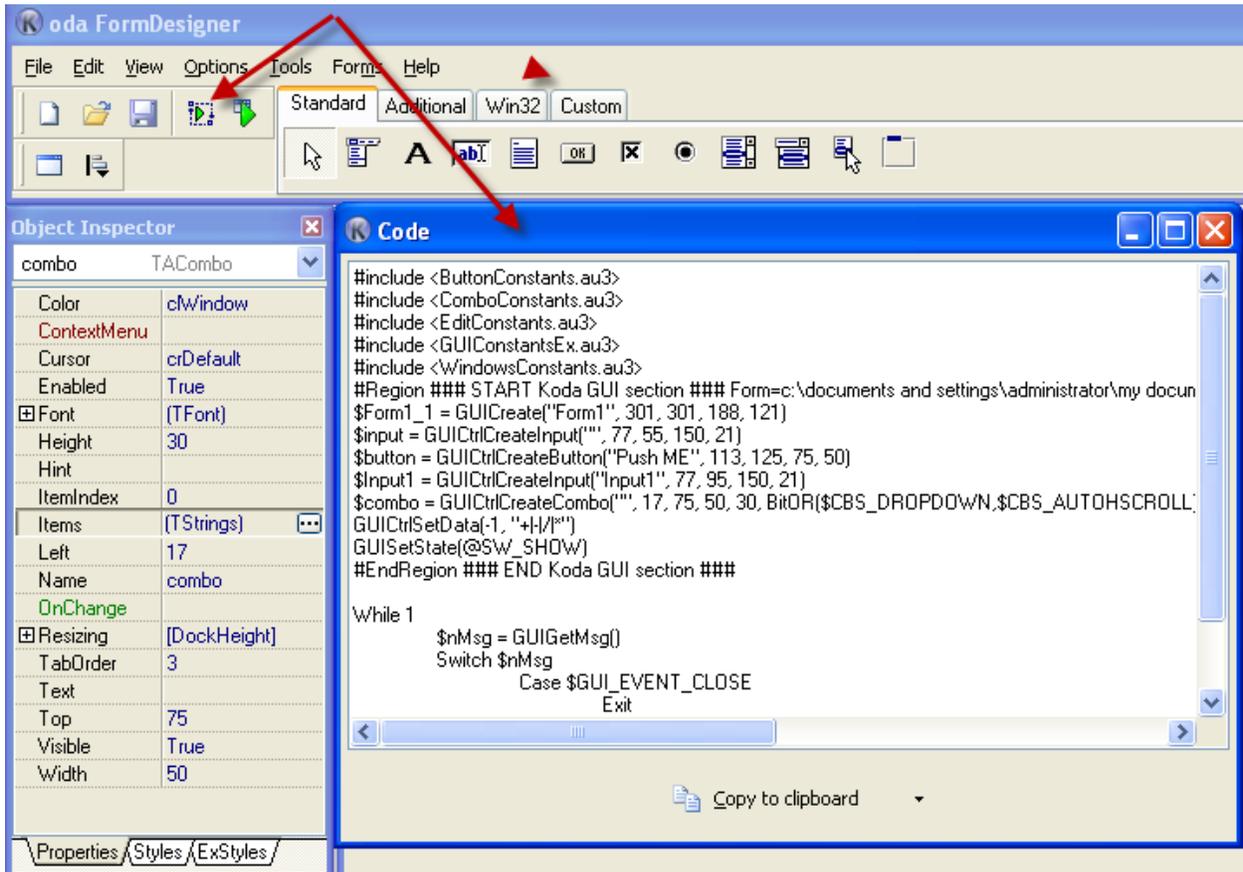
The bottom of the code window has a button that says "Copy to clipboard". Press that button then open SciTE and paste the code.

```
 1    #include <ButtonConstants.au3>
 2    #include <ComboConstants.au3>
 3    #include <EditConstants.au3>
 4    #include <GUIConstantsEx.au3>
 5    #include <WindowsConstants.au3>
 6    #Region ### START Koda GUI section ### Form=c:\documents and settings\administrator\my
 7    $Form1_1 = GUICreate("Form1", 301, 301, 188, 121)
 8    $input = GUICtrlCreateInput("", 77, 55, 150, 21)
 9    $button = GUICtrlCreateButton("Push ME", 113, 125, 75, 50)
10    $Input2 = GUICtrlCreateInput("", 77, 95, 150, 21)
11    $combo = GUICtrlCreateCombo("", 17, 75, 50, 30, BitOR($CBS_DROPDOWN,$CBS_AUTOHSCROLL))
12    GUICtrlSetData(-1, "+|-|/|*")
13    GUISetState(@SW_SHOW)
14    #EndRegion ### END Koda GUI section ###
15
16    While 1
17        $nMsg = GUIGetMsg()
18        Switch $nMsg
19            Case $GUI_EVENT_CLOSE
20                Exit
21
22        EndSwitch
23    WEnd
```

Notice that the code we pasted has several sections. First, there are the includes at the top. Those have been automatically generated for us by Koda based on the UI elements we selected to be part of our GUI. Next there is a block of GUI code that start and end with some comments using the hash sign (#). That is our GUI.   That code is similar to what we created by hand except Koda managed the placement of the controls based on our settings in the object explorer instead of using variables. Finally, there is a while loop at the bottom to listen for GUI events. You may also note that the exit event for $GUI_EVENT_CLOSE has also been created automatically. Save the new script as KodaCalculator.au3.

There are only two remaining differences from the calculator example we created in Chapter 9. First, we need to include "calc.au3" at the top as shown below. Make sure it is in the same directory as where you are saving this new script. Next, open the original calculator code from Chapter 9, copy the entire while loop, and replace the while loop for KodaCalculator.au3 with the copied while loop. The modified code appears below.

Code Chapter 10 Example 1

```
1    #include <ButtonConstants.au3>
2    #include <ComboConstants.au3>
3    #include <EditConstants.au3>
4    #include <GUIConstantsEx.au3>
5    #include <WindowsConstants.au3>
6
7    #include <calc.au3>   ⬅
8
9    #Region ### START Koda GUI section ### Form=c:\documents and settings\administrator\my d
10   $Form1_1 = GUICreate("Form1", 301, 301, 188, 121)
11   $input = GUICtrlCreateInput("", 77, 55, 150, 21)
12   $button = GUICtrlCreateButton("Push ME", 113, 125, 75, 50)
13   $Input2 = GUICtrlCreateInput("", 77, 95, 150, 21)
14   $combo = GUICtrlCreateCombo("", 17, 75, 50, 30, BitOR($CBS_DROPDOWN,$CBS_AUTOHSCROLL))
15   GUICtrlSetData(-1, "+|-|/|*")
16   GUISetState(@SW_SHOW)
17   #EndRegion ### END Koda GUI section ###
18
19
20   ;GUICtrlCreateInput(
21   While 1
22       $nMsg = GUIGetMsg()
23       Switch $nMsg
24           case $button
25               $firstNum=GUICtrlRead($input)
26               $secondNum=GUICtrlRead($input2)
27               $operator=GUICtrlRead($combo)
28               _mathWithAnswers($firstNum,$secondNum,$operator)
29           Case $GUI_EVENT_CLOSE
30               Exit
31
32       EndSwitch
33   WEnd
```

> Entire section copied from Chapter 9.

If you run this code you should see the calculator that we created in Chapter 9. Of course, the main difference is that we create the GUI using Koda instead of coding it by hand.

REMINDER: Koda is a tool that can be used to create GUIs for your AutoIt scripts. One advantage to using Koda is that you can drag and drop your components onto your GUI with a visual interface that provides quick access to repositioning and sizing of controls. It also contains several features that save time such as automatic vertical and horizontal centering. Controls named in the Koda object explorer do not require a "$" character. The "$" will be generated automatically by Koda along with the rest of the GUI code. Koda generated code can be copied and pasted into your script.

Chapter 11: What's in a name? String management.

In Chapter 1 we learned about various data types including strings – text that appears within quotation marks. You may find that when you write your scripts you may need to manipulate a string. We have already seen examples of concatenating two strings together with the ampersand "&" operator but what if you wanted to replace certain words? Trim off portions of a phrase or sentence? Add delimiters, etc.? All of those things (and more) are possible with AutoIt's built in string functions.   We will examine several of the most popular functions – the rest can be found in the AutoIt help file.

StringInStr

Searching for text within a string is a common task.   AutoIt's StiringinStr function does just that. As the name would suggest it searches for a string (i.e. a substring) within a string. If successful, it will return the position of the substring. The function, as documented by the AutoIt help file, appears below.

StringInStr

Checks if a string contains a given substring.

```
StringInStr ( "string", "substring" [, casesense = 0 [, occurrence = 1 [, start = 1 [,
count]]]] )
```

Parameters

string	The string to evaluate.
substring	The substring to search for.

casesense	[optional] Flag to indicate if the operations should be case sensitive.
	    $STR_NOCASESENSE (0) = not case sensitive, using the user's locale (default)
	    $STR_CASESENSE (1) = case sensitive
	    $STR_NOCASESENSEBASIC (2) = not case sensitive, using a basic/faster comparison
	Constants are defined in StringConstants.au3

occurrence	[optional] Which occurrence of the substring to find in the string. Use a negative occurrence to search from the right side. The default value is 1 (finds first occurrence).
start	[optional] The starting position of the search.
count	[optional] The number of characters to search. This effectively limits the search to a portion of the full string. See remarks.

It has several parameters starting with the string that will be searched. The only other mandatory parameter is the substring that you wish to find. You can also feed the function some optional parameters that dictate whether or not the search is case sensitive, which occurrence you are searching for (in the case where there is more than one), where to start the search (if you don't want to start at

the beginning of the string, and a count should you desire to limit the number of characters searched within the string.

Let's take a look at this function in action. If we create a simple string and then search for a term we can store the result in a variable and display it in a message box.

Code Chapter 11 Example 1

```
1    #include <MsgBoxConstants.au3>
2    $myString = "The quick brown fox jumped over the log"
3    $foxPosition = StringInStr($myString,"fox")
4    MsgBox($MB_OK,"",$foxPosition)
```

When we run the code we will see a message box that displays the position of the substring "fox" - which starts at the 17th character of $myString.

```
17

    OK
```
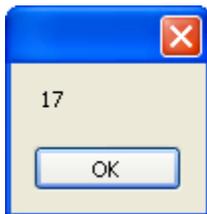
What would happen if we switched the substring that we are searching for to "donkey"? The message box would show us a "0" because that is what the function returns when the search fails.

StringLen

What if we wanted to know how many characters were in our initial string (i.e. $myString)? How would we figure that out? As you may have guessed, there is another function for that. The StringLen function will return the length of a given string. The function has only one parameter – the string that you wish to measure.

StringLen

Returns the number of characters in a string.

```
StringLen ( "string" )
```

Parameters

string	The string to evaluate.

Let's revise the code we used to search our string and measure it instead.

Chapter 11 Example 2

```
1    #include <MsgBoxConstants.au3>
2    $myString = "The quick brown fox jumped over the log"
3    $myStringLen = StringLen($myString)
4    MsgBox($MB_OK,"",$myStringLen)
5
```
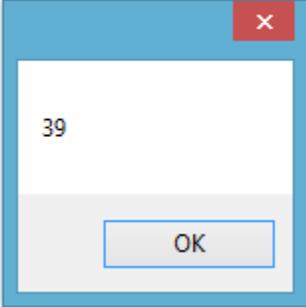
If we do that we can see that our original string was 39 characters in length. The string length can be very powerful when used in conjunction with other string functions. More on that in a moment.

StringReplace

String replace allows you to search a given string for a substring and replace it with another string. You can optionally dictate how many replacements to make and whether or not your search is case sensitive.

StringReplace

Replaces substrings in a string.

```
StringReplace ( "string", "searchstring/start", "replacestring" [, occurrence = 0 [,
casesense = 0]] )
```

Parameters

string	The string to evaluate.
searchstring/start	The substring to search for or the character position to start the replacement.
replacestring	The replacement string.
occurrence	[optional] The number of times to replace the searchstring. Use a negative occurrence to replace from the right side. 0 = all searchstrings will be replaced (default)

casesense	[optional] Flag to indicate if the operations should be case sensitive.
	    $STR_NOCASESENSE (0) = not case sensitive, using the user's locale (default)
	    $STR_CASESENSE (1) = case sensitive
	    $STR_NOCASESENSEBASIC (2) = not case sensitive, using a basic/faster comparison
	Constants are defined in StringConstants.au3

If we look at our starting string we can change fox to dog using this function.

Code Chapter 11 Example 3

```
1    #include <MsgBoxConstants.au3>
2    $myString = "The quick brown fox jumped over the log"
3    $newString=StringReplace($myString,"fox","dog")
4    MsgBox($MB_OK,"",$newString)
```
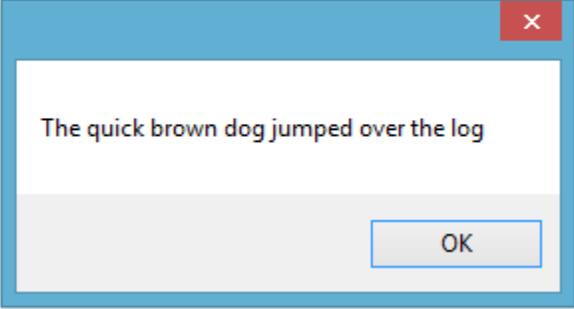
The quick brown dog jumped over the log

OK

There are many practical uses for this function including instances in which you desire to change a data file or "cleanse" it. Example: a work colleague wants to upload his or her pipe delimited file (i.e. a file where the values are separated by pipes "|") to a website. However, the website only accepts comma separated files. What could you do? You could read the file and use this function to convert the pipes to commas.

NOTE: We will cover basic file and directory management functions in the following chapter.

StringSplit

String split is another powerful function that will break a string into substrings based on a given delimiter and return the substrings in an array. The function has two mandatory arguments: the string that you would like to split and the text you are using to split it (i.e. the delimiter). There is also an optional flag parameter that allows you to dictate the behavior of the split. For example, the default behavior of the function returns an array. By default, the first element of the array contains a value representing the number of strings returned. The implication is that your substrings start at the index of 1 instead of 0 since 0 is occupied by the count. You can change that in the flag parameter with a value of 2 and the count will be disabled as the first element giving you a zero based index for your returned strings.

NOTE: The optional flag parameter says "…add multiple flag values together if required:" Therefore, if you wanted to use the $STR_ENTIRESPLIT and $STR_NOCOUNT flags together you would set the value of the flag parameter to 3 (1) + (2).

StringSplit

Splits up a string into substrings depending on the given delimiters.

```
StringSplit ( "string", "delimiters" [, flag = 0] )
```

Parameters

string	The string to evaluate.
delimiters	One or more characters to use as delimiters (case sensitive).

flag	**[optional]** Changes how the string split works, add multiple flag values together if required:
	    $STR_CHRSPLIT (0) = each character in the delimiter string will mark where to split the string (default)
	    $STR_ENTIRESPLIT (1) = entire delimiter string is needed to mark the split
	    $STR_NOCOUNT (2) = disable the return count in the first element - effectively makes the array 0-based (must use UBound() to get the size of the array in this case).
	Constants are defined in StringConstants.au3

Let's create our own example by storing a list of names separated by commas (i.e. a string) in a variable called $names. Then we will use StringSplit to break the names into an array using the comma delimiter. Finally, we will use an include to extend AutoIts array functionality so that we can display the results of the array on our screen with the _ArrayDisplay function (more on that later when we cover user defined functions or UDFs).

Code Chapter 11 Example 4

```
1   #include<Array.au3>
2   $names=" Bob, Jane, Sammy, David, Jonah, Billy, Duke, Lizzie, Greg"
3   $nameArray=StringSplit($names,",")
4   _ArrayDisplay($nameArray,"Names Array")
5
```

Names Array

Row	Col 0
[0]	9
[1]	Bob
[2]	Jane
[3]	Sammy
[4]	David
[5]	Jonah
[6]	Billy
[7]	Duke
[8]	Lizzie
[9]	Greg

The first element tells us the number of substrings (in this case names)

We can see from the displayed array that our string contained nine names that are now stored in an array. Why would this be useful? Now we can loop through them in the array with one of our may loop functions from Chapter 6. For example, we could personalize a message to each person:

Code Chapter 11 Example 5

```
1    #include<Array.au3>
2    $names=" Bob, Jane, Sammy, David, Jonah, Billy, Duke, Lizzie, Greg"
3    $nameArray=StringSplit($names,",")
4    ;_ArrayDisplay($nameArray,"Names Array")
5
6    for $a=1 to UBound($nameArray)-1
7        ConsoleWrite(@crlf&"This is a special message for"&$nameArray[$a]&@crlf)
8    Next
```

commented out - will be ignored by program

Note the for loop starts at 1 to avoid the 0 index that stores the count.

These are the results

The results will appear at the bottom of your Scite IDE (the console)

```
This is a special message for Bob

This is a special message for Jane

This is a special message for Sammy

This is a special message for David

This is a special message for Jonah

This is a special message for Billy

This is a special message for Duke

This is a special message for Lizzie

This is a special message for Greg
```

StringTrimLeft

As the name would suggest, string trim left allows you to "trim" (i.e. chop off) a portion of a given string by providing a given number of character to trim from the left hand side. It has only two parameters both of which are mandatory. They are the string to trim, and the count which will dictate the number of characters being trimmed. A successful operation will return the new string. If you trim too many (i.e. go out of bounds by trimming more characters than there are) an empty string will be returned.

StringTrimLeft

Trims a number of characters from the left hand side of a string.

```
StringTrimLeft ( "string", count )
```

Parameters

string	The string to evaluate.
count	The number of characters to trim.

Return Value

Returns the string trimmed by *count* characters from the left.

Remarks

If *count* is out-of-bounds, an empty string is returned.
StringTrimLeft($str, $n) is functionally equivalent to StringRight($str, StringLen($str) - $n)

This function is particularly useful when used in conjunction with StringInStr and StringLen. Let's reexamine our original string "The quick brown fox jumped over the log". We know that the length of the string is 39. We also know that fox appears at position 17 and is three characters long. What would we need to do in order to trim off everything up to the word "jumped"?

Code Chapter 11 Example 6

```
1  #include <MsgBoxConstants.au3>
2  $myString = "The quick brown fox jumped over the log"
3  $foxPosition = StringInStr($myString,"fox")
4  $trimmedString=StringTrimLeft($myString,$foxPosition+3)
5  MsgBox($MB_OK,"",$trimmedString)
```

jumped over the log

OK
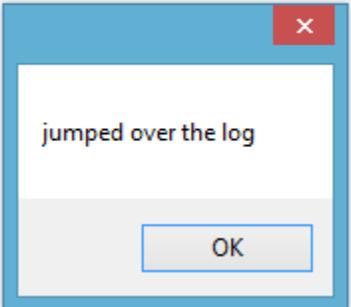
We would need to create a new variable to store the trimmed string returned from the StringTrimLeft function and trim the number of characters returned by our initial search + 3 (to cover the "o", "x" and whitespace. Is there another way where we would not have to manually count the number of

characters to get our desired result? How about if we stored the search term in a variable and measured its length with a function? Then we could add that value to the value returned from our search. It would accomplish the same thing. It might look something like this:

Code Chapter 11 Example 7

```
1    #include <MsgBoxConstants.au3>
2    $myString = "The quick brown fox jumped over the log"
3    $searchTerm="fox"
4    $searchtermLen=StringLen($searchTerm)
5    $foxPosition = StringInStr($myString,$searchTerm)
6    $trimmedString=StringTrimLeft($myString,$foxPosition+$searchtermLen)
7    MsgBox($MB_OK,"",$trimmedString)
```
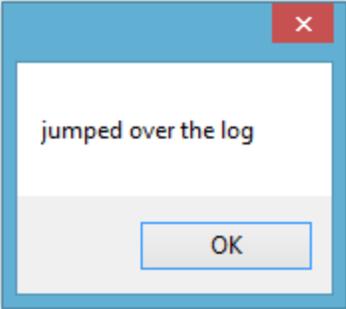
jumped over the log

OK

You can start to envision different ways to use the functions with another. Now what would you do if you wanted to trim some characters from the right hand side?

StringTrimRight
StringTrimRight works the same way as StringTrimLeft but the trimming is calculated from the right side instead of the left:

StringTrimRight

Trims a number of characters from the right hand side of a string.

```
StringTrimRight ( "string", count )
```

We know we want to chop off the word "log" from the right side and that the word log is three characters. It might look something like this:

Code Chapter 11 Example 8

```autoit
1   #include <MsgBoxConstants.au3>
2   $myString = "The quick brown fox jumped over the log"
3   $newString=StringTrimRight($myString,3)
4   MsgBox($MB_OK,"",$newString)
```

The quick brown fox jumped over the

OK
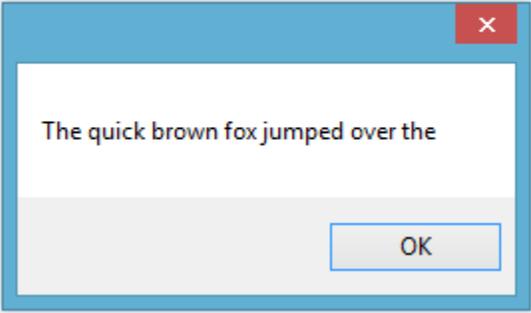
StringLeft and StringRight

NOTE: StringLeft is not the same as StringTrimLeft. It is a different function that we have not covered.   StringLeft returns the leftmost portion of a string based on the number of characters provided. It has a counterpart function StringRight that does the same thing but from the right hand side of the string (which is also different from StringTrimRight).

The functions are different from their trim counterparts but can still be used for that purpose (as well as other purposes). For example, if you want to trim from the right instead of the left while you could use StringTrimRight the AutoIt help file points out you could use also use StringLeft and do something like this: StringLeft($str, StringLen($str) - $n) where $str is the string to be evaluated and $n is the number of characters to from the left position to return as a new string.

What if we want to chop off the word log again from the end of our sentence using these functions instead of StringTrimRight? Let's try it:

```
1    #include <MsgBoxConstants.au3>
2    $myString = "The quick brown fox jumped over the log"
3    $trimmedString=StringLeft($myString,StringLen($myString)-3)
4    MsgBox($MB_OK,"",$trimmedString)
```

The quick brown fox jumped over the

OK
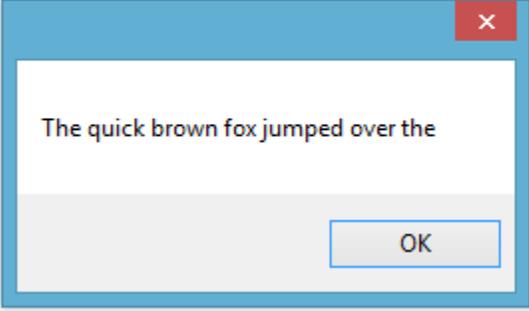
You can see that in this example we have successfully returned a new string without the word "log.

The simplified approach did not worry about creating a search term and measuring its length because in this case it was all very manageable. The more complex examples were provided because it won't always be that simple. You may need to create several measurements and user a variety of functions to create your desired outcome.

REMINDER: You may have need in your coding to manipulate strings. There are many powerful string functions built into AutoIt that can be used to do things like measure, search, split, and trim strings. You may need to use several functions in concert to obtain your desired result.

Chapter 12: Files and Directories

It won't be long until you'll have a need to create files, read them, move them etc. You will also need to understand how to work with the directories where the files are stored.   As in past chapters we will focus on some of the more popular functions to get you up and running with files and directories.

Files

File creation – FileOpen / FileWrite / FileClose

We have already created a slew of variables that we used to store different pieces of data within our scripts.   But that information is stored in the program with no way to "get it out". Moreover, when the program closes the information is lost. One way to produce externally accessible information and save it on a more permanent basis is to create a file and write the information to it. There are three steps to creating a file with AutoIt. First, we have to "open" a new file with the FileOpen function (which can also be used to open an existing file).   FileOpen has two parameters: the filename (which should include

the path to our file) and optional parameter of "mode" used to dictate certain options such as whether or not we are overwriting data in a file, appending, etc.

One line of code will create our new file:

Code Chapter 12 Example 1

```
2    $myFile = FileOpen(@ScriptDir&"\myfile.txt",2)
```

1. This is a macro. We will cover macros in the next chapter. For now we need to understand that this macro is a shortcut that provides the path to the directory in which our script is located. If we did not have that shortcut we would have to use the full path such as "c:\documents and settings\user\files\autoit\myfile.txt". Another benefit is that our program may be run on other people's machines that have different folders and directories. If their folders are different your program may not work. Using the macro ensures that whatever the path – your script will find it. Note the use of the ampersand and the backslash before the filename. That adds the backslash and the file name to the script path.
2. This is what we are calling the file that we are creating. In this case "myfile.txt".
3. This is the optional mode parameter. In this case we are using 2 which signals to the function that we would like to overwrite whatever is in the file and erase everything.

FileOpen returns the "handle" that we store in our $myFile variable when that line of code is executed. You can use that handle for subsequent file functions.

Now we need to add some data to our file. If we pull up our example from chapter 11 we can use our go to string: "The quick brown fox jumped over the log".   That string was stored in a variable. Let's take that variable and write it to our newly created file using the FileWrite function. FileWrite has two parameters the filename or filehandle.

FileWrite

FileWrite

Write text/data to the end of a previously opened file.

```
FileWrite ( "filehandle/filename", "text/data" )
```

Parameters

filehandle/filename	The handle of a file, as returned by a previous call to FileOpen(). Alternatively, you may use a string filename as the first parameter.
text/data	The text/data to write to the file. The text is written as is - no @CR or @LF characters are added. See remark for data type.

Code Chapter 12 Example 2

```
2    $myFile = FileOpen(@ScriptDir&"\myfile.txt",2)
3    $myString = "The quick brown fox jumped over the log"
4    FileWrite($myFile,$myString)
```

In the above example we called the FileWrite function with the handle for the filename we had created in the first line of code – which means we are writing to the file called myfile.txt located in the same directory as our script. The second parameter dictates what we are writing – in this case our favorite string about the fox and log. At this point, if you searched for the file in the directory you would see it but it would appear to be blank. Why? Because before the file can be used we must close it. If it is still open it won't be useable. To close it we call the function FileClose which has only one parameter Filename/handle.

```
1    $myFile = FileOpen(@ScriptDir&"\myfile.txt",2)
2    $myString = "The quick brown fox jumped over the log"
3    FileWrite($myFile,$myString)
4    FileClose($myFile)
```

This last line of code closes the file and makes it available for use. Now if you go to your script directory you should see the file.

    📄 myfile

If you were to open the file you should see the following:
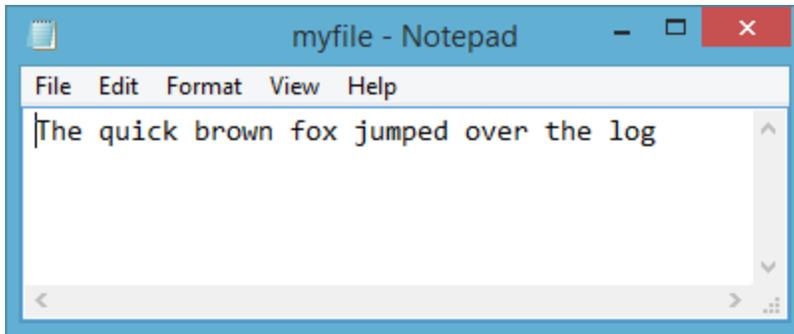
Just like that with a few lines of code we have created a file. That was a very simplistic example. What if we wanted to create a file of the personalized greetings we generated to the console when we explored the StringSplit function in the previous chapter. What would that look like?

Code Chapter 12 Example 3

```
1   $myFile = FileOpen(@ScriptDir&"\myfile.txt",2)
2   $names=" Bob, Jane, Sammy, David, Jonah, Billy, Duke, Lizzie, Greg"
3   $nameArray=StringSplit($names,",")
4
5   for $a=1 to UBound($nameArray)-1
6       FileWrite($myFile,@crlf&"This is a special message for"&$nameArray[$a]&@crlf)
7   Next
8   FileClose($myFile)
```

1. We create our file the same we did in the prior example. The optional mode parameter will wipe the previous contents.
2. We use the string containing our names from Chapter 11.
3. We break the names into an array as we did in Chapter 11 using StringSplit.
4. This time, instead of writing the personalized greeting to the console that we could view in our SciTE editor we are writing them to our file.
5. After the for loop ends the script will continue. The next thing we do is close the file.

Now when we re-open our myfile.txt file we will see the following:
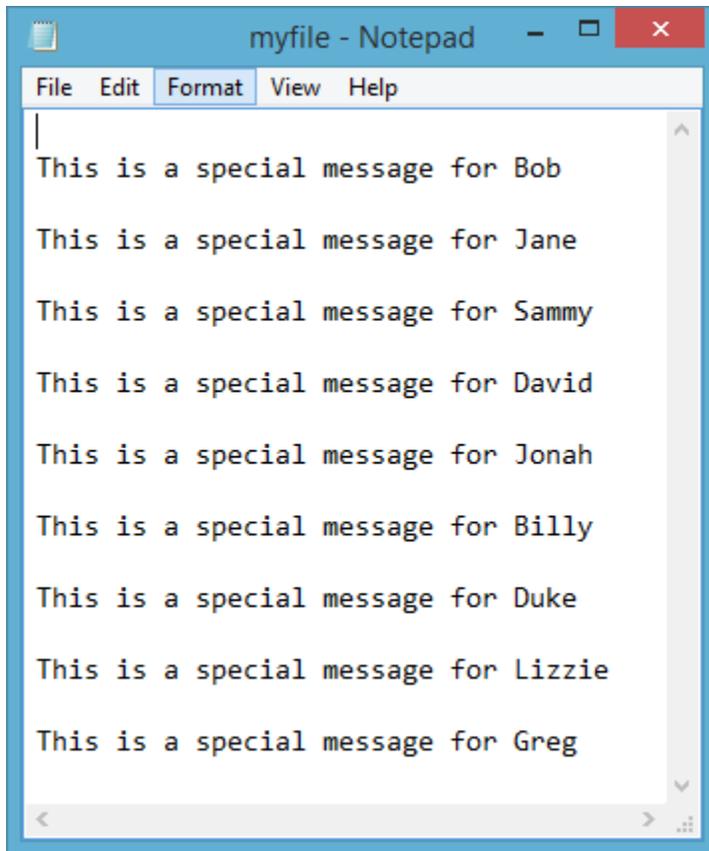
This example was a bit more powerful because we were able to use a for loop to produce a larger file. In addition, our file contains nine rows with different content on each row.

FileRead

To this point we have successfully created a file, written data to it, and "closed" it so that we could access it and review its contents. That last step, where we manually found the file and opened it can also be automated. FileRead is an AutoIt function that, as the name suggests, allows us to programmatically read a file. It has one mandatory parameter for the filename / handle. It also has an optional parameter if you want to limit the number of characters to read. If only we had a file to test this on … oh wait, we do, we just created one.

Code Chapter 12 Example 4

```
1    #include <MsgBoxConstants.au3>
2    $fileIn = FileRead(@ScriptDir&"\myfile.txt")
3    MsgBox($MB_OK,"",$fileIn)
```

In the code above we created a variable called $fileIn to store the contents of the file read using the FileRead function. The file we are reading is the same one we just created. Therefore, when we put the contents of the $fileIn variable into a message box we see the following:

This is a special message for Bob

This is a special message for Jane

This is a special message for Sammy

This is a special message for David

This is a special message for Jonah

This is a special message for Billy

This is a special message for Duke

This is a special message for Lizzie

This is a special message for Greg

OK

NOTE: if you are only interested in reading a single line from the file try FileReadLine instead. It has a parameter for the filename / handle and an optional parameter for a specific line.

FileCopy

FileCopy is worth mentioning. It allows you to create a copy of a given file. It has three parameters: source (i.e. the source file you wish to copy), dest (the destination to place the copy) and a flag that allows you to dictate whether or not the copy will refrain from overwriting, overwrite, and/or create a path if the path you provided does not exist. A ingle line of code can create a duplicate of our file and place it on our desktop:

Code Chapter 12 Example 5

```
1    FileCopy ( @ScriptDir&"\myfile.txt",@DesktopDir)
```

You may notice that this example introduces a second macro – @DesktopDir which is a path to the users desktop directory. If you run that line of code you will find a copy of the file we created on your desktop.

REMINDER: we will cover macros in the following chapter.

FileOpenDialog

There is another nifty file function that will open up a browse for file or directory window so a user can select a given file. It is the FileOpenDialog function. The AutoIt help file shows the parameters this function accepts and their meaning.

FileOpenDialog

Initiates a Open File Dialog.

```
FileOpenDialog ( "title", "init dir", "filter" [, options = 0 [, "default
name" [, hwnd]]] )
```

Parameters

title	Title text of the Dialog GUI.
init dir	Initial directory selected in the GUI file tree.
filter	File type single filter such as "All (*.*)" or "Text files (*.txt)" or multiple filter groups such as "All (*.*)|Text files (*.txt)" (See Remarks).

options	[optional] Dialog Options: To use more than one option, BitOR the required values together.
	    $FD_FILEMUSTEXIST (1) = File Must Exist (if user types a filename)
	    $FD_PATHMUSTEXIST (2) = Path Must Exist (if user types a path, ending with a backslash)
	    $FD_MULTISELECT (4) = Allow MultiSelect
	    $FD_PROMPTCREATENEW (8) = Prompt to Create New File (if does not exist)
	Constants are defined in FileConstants.au3

default name	[optional] Suggested file name for the user to open. Default is blank ("").
hwnd	[optional] The window handle to use as the parent for this dialog.

Instead of hard coding the file that we wanted to read in our previous example we could have allowed a user to browse for it and select it.

Code
Chapter 12 Example 6

> Title of the window that will appear when we select our file.

```
1    $file = FileOpenDialog("Select File",@ScriptDir,"Text files (*.txt)")
```

> initial directory to start our file search

> filter: what types of files to look for

That code will create a new file open dialog box with a title of "Select File" that will default to displaying the available files in the initial directory we specified – in this case our script directory. The last parameter is a filter. In our case we are only interested in viewing text files. If you run the code, and you still have the file we created earlier the chapter in your script directory, you should see the following:

> our file from earlier

> Preset to text files thanks to our filter
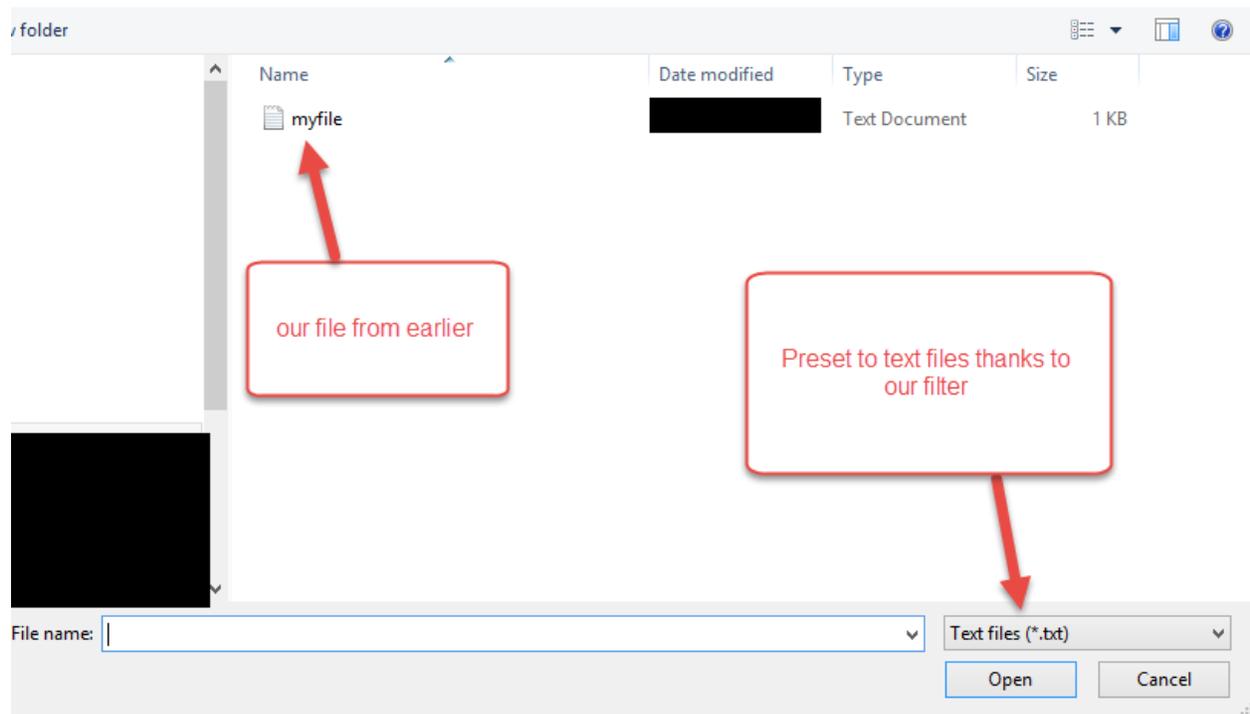
Directories

Directories are the folder structures in which files are stored. The standard functions in AutoIt allow us to copy, create, move, remove, and get the size of directories. You might note that there is no option to rename them. AutoIt does not have a separate function for renaming directories so you need to

DirMove (to move a directory) to rename it (the same is true of files which must rely upon FileMove for file renaming).

DirCreate

The DirCreate function creates a directory and has a single argument for path which will dictate the path of your new folder. We can create a new directory with a single line of code as follows:

Code Chapter 12 Example 7

```
1    DirCreate(@ScriptDir&"\new folder\")
```

Now if you look in the directory where your script is located you should see a new folder called "new folder".

DirGetSize

The folder is currently empty so we would expect it to have a size of zero. Sure enough, if we use the function that is exactly what we will see.

Code Chapter 12 Example 8

```
1    #include <MsgBoxConstants.au3>
2    $size=DirGetSize(@ScriptDir&"\new folder\")
3    MsgBox($MB_OK,"",$size)
```

If we wanted to add a file and re-measure the size of the directory we could make a copy of our text file and place it in the directory as follows:

Code Chapter 12 Example 9

```
1    #include <MsgBoxConstants.au3>
2    FileCopy ( @ScriptDir&"\myfile.txt",@ScriptDir&"\new folder\")
3    $size=DirGetSize(@ScriptDir&"\new folder\")
4    MsgBox($MB_OK,"",$size)
```

347

OK
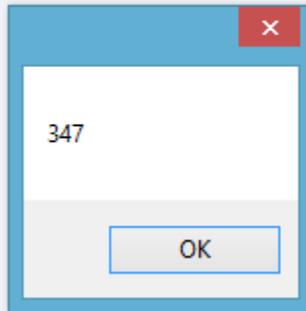
Now the directory has a size of 347 bytes because it contains a copy of the text file that we created earlier.

Let's kill two birds with one stone and rename our directory. We know that there is no function for this so we need to use DirMove. If the source and target destinations are the same we should be able to rename it without actually changing its location.

Code Chapter 12 Example 10

```
1    DirMove(@ScriptDir&"\new folder\",@ScriptDir&"\new name\",1)
```

That single line of code "moved" the directory with a target that contained a new name. If you look in your script directory you should see that the name of the directory changed. Now if want to undo all that we can delete the directory with a single line of code.

Code Chapter 12 Example 11

```
1    DirRemove(@ScriptDir&"\new name\",1)
```

1

Note the use of the optional parameter called "recurse" (you will see it referred by that name in the AutoIt help file). The default behavior of the DirRemove function is to delete a folder only if it is empty. We placed a copy of our text file in the folder. Therefore, the function will not work unless we pass a flag saying that we would also like it to delete the folders contents. If you run that code you should see

that the directory is no longer present in your scripts folder. WARNING: be careful with delete functions – you could wipe out unintended files if you're not careful.

Drives

Disk management is also part of the same grouping of functions that deal with files and directories. However, instead of dealing with files and folders these functions address all things disk. For example, if you wanted to know how much free space you had on your "C:\" drive all you could do the following:

Code Chapter 12 Example 12

```
1   #include <MsgBoxConstants.au3>
2   MsgBox($MB_OK,"",DriveSpaceFree("c:\"))
3
```
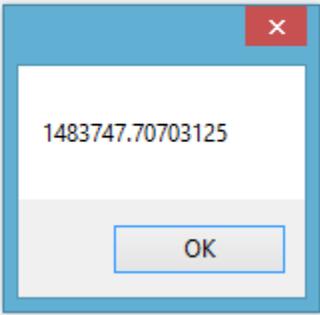
```
                    ×

        1483747.70703125


              OK
```

The free space is returned as a value in Megabytes.

DriveGetDrive

If you want to create an array of all your drives you could use the DriveGetDrive function. It has a single parameter for "type" that dictates which drives are returned. You could do everything from ALL to those that removable, CD ROM, network, etc.

Code Chapter 12 Example 13

```
1   #include<Array.au3>
2   $driveArray=DriveGetDrive("ALL")
3   _ArrayDisplay($driveArray)
4
5
6
7
```
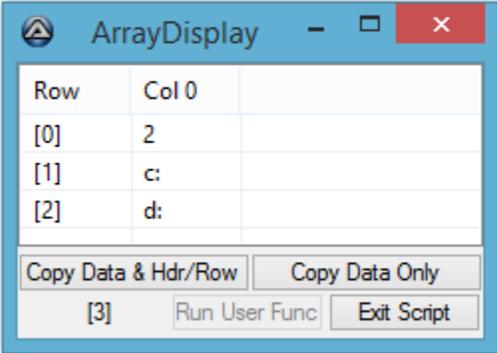
```
   ⬥   ArrayDisplay      –  ☐   ×

   Row        Col 0
   [0]        2
   [1]        c:
   [2]        d:

   Copy Data & Hdr/Row    Copy Data Only
        [3]      Run User Func    Exit Script
```

The above code shows 2 drives (the first element in the array is the number of drives). The two drives are c: and d: respectively.

NOTE: There is a little drive goodie function tucked away under "Misc. Management" in the help file called "CDTray". It lets you automatically open and close the CD Drive on your computer with code. The function has two parameters: the drive and the status. The status can be either "open" or "closed". Try it out: CDTray ("drive", "status").

REMINDER: We can create files, add content, read, copy, move and delete them all from our code. We can also perform a handful of similar operations with directories. Finally, we can get information about disk drives. We also learned that there is no renaming function for files or directories so we must use FileMove and DirMove respectively to rename files and directories.

Chapter 13: Macros

We have already used some macros in our scripts. In Chapter 6 we were exposed to @CRLF which is the macro for a hard carriage return. Later we explored @ScriptDir and @DesktopDir as shortcuts for the paths to our script and desktop directories respectively. There are currently 102 different macros in the AutoIt language. All are preceded by the "@" character to distinguish them from other variables and all are documented in Macro Reference section of the help file. In addition, they are grouped by Autoit-related, Directory, System Info, and Time and Date macros. We will examine a handful of the more popular macros from each grouping.

AutoIt-related macros

We have already covered one example of common AutoIt-related macros. Specifically, @ScriptDir. The macro we use to get the directory path of our scripts. Another prominent macro in this category is:

@error

You will see reference to @error throughout the AutoIt help file. It obtains the status of the error flag set by SetError(). Many of the functions used throughout AutoIt return values that indicate success or failure. However, many of them also set the error flag that can be retrieved with this macro. Consider the example of GUICreate that encountered in Chapter 9. If that function fails it will return "0 if the window cannot be created and sets the @error flag to 1." This means that you can check for the returned value of 0 or the @error value of 1 to see if the GUI creation failed.

Directory macros

We have already used @DesktopDir to place a copy of a file we created on our desktop using FileCopy. The following is a complete listing of directory macros from the help file:

@AppDataCommonDir	Path to Application Data
@DesktopCommonDir	Path to Desktop
@DocumentsCommonDir	Path to Documents
@FavoritesCommonDir	Path to Favorites
@ProgramsCommonDir	Path to Start Menu's Programs folder
@StartMenuCommonDir	Path to Start Menu folder
@StartupCommonDir	Path to Startup folder

Macros for Current User data.

@AppDataDir	Path to current user's Roaming Application Data
@LocalAppDataDir	Path to current user's Local Application Data
@DesktopDir	Path to current user's Desktop
@MyDocumentsDir	Path to My Documents target
@FavoritesDir	Path to current user's Favorites
@ProgramsDir	Path to current user's Programs (folder on Start Menu)
@StartMenuDir	Path to current user's Start Menu
@StartupDir	current user's Startup folder
@UserProfileDir	Path to current user's Profile folder.

Other macros for the computer system:

@HomeDrive	Drive letter of drive containing current user's home directory.
@HomePath	Directory part of current user's home directory. To get the full path, use in conjunction with @HomeDrive.
@HomeShare	Server and share name containing current user's home directory.
@LogonDNSDomain	Logon DNS Domain.
@LogonDomain	Logon Domain.
@LogonServer	Logon server.
@ProgramFilesDir	Path to Program Files folder
@CommonFilesDir	Path to Common Files folder
@WindowsDir	Path to Windows folder
@SystemDir	Path to the Windows' System (or System32) folder.
@TempDir	Path to the temporary files folder.
@ComSpec	Value of %COMSPEC%, the SPECified secondary COMmand interpreter; primary for command line uses, e.g. Run(@ComSpec & " /k help | more")

REMINDER: Directory macros are shortcuts to popular directories that circumvent the need to type in the entire path. They are also dynamic enough to find the paths on other users computers.

System Info macros

The following is a complete list of the System Info macros from the AutoIt help file.

Macro	Description
@CPUArch	Returns "X86" when the CPU is a 32-bit CPU and "X64" when the CPU is 64-bit.
@KBLayout	Returns code denoting Keyboard Layout. See Appendix for possible values.
@MUILang	Returns code denoting Multi Language if available (Vista is OK by default). See Appendix for possible values.
@OSArch	Returns one of the following: "X86", "IA64", "X64" - this is the architecture type of the currently running operating system.
@OSLang	Returns code denoting OS Language. See Appendix for possible values.
@OSType	Returns "WIN32_NT" for XP/2003/Vista/2008/Win7/2008R2/Win8/2012/Win8.1/2012R2.
@OSVersion	Returns one of the following: "WIN_81", "WIN_8", "WIN_7", "WIN_VISTA", "WIN_XP", "WIN_XPe", for Windows servers: "WIN_2012R2", "WIN_2012", "WIN_2008R2", "WIN_2008", "WIN_2003"".
@OSBuild	Returns the OS build number. For example, Windows 2003 Server returns 3790
@OSServicePack	Service pack info in the form of "Service Pack 3".
@ComputerName	Computer's network name.
@UserName	ID of the currently logged on user.
@IPAddress1	IP address of first network adapter. Tends to return 127.0.0.1 on some computers.
@IPAddress2	IP address of second network adapter. Returns 0.0.0.0 if not applicable.
@IPAddress3	IP address of third network adapter. Returns 0.0.0.0 if not applicable.
@IPAddress4	IP address of fourth network adapter. Returns 0.0.0.0 if not applicable.
@DesktopHeight	Height of the primary display in pixels. (Vertical resolution)
@DesktopWidth	Width of the primary display in pixels. (Horizontal resolution)
@DesktopDepth	Depth of the primary display in bits per pixel.
@DesktopRefresh	Refresh rate of the primary display in hertz.

Let's use one of these in an example to find out the version of our operating system. A single line of code with a macro can display our results in a message box.

Code Chapter 13 Example 1

```
1    #include <MsgBoxConstants.au3>
2    MsgBox($MB_OK,"",@OSVersion)
```
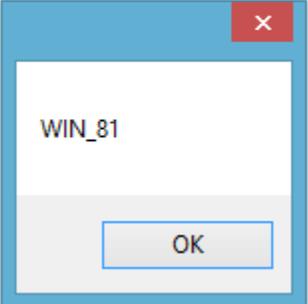
WIN_81

OK

As you can see from the output above, my machine is currently running Windows version 8.1.

Time and Date macros

Time and date macros can be used to find the current second, minute, hour, day, month, year etc. The following is a complete listing of these macros from the AutoIt help file:

Macro	Description
@MSEC	Milliseconds value of clock. Range is 00 to 999. The update frequency of this value depends on the timer resolution of the hardware and may not update every millisecond.
@SEC	Seconds value of clock. Range is 00 to 59
@MIN	Minutes value of clock. Range is 00 to 59
@HOUR	Hours value of clock in 24-hour format. Range is 00 to 23
@MDAY	Current day of month. Range is 01 to 31
@MON	Current month. Range is 01 to 12
@YEAR	Current four-digit year
@WDAY	Numeric day of week. Range is 1 to 7 which corresponds to Sunday through Saturday.
@YDAY	Current day of year. Range is 001 to 366 (or 001 to 365 if not a leap year)

If we wanted to display the current date we could do something like this:

Code Chapter 13 Example 2

```
1    #include <MsgBoxConstants.au3>
2    MsgBox($MB_OK,"",@MON&"/"&@MDAY&"/"&@YEAR)
```

Code Chapter 13 Example 3

If we wanted to create the date with a timestamp we might alter that a bit as follows:

```
1    #include <MsgBoxConstants.au3>
2    MsgBox($MB_OK,"",@MON&"/"&@MDAY&"/"&@YEAR&" Timestamp: "&@HOUR&":"&@MIN&":"&@SEC)
```

REMINDER: macros are predefined variables that provide shortcuts to certain system information. They are preceded by an @ as opposed to a $ to avoid confusion with other variables.

Chapter 14: User defined functions

User defined functions, sometimes referred to as UDFs, are functions created by users in the AutoIt community that can be included in your scripts through the creation of custom functions discussed in Chapter 7. These UDF take two forms: those that come with AutoIt and are referenced in the help file and those that do not. We have already seen an example of the former when we included the Array.au3 UDF so that we could use the function _ArrayDisplay. In this case, we found a well-documented UDF in the AutoIt help file with a complete listing of all its functions. The UDF functions that come with AutoIt look like other standard functions in all material respects except they are preceded by an underscore – just like the custom functions we wrote in Chapter 7. The reason that the UDF functions appear to be so similar and are part of the standard documentation and files that are downloaded with AutoIt is because they were created and maintained by power users within the AutoIt community and deemed valuable enough to be made available to everyone as part of the standard package. There are however, many more UDFs available – many of which are equally powerful and useful but they are not part of the standard download. You can find them in AutoIt forums as attachments to posts or in the forum download section. They are maintained there by the authors and you can find them by searching the forum. Forum members will often recommend them if you create a post that could be solved through the use of a UDF.

We can't cover every UDF in this section as the UDFs are larger than the actual help file at this point. However, we can pick out a few key UDFs to expand on some of the earlier concepts that we have discussed. Let's start with a closer examination of

Array.au3

We learned about arrays in Chapter 8. If you reflect back on that information you may note that we did not address any specific functions relating to arrays. Why? Because most of the array functions are handled by the Array.au3 UDF (which we had not covered at that point). If you expand the User Defined Function Reference at the bottom of your help file you will see Array Management at the top of the tree. If you expand that section you will see a series of functions that allow you to do all sorts of things with arrays including display (we have already seen this), sort, search and more.

_ArrayDisplay

If we create a new array of popular fruits we can revisit the _ArrayDisplay function:

Code Chapter 14 Example 1

The include for our array UDF

our new array of poplar fruits

The UDF function that allows us to display the array in a

The result of running the _ArrayDisplay function

```
1  #include<Array.au3>
2  dim $fruits[10]=["Grapefruit", "Banana", "Watermelon", "Grape", "Apple", "Guava","Star fruit", "Mango", "Coconut", "Blueberry"]
3  _ArrayDisplay($fruits)
```

Row	Col 0
[0]	Grapefruit
[1]	Banana
[2]	Watermelon
[3]	Grape
[4]	Apple
[5]	Guava
[6]	Star fruit
[7]	Mango
[8]	Coconut
[9]	Blueberry

Every function in the Array Management UDF will require the UDF be included in your script. The first line of the above example includes the file needed to access the additional functions. Next, we created an array containing ten popular fruits. The array has a zero based index so it is numbered $0 - 9$ but it contains 10 values in all. Finally, we called then _ArrayDisplay function and passed it the only mandatory parameter of an array that we want to display. The result is that we see a listview window open with the values of our array. This function is especially useful when you are debugging your scripts. It allows you to visualize the contents of an array and validate that the arrays are formed the way you had intended. If they are not, you can often get an error that you are trying to access information that is outside the bounds of the array. Those errors can be difficult to resolve when you can't see the array to better understand the problem.

_ArraySearch

The _ArraySearch function allows us to search a given array for a value. If successful the index (position in the array) for that value will be returned. If unsuccessful the error flag will be set with information as to the cause of the failure. You can use the @error macro from Chapter 13 to explore the cause of the failure in more detail.

Code Chapter 14 Example 2


```
1  #include<Array.au3>
2  #include <MsgBoxConstants.au3>
3  dim $fruits[10]=["Grapefruit", "Banana", "Watermelon", "Grape", "Apple", "Guava","Star fruit", "Mango", "Coconut", "Blueberry"]
4  $grapePos=_ArraySearch($fruits,"grape")
5  MsgBox($MB_OK,"Position of grape",$grapePos)
```

Using _ArraySearch on our fruits array we can search for the term "grape". Grape is the fourth item in the array. However, because the array has a zero based index the index position for "grape" is 3.

NOTE: _ArrayFindAll is another search function that will return an array of all the indices (positions) of the searched term. You can use this function when you are looking for more than one occurrence.

_ArraySort

NOTE: the values contained in our array appear in the order in which they were entered. They are not sorted by default.

What if wanted to sort the contents of our array alphabetically? We could use the _ArraySort function. Let's try it and combine it with _ArrayDisplay to see the results.

Code Chapter 14 Example 3

```
1  #include<Array.au3>
2  dim $fruits[10]=["Grapefruit", "Banana", "Watermelon", "Grape", "Apple", "Guava","Star fruit", "Mango", "Coconut", "Blueberry"]
3  _ArraySort($fruits)
4  _ArrayDisplay($fruits)
```

Row	Col 0
[0]	Apple
[1]	Banana
[2]	Blueberry
[3]	Coconut
[4]	Grape
[5]	Grapefruit
[6]	Guava
[7]	Mango
[8]	Star fruit
[9]	Watermelon

As you can see, adding a line of code with the _ArraySort function alphabetized the contents of our array.

File Management

The file management UDF contains a handful of powerful functions relating to files and directories. We had to include the array UDF in order to use the array management functions. Likewise, in order to use the file functions under the file management UDF we need to include the UDF in our script with the following: #include <File.au3>.

_FileListToArray

_FileListToArray is very useful as it will return an array of all files and folders (or just one or the other based on your parameters) for a given path. It also has options to filter the results and return the full path for each entry.

_FileListToArray

Lists files and\or folders in a specified folder (Similar to using Dir with the /B Switch)

```
#include <File.au3>
_FileListToArray ( $sFilePath [, $sFilter = "*" [, $iFlag = 0 [,
$fReturnPath = False]]] )
```

Parameters

$sFilePath	Folder to generate filelist for.
$sFilter	[optional] the filter to use, default is *. Search the helpfile for the word "WildCards" For details.
$iFlag	[optional] specifies whether to return files folders or both 0 = (Default) Return both files and folders 1 = Return files only 2 = Return Folders only
$fReturnPath	[optional] If True the full path is appended to the file\folder path, otherwise it's relative to the $sFilePath folder. Default is False.

If we use the _FileListToArray function and look for everything (using the "*" as our filter which means all) in the Windows directory (with one of our handy macros from Chapter 13) we can also use a flag (2) to limit the contents of the array to directories. When we run this code we can see that the function found 71 directories in my version of Windows under the windows folder.

Code Chapter 14 Example 4

```
1   ┌─ #include<Array.au3>
2   └  #include<File.au3>
3      $dirArray= _FileListToArray(@WindowsDir,"*",2)
4      _ArrayDisplay($dirArray,"Windows Directories")
5
6
```
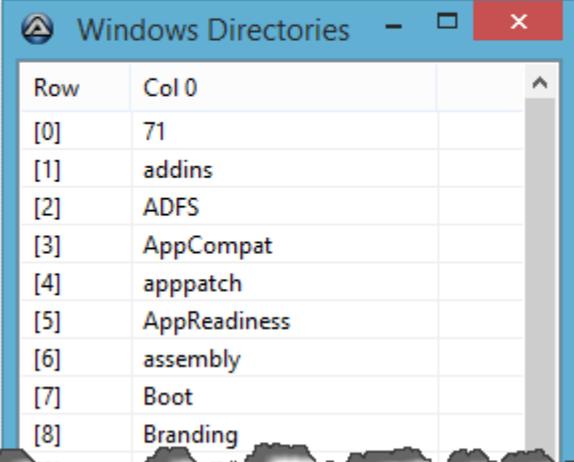
Row	Col 0
[0]	71
[1]	addins
[2]	ADFS
[3]	AppCompat
[4]	apppatch
[5]	AppReadiness
[6]	assembly
[7]	Boot
[8]	Branding

Windows Directories

_FilePrint

_FilePrint enables us to print a text file to our default printer. We can test this function on the file that we created in Chapter 12. If you deleted that file you can recreate it with the script we used - or just choose a different text file. Just make sure it is in the same directory as the code below.

Code Chapter 14 Example 5

```
1      #include<File.au3>
2      _FilePrint(@ScriptDir&"\myfile.txt")
```

Here again, there are many more functions to explore in this UDF. These are just a couple that you have occasion to use early on in your scripting.

GUI UDFs

Most of the GUI components have a UDF that extends their capabilities. They are all grouped under GUI Reference in the latest help file. That grouping includes sub-folders for specific components. You may find that if you want to make a GUI control do something – and it is not clear from the standard help file function as to how to do it – it may be the case that someone has made a UDF for it.

- GUI Reference
  - GuiAVI Management
  - GuiButton Management
  - GuiComboBox Management
  - GuiComboBoxEx Management
  - GuiDateTimePicker Management
  - GuiEdit Management
  - GuiHeader Management
  - GuiImageList Management
  - GuiIPAddress Management
  - GuiListBox Management
  - GuiListView Management
  - GuiMenu Management
  - GuiMonthCal Management
  - GuiReBar Management
  - GuiRichEdit Management
  - GuiScrollBars Management
  - GuiSlider Management
  - GuiStatusBar Management
  - GuiTab Management
  - GuiToolbar Management
  - GuiToolTip Management
  - GuiTreeView Management

GUI reference directory and sub directories from the AutoIt help file

SQLite

We have already discussed file creation in Chapter 12. We covered how to create files and read the information back into your program. However, that may not be the best way to store and retrieve information for you application. If you application stores a lot of data and needs to query it you may find that you need to use a database. SQLite is a truly free open source database.[4] The authors have renounced any ownership and have placed it in the public domain. That means you are free to use it in your projects. To do that, you can use the SQLIte UDF.

NOTE: The use of SQLite requires you to write Structured Query Language (SQL) statements using SQLite's syntax. The database is well documented on their website. However, it is not very intuitive. The good news is that the database is so widely used that a simple Google search for what you are trying to do will usually yield strong results. You can also post your questions to the AutoIt forum.

---

[4] All of the code and documentation in SQLite has been dedicated to the public domain by the authors. All code authors, and representatives of the companies they work for, have signed affidavits dedicating their contributions to the public domain and originals of those signed affidavits are stored in a firesafe at the main offices of Hwaci. Anyone is free to copy, modify, publish, use, compile, sell, or distribute the original SQLite code, either in source code form or as a compiled binary, for any purpose, commercial or non-commercial, and by any means. http://www.sqlite.org/copyright.html

SQLite has two different modes that can be used in your application. It can be run "in-memory" which means that the database you are creating is temporary – only available while the program is running – or you can create a physical database that will store the data "offline" when the program exits and be available the next time the program is started.   Let's create one of each and, though it is beyond the scope of this text, create a simple query from the database. We will use the code in the AutoIt help file for both examples.

In-memory database

If you navigate to _SQLite_Exec in the help file you will see the code below:

Code
Chapter 14 Example 6

```
#include <SQLite.au3>          ◄─── Required includes for SQLite to function
#include <SQLite.dll.au3>

Local $hQuery, $aRow                      ◄─── Declaration of variables
_SQLite_Startup()
ConsoleWrite("_SQLite_LibVersion=" & _SQLite_LibVersion() & @CRLF)
_SQLite_Open()
; Without $sCallback it's a resultless statement
_SQLite_Exec(-1, "Create table tblTest (a,b int,c single not null);" & _
        "Insert into tblTest values ('1',2,3);" & _
        "Insert into tblTest values (Null,5,6);")

Local $d = _SQLite_Exec(-1, "Select rowid,* From tblTest", "_cb") ; _cb will be called for each row

Func _cb($aRow)
    For $s In $aRow
        ConsoleWrite($s & @TAB)
    Next
    ConsoleWrite(@CRLF)
    ; Return $SQLITE_ABORT ; Would Abort the process and trigger an @error in _SQLite_Exec()
EndFunc   ;==>_cb
_SQLite_Close()
_SQLite_Shutdown()

; Output:
; 1    1   2   3
; 2        5   6
```

1.) This loads the SQLite.dll. A "DLL" is a Dynamic Link Library – a collection of functions that can be used by more than one program. It is a lot like a script that you include in your program except is compiled into a specific format and cannot be modified.

2.) This creates our database. It is worth noting that since we are not passing any parameters this will be an **in-memory** database. If we passed optional parameters for a filename, as we will do in the next example, it would be a physical database.

3.) The _SQLite_Exec function executes SQLite statements (that is database speak for runs them). In this case, the statement that is being executed is creating a table (a collection of rows and columns similar to what you may envision in a spreadsheet but they can contain different

datatypes and be queried). It is also conveniently inserting some data into the table that we can retrieve in the next step.

4.) The statement being executed in this line of code is selecting everything in the table by row id (the row number in the metaphorical spreadsheet we envisioned) . Then it is calling a custom function at the end called "_cb" which is an abbreviation for "callback".   The callback function is an optional parameter that allows you to call another function for each row. In this case the callback function is using a for loop to write the contents of the query to the console.

5.) This is the callback function referenced in #4 above.

   NOTE: the use of the for loop use on this line of code. It differs from the others that we covered in Chapter 6. It is a special kind of a for loop referred to as For … In … Next. It allows us to enumerate elements in an object collection or an array (from the AutoIt help file). That is a fancy way of saying do something for as many times as there are elements in an array or an object.

   NOTE: Objects in this context are data structures that we have not covered. For now all we need to understand is that they can hold collections of information similar to an array.

6.) This closes the open database.

7.) This unloads the SQLite.dll.

When you run the code in this example you will see the output that is commented out at the bottom of the screen appear in your console. The output contains all the information in our in-memory database.

```
rowid   a       b       c
1       1       2       3
2               5       6
```

Physical database

To create the same example but make the database a physical database that we can re-use we need to change the code in step 2.

```
_SQLite_Open(@ScriptDir&"\example.db")
```

We know the @ScriptDir macro from Chapter 13. Using it as part of this parameter will place a copy of the newly created database in the same directory as our script. Once we do that we can modify our example to read the physical database and output the results the exact same way:

```autoit
1    #include <SQLite.au3>
2    #include <SQLite.dll.au3>
3
4    Local $hQuery, $aRow
5    _SQLite_Startup()
6    ConsoleWrite("_SQLite_LibVersion=" & _SQLite_LibVersion() & @CRLF)
7    _SQLite_Open(@ScriptDir&"\example.db")
8    Local $d = _SQLite_Exec(-1, "Select rowid,* From tblTest", "_cb") ; _cb will be called for each row
9
10   Func _cb($aRow)
11       For $s In $aRow
12           ConsoleWrite($s & @TAB)
13       Next
14       ConsoleWrite(@CRLF)
15       ; Return $SQLITE_ABORT ; Would Abort the process and trigger an @error in _SQLite_Exec()
16   EndFunc    ;==>_cb
17   _SQLite_Close()
18   _SQLite_Shutdown()
19
20   ; Output:
21   ; 1      1    2    3
22   ; 2           5    6
23
```

Note this example removed the code that created and populated the table because we already save that table and it's data when we save a copy of the physical database

The two major differences in this example are: (1) we are opening a copy of the physical database that we created when we changed line 2 (we would have had to run that code for it to be there), and (2) we removed the creation of the table and data because they already exist when we saved our physical database. Now we are opening up that database, reading its contents, and displaying them just as we did for the in-memory version.

Chapter 15: Automating other applications

The help file introduces AutoIt as "… a freeware BASIC-like scripting language designed for automating the Windows GUI and general scripting." So far we have focused on the general scripting side of AutoIt because the information is very similar to other languages and provides a nice foundation for introductory programming techniques.   This chapter will explore the other main design tenant of AutoIt - the automation of other applications.

There are many ways to automate other applications with AutoIt.   We'll focus on two main categories: sending commands to existing windows and controls and use of application interfaces with existing UDFs.

Sending commands to existing windows and controls (targets) requires that we either identify the targets or that we send a series of keystrokes. Identifying the targets and interacting with them is a more reliable methodology.   Just know that some programs may be written with technologies that make it more difficult to identify their controls in a way that is conducive to automation. There are

some advanced techniques for those types of situations but our focus will be basic Windows form similar to the GUIs that we created in Chapters 9 & 10.[5]

Windows Management

AutoIt has many pre-built functions for interacting with windows. To get started we will need to open a couple of windows so that we can control them. Let's start with two that everyone should have: notepad and calculator. Locate those applications on your PC and start them up. You should see the following:

---
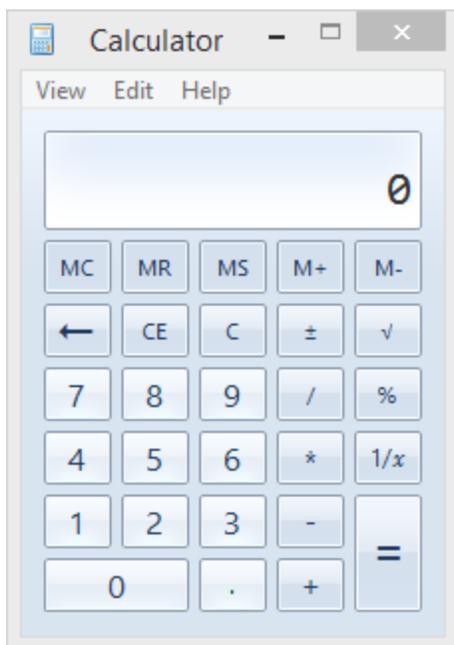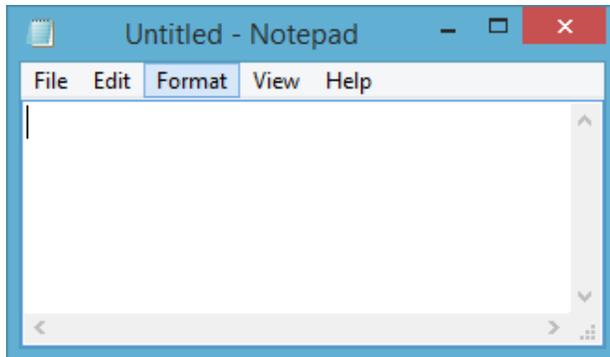
[5] One such advanced technique is the use of Microsoft's UIAutomation framework. There is an existing UDF for this that has been "pinned" in the AutoIt forum's example scripts – meaning it appears at the top in that area of the forum: http://www.autoitscript.com/forum/topic/153520-iuiautomation-ms-framework-automate-chrome-ff-ie/

Notice that each application has a title at the top of the window - "Untitled - Notepad" and "Calculator" respectively.   It is also important to understand that when you click on the window for notepad or calculator the window is "activated" and takes on a slightly different color.   An activated window will be the default recipient of all your keystrokes.   If you were to activate notepad with a mouse click and then start typing that text would appear within the notepad application. The same thing would happen for the calculator application except it would only accept certain values (i.e. numbers). Meanwhile, if you were to click somewhere else on your desktop the window that was activated would still be there – but it would be deactivated.

WinAcvitate

WinActivate

Activates (gives focus to) a window.

```
WinActivate ( "title" [, "text"] )
```

Parameters

title	The title/hWnd/class of the window to activate. See Title special definition.
text	[optional] The text of the window to activate. Default is an empty string. See Text special definition.

With our two programs running we can write a simple script to activate one window and then the other. The WinActivate function will do this for us and accepts the parameter "title/hWnd/class" to identify the window. The title is self-explanatory; we already know that this will either be "Untitled - Notepad" or "Calculator". The help file offers a detailed explanation of hWnd: "The variant datatype in AutoIt natively supports window handles (HWNDs). A window handle is a special value that Windows assigns to windows each time they are created. When you have a handle you may use it in place of the title parameter in any of the function calls that use the title/text convention. The advantage of using window handles is that if you have multiple copies of an application open which have the same title/text then you can uniquely identify them using handles."[6] Class refers to the "… internal window classname". Any one of those three values will allow us to activate the windows.

---

[6] From the Window Titles and Text (Advanced) section of the AutoIt help file.

Code Chapter 15 Example 1

Sleep

```
1    $notepad=WinActivate("Untitled - Notepad")
2    sleep(2000)
3    $calculator=WinActivate("Calculator")
```

If you run the above code you will activate the notepad window, then there will be a two second delay, and then the calculator window will activate. Activating the windows with WinActivate is pretty well understood but why and how did we create the two second delay? We used a delay because If we did not we might not be able to detect the change from activating notepad and then the calculator because the program would execute in the instructions very quickly. Therefore, we used a function called "sleep" that accepts milliseconds (thousandths of a second) as a parameter. By using 2000 milliseconds we created a two second delay.

NOTE: On success the function will return a handle for the activated window. The handle will be stored in the variables that we created called $notepad and $calculator respectively. We can use those handles for additional function calls to further interact with the windows.

WinClose

If we want to close a window we can use the WinClose function. We can modify our code above to add an additional line that will close notepad after calculator is activated. However, this time, we can use notepad's handle that we captured from its activation.

Code Chapter 15 Example 2

```
1    $notepad=WinActivate("Untitled - Notepad")
2    sleep(2000)
3    $calculator=WinActivate("Calculator")
4
5    WinClose($notepad)    ⬅
```

Now our program will activate notepad, pause, activate calculator, then close notepad using its handle.

WinWaitActivate

You will invariably want to do more than simply activate a window. The activation is what sets you up to send instructions to the window. Sometimes you may incur a small delay while your window is opening For example – a program that says "loading" or one that takes a few seconds to fully appear before you can start to use it. What would happen if you started to send instructions to that window before it was ready? It would not be able to receive all the instructions. Therefore, we may want to use an activation that pauses the rest of our script until the targeted window becomes active. That is exactly what WinWaitActivate does.

WinWaitActive

Pauses execution of the script until the requested window is active.

```
WinWaitActive ( "title" [, "text" [, timeout = 0]] )
```

Parameters

title	The title/hWnd/class of the window to check. See Title special definition.
text	[optional] The text of the window to check. Default is an empty string. See Text special definition.
timeout	[optional] Timeout in seconds if the window is not active. Default is 0 (no timeout).

You may notice that the parameters of WinWaitActivate are very similar to those of WinActivate except there is an additional optional parameter for timeout. What would happen if you paused your script to wait for a window to become activated but it never did (maybe it was not open and you forgot to open it)? Your script would "hang" (become paused indefinitely). Therefore, it is a good idea to use a timeout parameter that will abandon the paused state after so many seconds if the window has not become activated.

NOTE: The timeout parameter is stated in seconds for this function. Some functions (most) use seconds while others may use milliseconds (i.e. sleep). Always be careful to check the unit of time measure for these functions.

NOTE: Autoit has a function to set certain options aptly named "AutoItSetOption". One such option is to set the windows title matching mode as follows:

Opt("WinTitleMatchMode", 1) ;1=start, 2=subStr, 3=exact, 4=advanced, -1 to -4=Nocase

This option is handy when differentiating between situations where you know the exact title, a portion of the title, and whether or not it is case sensitive, etc.

Send

The send function sends simulated keystrokes to the active window. Let's re-open notepad and alter our code a bit to send some text to it.

Code Chapter 15 Example 3

```
1    $notepad=WinActivate("Untitled - Notepad")
2    WinWaitActive($notepad,"",3)
3    send("Learning about automation is fun")
4
```

Running the above code will activate an existing notepad window (i.e. one that we have already opened), wait until it is active, and then send the text "Learning about automation is fun". The sent text will appear in notepad after the program is run because that is the active window.
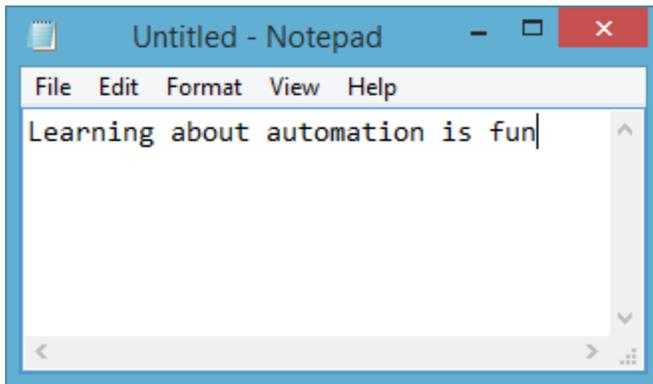
What would happen if for some reason we did not wait for that window to become active or if the timeout delay ended and the window was still not active (perhaps because the program was not running?). If we use the timeout parameter the script would continue and send the simulated keystrokes to the active window – whatever that may be – which could cause some strange things to happen. Therefore, you need to careful when using send.

NOTE: we also could have started notepad from our code instead of opening it first by hand and then activating the windows.   To do that we could use the Run function.

Code Chapter 15 Example 4

Run

```
1    Run("notepad.exe")
2    WinWaitActive("Untitled - Notepad")
3    Send("Learning about automation is fun")
```

Run is used to run external programs. In this case we could use it to run notepad with the above code, wait for the window to become active (we are more likely to incur a delay when we run it from scratch then if it were already open) and send our keystrokes.

Send supports many different combinations to simulate different types of keystrokes. For example, notepad contains a shortcut to the file menu in the form of pressing ALT + F. The exclamation point "!" simulates the ALT key so sending "!F" would send ALT+F. Let's try it.

Code Chapter 15 Example 5

```
1    Run("notepad.exe")
2    WinWaitActive("Untitled - Notepad")
3    Send("!F")
```

Running the above code ran an instance of notepad, waited for the window to become active, and then sent ALT+F to the activated window. This resulted in the activation of notepad's file menu.
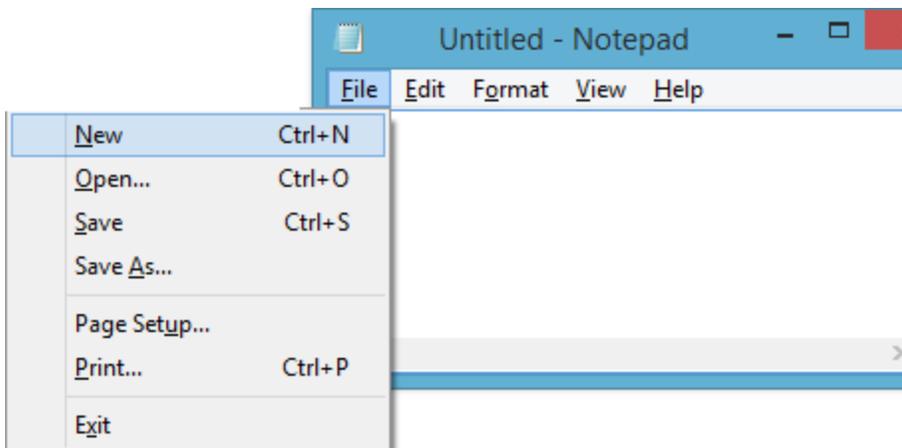
You can refer to the help file for a complete listing of all send key combinations and approaches.

Updating controls

At this point we can manipulate windows and send simulated keystrokes to the active window. We can also use keys in order to open a menu item that has a shortcut before sending additional simulated keystrokes. That is all good stuff but what if we wanted to update a specific input control? If we created the control from within our application then it would be easy – we could use the control handle returned to us when we created the control with the GUIControlSetData (not previously discussed but documented in the help file) function to update it. What if we wanted to update the calculator screen and buttons with some text? Since we did not create the control we have to find its ControlID some other way. To do this we will use a tool that ship with AutoIt called "Au3Info". First, run the calculator program. Next, go to SciTE and open a new or existing au3 file (if it is new you will need to save it with an au3 extension). Finally, go to the tools menu and click on Au3Info.

Compile	Ctrl+F7
Build	F7
Go	F5
SyntaxCheck Prod	Ctrl+F5
AU3Info	Ctrl+F6
AU3Recorder	Alt+F6
Tidy AutoIt Source	Ctrl+T
CodeWizard	Alt+W
Koda(FormDesigner)	Alt+m
SciTe Config	Ctrl+1
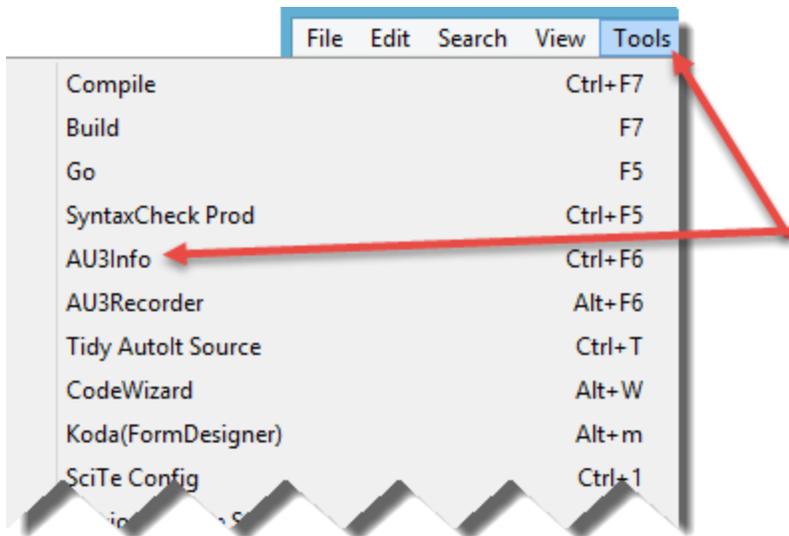
If you successfully launched the tool you should see the following.

Au3Info
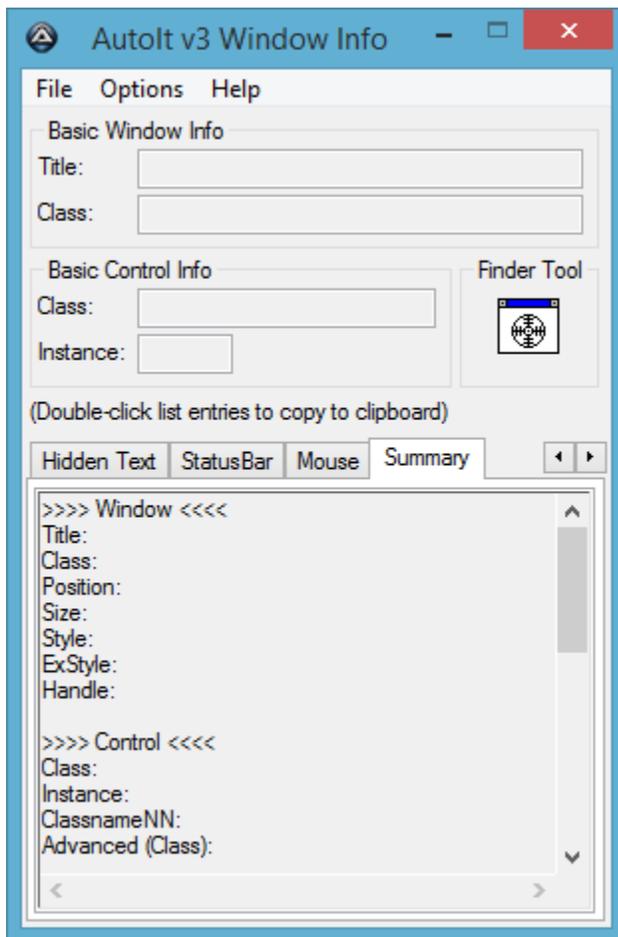
The finder tool is a drag-and-drop crosshair that can be used to get more information about controls in other applications.[7] If we drag the cross hair into the window of our calculator application where the numbers appear we will get some information about the control. If we were to do that we would see that the control has a very specific class and instance of the class.
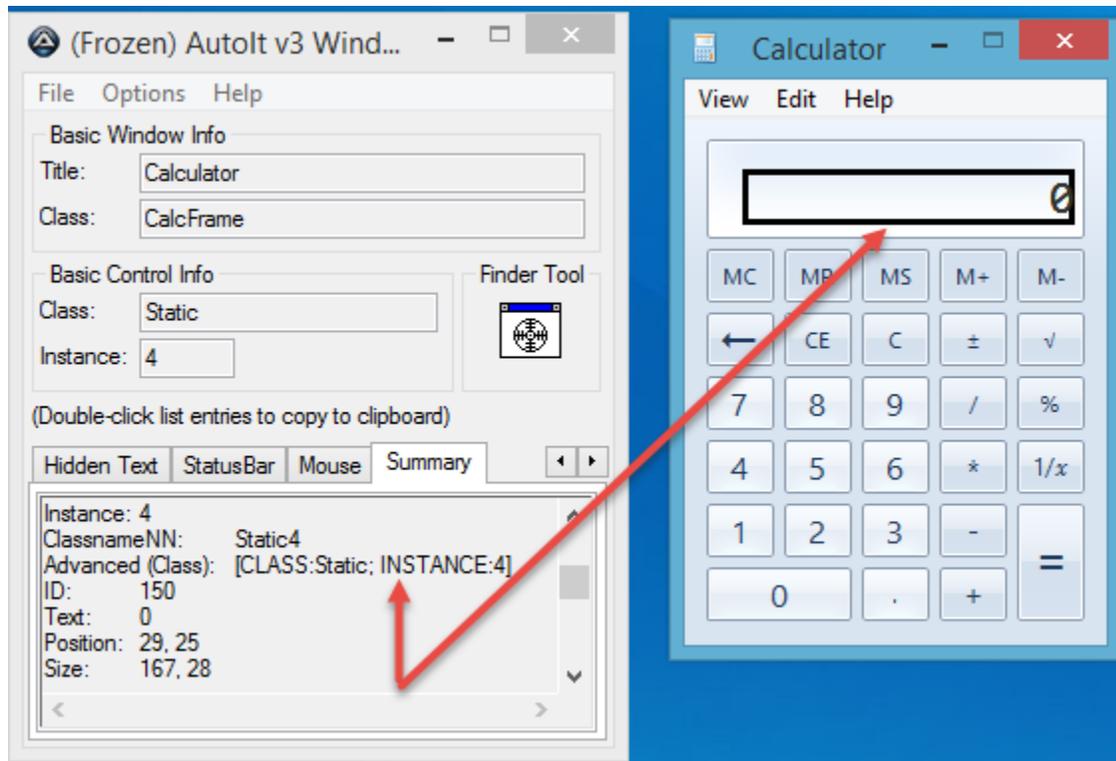
We can repeat that same exercise and discover the names of all the numbered buttons. We might also discover that there is a small input area above the number display that we can read. Now comes the cool part … remember when we were sending key strokes? That process allowed us to interact with the calculator program in a way that was consistent with its contemplated design. This meant that we could not send text to the numbers area – it would only accept numeric values as an activated window. However, now that we know the IDs of the individual controls we have more flexibility to send text to them using ControlSetText. We could make a small program that would do something like this:

---

[7] Note: this may not always work. Some applications may be written with technologies that require another approach. See footnote 5 re: MS UIAutomation.
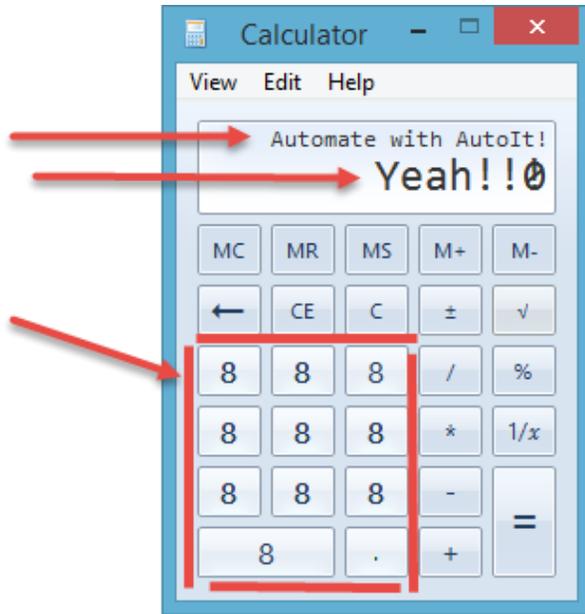
We will go over the code in just a moment. However, notice that we were able to update the pane above the numbers panel and then also send text to the area traditionally reserved for numbers only. This is especially handy when you consider the user interface (the calculator itself) has no way to enter non numeric values. As an added goof we changed all the numeric buttons to the number 8 from their original values of 1-9 and 0. Let's review the code:

Code

Chapter 15 Example 6 (with final mods for the "A" change discussed in a moment)

```
; First we opened an instance of calculator
run("calc.exe")
$calc=WinWaitActive("Calculator") ; activates the window

; Panes
$controlTopPane = "[CLASS:Static; INSTANCE:2]" ; pane above numbers
$controlNumberPane = "[CLASS:Static; INSTANCE:4]"; numbers pane
; update panes with our text
ControlSetText("Calculator","",$controlTopPane,"Automate with AutoIt!")
ControlSetText("Calculator","",$controlNumberPane,"Yeah!!!")

;Buttons
$controlOneBtn = "[CLASS:Button; INSTANCE:5]"; one button
$controlTwoBtn = "[CLASS:Button; INSTANCE:11]"; two button
$controlThreeBtn = "[CLASS:Button; INSTANCE:16]"; three button
$controlFourBtn = "[CLASS:Button; INSTANCE:4]"; four button
$controlFiveBtn = "[CLASS:Button; INSTANCE:10]"; five button
$controlSixBtn = "[CLASS:Button; INSTANCE:15]"; six button
$controlSevenBtn = "[CLASS:Button; INSTANCE:3]"; seven button
$controlEightBtn = "[CLASS:Button; INSTANCE:9]"; eight button
$controlNineBtn = "[CLASS:Button; INSTANCE:14]"; nine button
$controlZeroBtn = "[CLASS:Button; INSTANCE:6]"; nine button

; an array of the buttons that we can loop through to update them all
dim $btnArray[10]=[$controlOneBtn,$controlTwoBtn,$controlThreeBtn,$controlFour

$text="A" ; used to update all buttons
; The loop that sets all the buttons to "A"
for $a=0 to ubound($btnArray)-1
ControlSetText("Calculator","",$btnArray[$a],$text)
next
```

1.) We ran the calculator program and waited for the window to be active.
2.) We used the values obtained from Au3Info about the main numbers pane and the invisible pane above it (where we wrote "Automate with AutoIt!) which we assigned to variables.
3.) We set the text of the two panes to "Yeah!!!" and "Automate with AutoIt!" respectively.
4.) We used the Au3Info tool to get the class names and instances for all the numeric buttons on the calculator and assigned them to variables.
5.) We created an array that stores all 10 button variables – it is cut off in the above picture. The full text looks like this:
$btnArray[10]=[$controlOneBtn,$controlTwoBtn,$controlThreeBtn,$controlFourBtn,$controlFiveBtn,$controlSixBtn,$controlSevenBtn,$controlEightBtn,$controlNineBtn,$controlZeroBtn]
6.) We created a variable that holds the text we used to replace all the previous text on the buttons (i.e. 1-9 and 0). This way we could change the buttons to any other value if we re-run it.
7.) We used a for loop to loop through the array containing the buttons and change their text to the text stored in our variable. The array is a zero based array meaning the first value is stored in

the "0" index. Therefore, our loop will run from "0" to the upper bound of the array (which is 10 because there are 10 buttons) less one. That is because, as we learned earlier though there are ten pieces of information the last one is stored at position nine (0-9 = 10 items). Finally we used ControlSetText to set text to the control. This is an example of using arrays in our code to save a lot of time. The alternative could have been to write 10 lines of code each of which would have had to have been customized. Instead, we were able to loop through the array and use variables so that if wanted to change them to anything else in the future it would be easy. For example – if we change the "8" in our $text variable to "A" it would look like this:
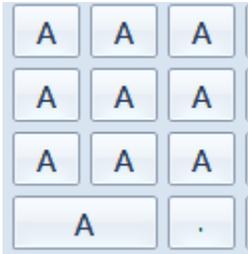
ControlSetText

ControlSetText

Sets text of a control.

```
ControlSetText ( "title", "text", controlID, "new text" [,
flag = 0] )
```

Parameters

title	The title/hWnd/class of the window to access. See Title special definition.
text	The text of the window to access. See Text special definition.
controlID	The control to interact with. See Controls.
new text	The new text to be set into the control.
flag	[optional] when different from 0 (default) will force the target window to be redrawn.

ControlSetText requires the title of the window that we want to access (in this case "Calculator") and the text of the window (which we left as blank). The controlID is the class that we found using Au3Info which we are storing in our variables. Finally, the last parameter is the text we want to set on the control (in our case this came from our $text variable).

 NOTE: even though we changed the text on the controls we did not impact their underlying functions. Therefore, the 1 -9 and 0 buttons would all still produce their true values on the screen. All we did was change the text on the buttons.

ControlCommand

Sometimes setting the text of a control is enough. You may want to send it commands. We have already seen that you can simulate this with send. You can also use ControlCommand to send commands to certain controls. Let's add the following to the end of our calculator application:

 Chapter 15 Example 7

```
sleep (2000) ; small delay to view original results
send("!2") ; invoke scientific calc mode ALT + 2 shortcut
$radiansRadio="[CLASS:Button; INSTANCE:30]" ; the radians radio button Au3Info
WinWaitActive("Calculator"); wait for new mode to become active
ControlCommand("Calculator","",$radiansRadio,"check"); select radian radio
```

The comments in the above code explain what we are doing. First, because this code is placed at the end of our existing program that changed the numbers on the calculator we are adding a two second delay with sleep so that we can still observe our updated controls. Then we are using send to invoke the shortcut that switches the calculator into scientific mode. That may take a second or two so we wait for that window to become active again before we send a command to the scientific calculator to switch to radians. This last step uses ControlCommand. The first three parameters are similar to those found in ControlSetText. However, the last parameter is from a list of possible commands found in the help file.

The final result will look like this:
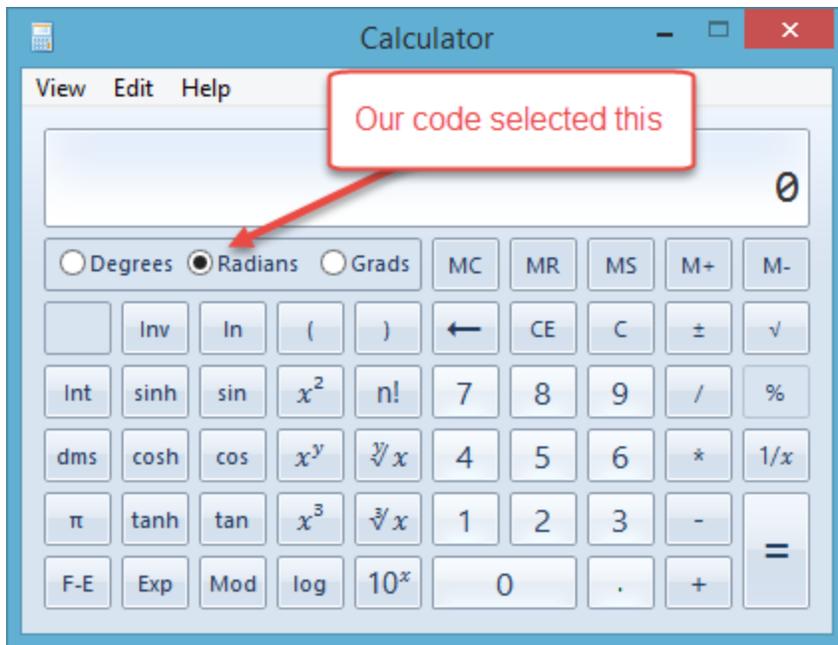
ControlSend

This function works similar to Send but instead of sending the keystrokes to the active window it sends the directly to a specific window / control. Therefore, unlike ControlSetText we can't use it to do things that were not intended by the original program. That means we can only send keystrokes to the calculator that it would understand. To do this we could use the following code:

```
ControlSend ( "Calculator", "", $controlNumberPane, "99999"); enters 99999 into the num pane
```

The above code would make the number pane on the initial calculator screen show "99999". What would happen if we tried to send "Hello World to that control? It may not understand what we are sending and it could throw an error:
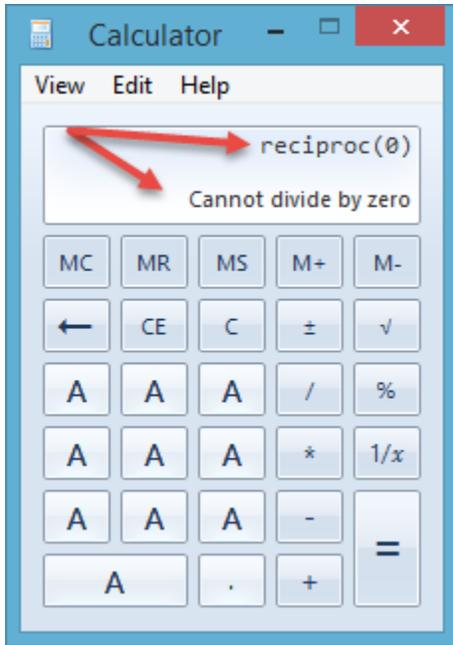
NOTE: Compare to ControlSetText where we forced the text on the control instead of sending simulated keystrokes to it.

Mouse Management

We have already covered sending opening programs, simulating keystrokes, and even interacting with windows and controls directly. However, sometimes you may have a need to simulate mouse movements and clicks.   The help file has an entire section on this topic and lists several useful functions. We will cover MouseGetPos and MouseMove as running those two functions will be benign whereas some of the other functions would simulate mouse clicks somewhere on your screen and we all have different screens so you can explore those on your own.

Function	Description
MouseClick	Perform a mouse click operation.
MouseClickDrag	Perform a mouse click and drag operation.
MouseDown	Perform a mouse down event at the current mouse position.
MouseGetCursor	Returns the cursor ID Number for the current Mouse Cursor.
MouseGetPos	Retrieves the current position of the mouse cursor.
MouseMove	Moves the mouse pointer.
MouseUp	Perform a mouse up event at the current mouse position.
MouseWheel	Moves the mouse wheel up or down.

MouseGetPos

MouseGetPos will return the X (horizontal), Y (vertical) coordinates of your mouse on your screen. The results will be stored in an array with the $array[0] position representing x and the $array[1] position representing y.

MouseMove

MouseMove will move your move your mouse to a given x,y coordinate at an optionally supplied speed.

Code Chapter 15 Example 8

```
Sleep (2000)
$mousePos=MouseGetPos()
$mouseX=$mousePos[0]
$mouseY=$mousePos[1]
mousemove($mouseX+100,$mouseY+100)
```

The above code determines the current position of the mouse and returns the X, Y coordinates in an array. Then it moves the mouse 100 pixels to the right and 100 pixels down. Notice the Sleep(2000) at the start. That is a two second delay in case you want to position your mouse after you start the script.

External application UDFs

Another useful way to work with external applications is through User Defined Functions (UDFs). We already saw this when we covered UDFs in Chapter 14. Do you recall the SQLite UDF that allowed us to create in-memory and physical databases with SQLite? That was an example of using a UDF with an external application. AutoIt ships with several UDFs that allow you to manipulate other applications such as Excel and Word. There are also many more UDFs that automate external applications which have been posted on the forum by various users.

But how do they work? Most of these UDFs are using an application programming interface (i.e. specific functions that are allowed to interact with other programs). The beauty of the UDFs is that they hide much of this complexity. Though it would be useful, you don't need to understand the functions at the code level to use them. You can simply include the functions in your script and enjoy the benefits.

Excel UDF

NOTE: You will need to have Microsoft Excel installed on your PC to test these scripts.

The Excel UDF allows you to open up a new or existing Excel workbook and then automate a host of useful functions.

The first thing we need to do is open a new instance of Excel to obtain a reference to the Excel object that we intend to automate.

_Excel_Open

Connects to an existing Excel instance or creates a new one

```
#include <Excel.au3>
_Excel_Open ( [$bVisible = True [, $bDisplayAlerts = False [, $bScreenUpdating = True [,
$bInteractive = True [, $bForceNew = False]]]]] )
```

Parameters

$bVisible	[optional] True specifies that the application will be visible (default = True)
$bDisplayAlerts	[optional] False suppresses all prompts and alert messages while opening a workbook (default = False)
$bScreenUpdating	[optional] False suppresses screen updating to speed up your script (default = True)
$bInteractive	[optional] If False, Excel blocks all keyboard and mouse input by the user (except input to dialog boxes) (default = True)
$bForceNew	[optional] True forces to create a new Excel instance even if there is already a running instance (default = False)

This function allows us to create a new instance of Excel or connect to a new one. All of its parameters are optional and control the visibility, alerts, etc. To create a new Excel instance all we have to do is type the following:

Code Chapter 15 Example 9

```
1    $oExcel = _Excel_Open()
```

Upon successful completion the function will return an Excel application object which we will store in the $oExcel variable. Next, we need to create a new instance of an Excel workbook. For that we will use _Excel_BookNew.

_Excel_BookNew

Creates a new workbook

```
#include <Excel.au3>
_Excel_BookNew ( $oExcel [, $iSheets = Default] )
```

Parameters

$oExcel	Excel application object where you want to create the new workbook
$iSheets	**[optional]** Number of sheets to create in the new workbook (default = keyword Default = Excel default value). Maximum is 255

Let's use a couple of lines of code to create an Excel object and open a workbook:

Code Chapter 15 Example 10
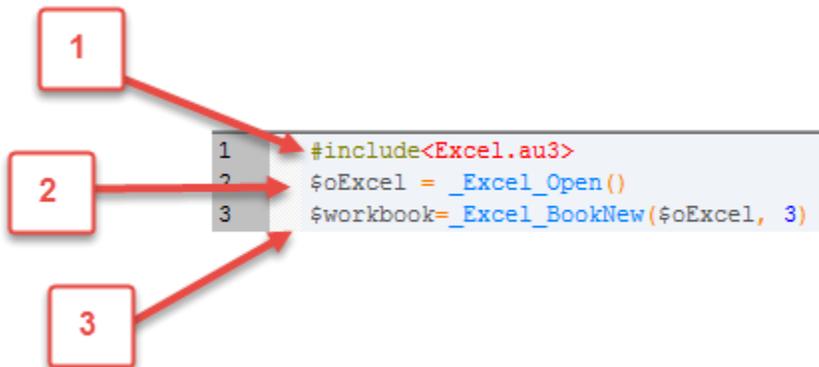
1. This is a reference to the Excel UDF that must be included in our script in order for us to have access to its functions.
2. This creates the Excel application object.
3. This creates an Excel workbook and optionally specifies that it should have three sheets.

If you run this code on a machine that has Excel installed you should see the application pop open with new workbook containing three sheets.
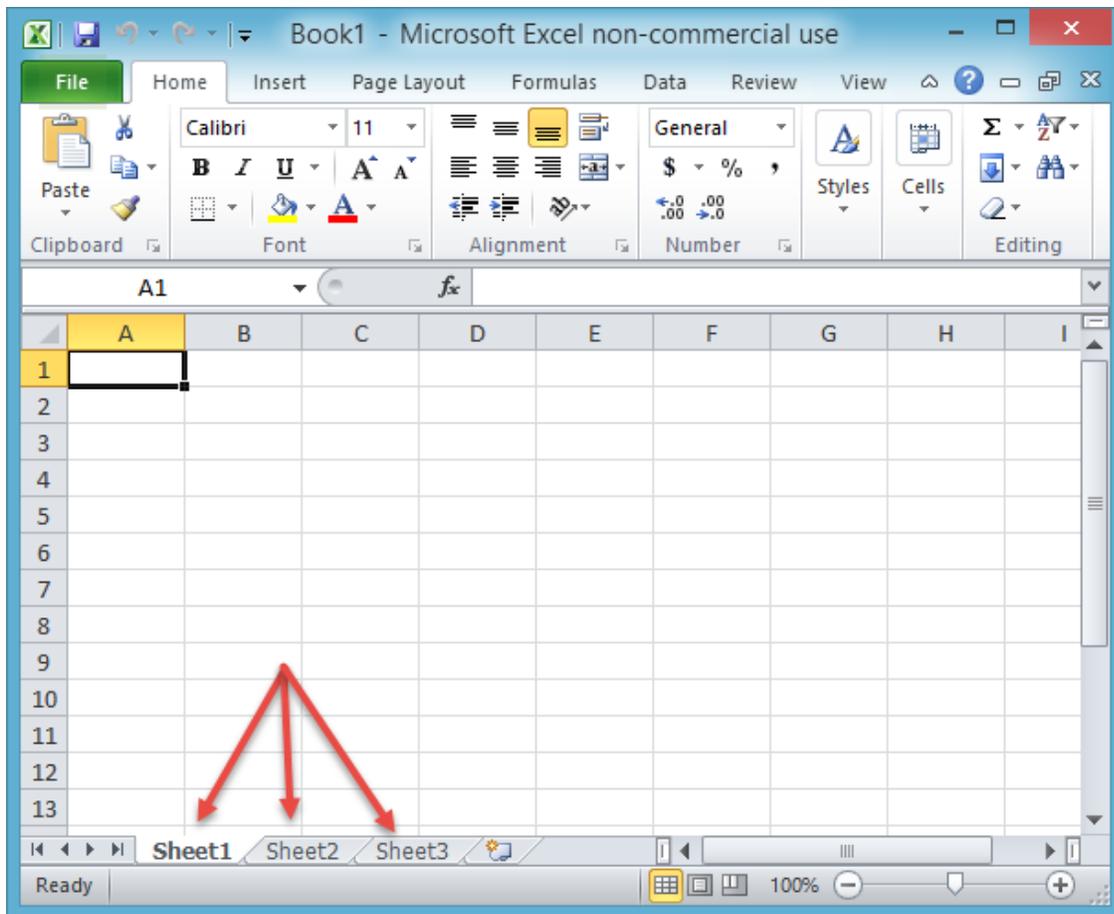
Opening a new copy of Excel and a blank workbook is a pretty good trick but we will need to do more than that if our program is to be useful. The ability to read and write to and from a spreadsheet will open up a lot of automation opportunities especially if were to combine those operations with some of the concepts from prior chapters such as loops and string management.

NOTE: by using arrays, loops, and string management along with Excel automation we can create powerful data cleansing / transformation programs.

Let's add some data to our blank workbook to demonstrate some additional concepts of the Excel UDF. You may remember our $people array from Chapter 8. We have re-created it in the code snippet below. After we create the array we use _Excel_RangeWrite to write the data to the workbook. If you run the code snippet below you should see the data added to the workbook.

Code Chapter 15 Example 11

```autoit
1  #include<Excel.au3>
2  #include<Array.au3>
3  $oExcel = _Excel_Open()
4  $workbook=_Excel_BookNew($oExcel, 3)
5
6  global $people[3][2] = [["John",34],["Steve",23],["Bob",41]]
7  _Excel_RangeWrite($workbook,1,$people)
```
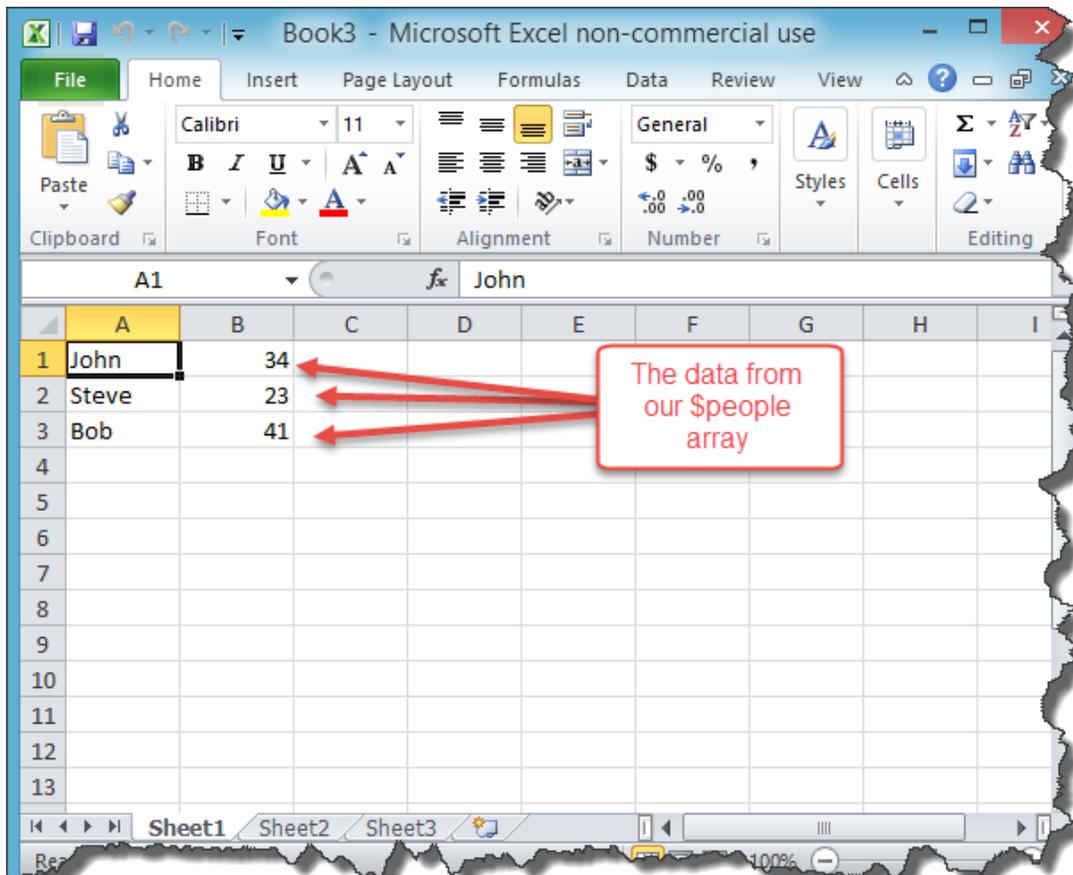

Let's take a closer look at the _Excel_RangeWrite function:

_Excel_RangeWrite

Writes value(s) or formula(s) to a cell or a cell range on the specified workbook and worksheet

```
#include <Excel.au3>
_Excel_RangeWrite ( $oWorkbook, $vWorksheet, $vValue [, $vRange = "A1" [, $bValue = True
[, $bForceFunc = False]]] )
```

Parameters

$oWorkbook	Excel workbook object
$vWorksheet	Name, index or worksheet object to be written to. If set to keyword Default the active sheet will be used
$vValue	Can be a string, a 1D or 2D zero based array containing the data to be written to the worksheet
$vRange	**[optional]** Either an A1 range or a range object (default = "A1")
$bValue	**[optional]** If True the $vValue will be written to the value property. If False $vValue will be written to the formula property (default = True)
$bForceFunc	**[optional]** True forces to use the _ArrayTranspose function instead of the Excel transpose method (default = False). See the Remarks section for details.

The function can be used to write values or formula to a specific cell or to a range of cells on a specified worksheet within a workbook. In our case, we wrote an array to worksheet 1. The code looked like this:

_Excel_RangeWrite($workbook,1,$people)

That code supplies the first three mandatory parameters: the workbook object, which sheet to write to, and the value to be written. Note that the second parameter could be the name of the sheet (as in "Sheet1", an index (which is what we used when we supplied 1 to as a reference to the first sheet, or a worksheet object. The value could be string or an array. In our case it was an array containing names and ages.

Did you notice that our array started in the first row and column of the worksheet (i.e. A1)? What if we wanted it to start on B2 instead? Adding the optional fourth parameter for range could help us do exactly that:

```
_Excel_RangeWrite($workbook,1,$people,"B2")
```

Now, having specified we want to start at B2 our results would look like this:

Now the data is written to B2 as a starting point:
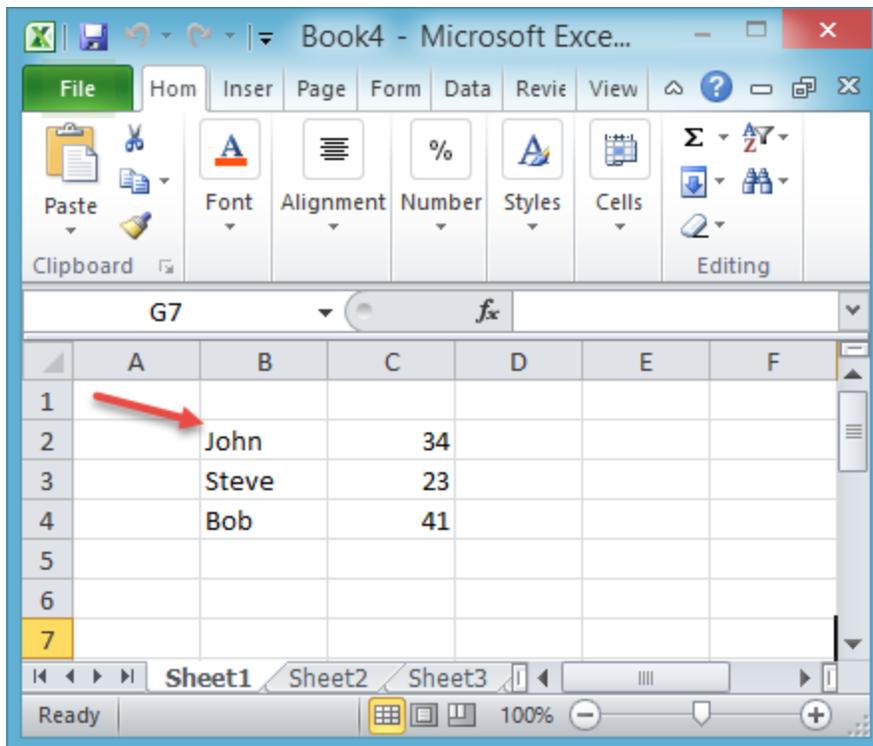
After we add some data to our workbook we may want to save it under a different name. To do that we can use the _Excel_BookSaveAs function.

_Excel_BookSaveAs

Saves the specified workbook with a new filename and/or type

```
#include <Excel.au3>
_Excel_BookSaveAs ( $oWorkbook, $sFilePath [, $iFormat = $xlWorkbookDefault [,
$bOverWrite = False [, $sPassword = Default [, $sWritePassword = Default [,
$bReadOnlyRecommended = False]]]]] )
```

Parameters

$oWorkbook	Workbook object to be saved
$sFilePath	Path and filename of the file to be read
$iFormat	[optional] Excel writeable filetype. Can be any value of the XlFileFormat enumeration.
$bOverWrite	[optional] True overwrites an already existing file (default = False)
$sPassword	[optional] The string password to protect the sheet with. If set to keyword Default no password will be used (default = keyword Default)
$sWritePassword	[optional] The string write-access password to protect the sheet with. If set to keyword Default no password will be used (default = keyword Default)
$bReadOnlyRecommended	[optional] True displays a message when the file is opened, recommending that the file be opened as read-only (default = False)

Using this function we can save our workbook to our script directory and call it names.xlsx with the following code:

Code Chapter 15 Example 12

```
_Excel_BookSaveAs($workbook,@ScriptDir&"\names.xlsx")
```

In the above code we have supplied a reference to the workbook and the path, including the filename, used to save a copy of the workbook. We could have optionally supplied dome parameters to change the file type of our saved workbook, specify whether or not it can be overridden, and password protect it.

So far we have created a brand new workbook, added data to it, and saved the workbook down with a new name. You may also have need to open an existing workbook that already contains data. For this, you can use _Excel_BookOpen.

_Excel_BookOpen

Opens an existing workbook

```
#include <Excel.au3>
_Excel_BookOpen ( $oExcel, $sFilePath [, $bReadOnly = False [, $bVisible = True [,
$sPassword = Default [, $sWritePassword = Default]]]] )
```

Parameters

$oExcel	Excel application object where you want to open the workbook
$sFilePath	Path and filename of the file to be opened
$bReadOnly	[optional] True opens the workbook as read-only (default = False)
$bVisible	[optional] True specifies that the workbook window will be visible (default = True)
$sPassword	[optional] The password that was used to read-protect the workbook, if any (default is none)
$sWritePassword	[optional] The password that was used to write-protect the workbook, if any (default is none)

_Excel_BookOpen will need a reference to the application object and the path and filename to the workbook that to be opened.

Code Chapter 15 Example 13

```
1   #include <MsgBoxConstants.au3>
2   #include<Excel.au3>
3   #include<Array.au3>
4   $oExcel = _Excel_Open()
5   $workbook=_Excel_BookNew($oExcel, 3)
6
7   global $people[3][2] = [["John",34],["Steve",23],["Bob",41]]
8   _Excel_RangeWrite($workbook,1,$people)
9
10  _Excel_BookSaveAs($workbook,@ScriptDir&"\names.xlsx",default,True)
11  _Excel_BookClose($workbook)
12  MsgBox($MB_OK,"","Workbook closed click okay to reopen")
13  _Excel_BookOpen($oExcel,@ScriptDir&"\names.xlsx")
```
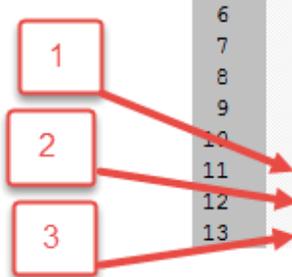
In the above example we have created a new workbook and array which we used to add data to the first sheet in the workbook. Then we saved the workbook under the name: "names.xlsx" because the sheet contained a bunch of names. From there we did the following:

1.   This line closes the workbook that we had saved with our new name.
2.   This line creates a message box that must be manually closed before the code will continue.

3. This line reopens the workbook that we had saved using _Excel_BookOpen.

The final task that we should look at to complete the circle is to read data from the workbook that we just opened (ignoring for a moment that we are the ones who created the workbook in the first place with data from our program). To do that, we need to use _Excel_RangeRead.

_Excel_RangeRead

_Excel_RangeRead, as the name would suggest, will allow us to read the contents of a specified range of cells within an Excel workbook.

_Excel_RangeRead

Reads the value, formula or displayed text from a cell or range of cells of the specified workbook and worksheet

```
#include <Excel.au3>
_Excel_RangeRead ( $oWorkbook [, $vWorksheet = Default [, $vRange = Default [, $iReturn =
1 [, $bForceFunc = False]]]] )
```

Parameters

$oWorkbook	Excel workbook object
$vWorksheet	[optional] Name, index or worksheet object to be read. If set to keyword Default the active sheet will be used (default = keyword Default)
$vRange	[optional] Either a range object or an A1 range. If set to Default all used cells will be processed (default = keyword Default)
$iReturn	[optional] What to return from the specified cell: 　1 - Value (default) 　2 - Formula 　3 - The displayed text
$bForceFunc	[optional] True forces to use the _ArrayTranspose function instead of the Excel transpose method (default = False). 　See the Remarks section for details.

The function has several parameters but only a reference to the workbook is required. The default behavior will be to read everything and display the values in an array unless you specify a specific range and/or that you would like to read formulae or text instead. A couple of more lines added to our example will demonstrate the function:

Code Chapter 15 Example 14

```
1   #include <MsgBoxConstants.au3>
2   #include<Excel.au3>
3   #include<Array.au3>
4   $oExcel = _Excel_Open()
5   $workbook=_Excel_BookNew($oExcel, 3)
6
7   global $people[3][2] = [["John",34],["Steve",23],["Bob",41]]
8   _Excel_RangeWrite($workbook,1,$people)
9
10  _Excel_BookSaveAs($workbook,@ScriptDir&"\names.xlsx",default,True)
11  _Excel_BookClose($workbook)
12  MsgBox($MB_OK,"","Workbook closed click okay to reopen")
13  $newworkbook=_Excel_BookOpen($oExcel,@ScriptDir&"\names.xlsx")
14  sleep(2000)
15  $data=_Excel_RangeRead($newworkbook)
16  _ArrayDisplay($data)
```

1

2

In the above example, after we opened our saved workbook, we created a message box that prompted us to re-open it. Then we used the "sleep" function to pause the script for two seconds (or 2000 milliseconds). After that, we read all the data the spreadsheet contained into an array. Then we displayed that array. If run to completion we would see a listview with the array data that looks like this:
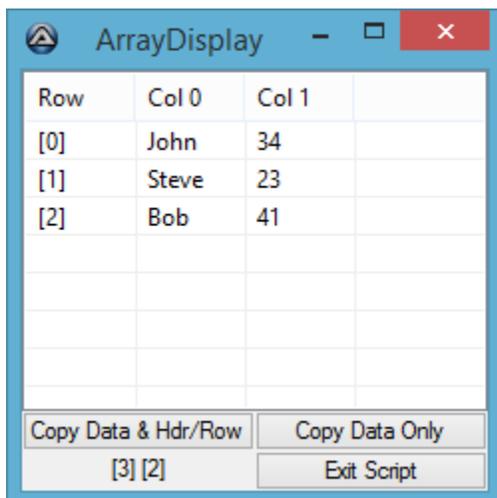
ArrayDisplay		
Row	Col 0	Col 1
[0]	John	34
[1]	Steve	23
[2]	Bob	41
Copy Data & Hdr/Row		Copy Data Only
[3] [2]		Exit Script

Your functions

Once you have an object identifier you can even use it to create your own functions. However, you will need to look at the program's API documentation - that complexity that was so nicely hidden with the UDF. You can use the same object identified returned with the UDF functions with your own functions. Here is an example of a script that inserts an image into an Excel workbook:

Code Chapter 15 Example 15

```
1    #include<Excel.au3>
2    $oExcelApp=_Excel_Open()
3    $oExcel = _Excel_BookNew($oExcelApp)
4    $path = @ScriptDir&"\au3logo.png"
5    $targetRange=$oExcel.ActiveSheet.Range("A1:D10")
6
7    func _InsertPictureInRange($path, $targetRange)
8    ; inserts a picture and resizes it to fit the TargetCells range
9    Dim $p, $t, $l, $w, $h
10       ; import picture
11       $p = $oExcel.ActiveSheet.Pictures.Insert($path)
12       ; determine positions
13       With $targetRange
14           $t = .Top
15           $l = .Left
16           $w = .Offset(0, .Columns.Count).Left - .Left
17           $h = .Offset(.Rows.Count, 0).Top - .Top
18       EndWith
19       ; position picture
20       With $p
21           .Top = $t
22           .Left = $l
23           .Width = $w
24           .Height = $h
25       EndWith
26       $p = "Nothing"
27    EndFunc
28    _InsertPictureInRange($path, $targetRange)
29
```

In the above code a new workbook with its own object identified was created. Then a variable was used to store the location of the image that we wanted to paste into the excel workbook. Next, we created a target range that would be used as the area in the worksheet where we want to insert our picture. We then left the nice tidy world of prepackaged Excel UDF functions and used a method that we found on the Microsoft website as part of the Excel COM API (the application programming interface we are using to connect to and control Excel) that accepted the path parameter to our picture and the range where we want to paste the picture as arguments. The remaining portion of the function calls the insert picture interface and manages the position of the image.

The result looks like this:
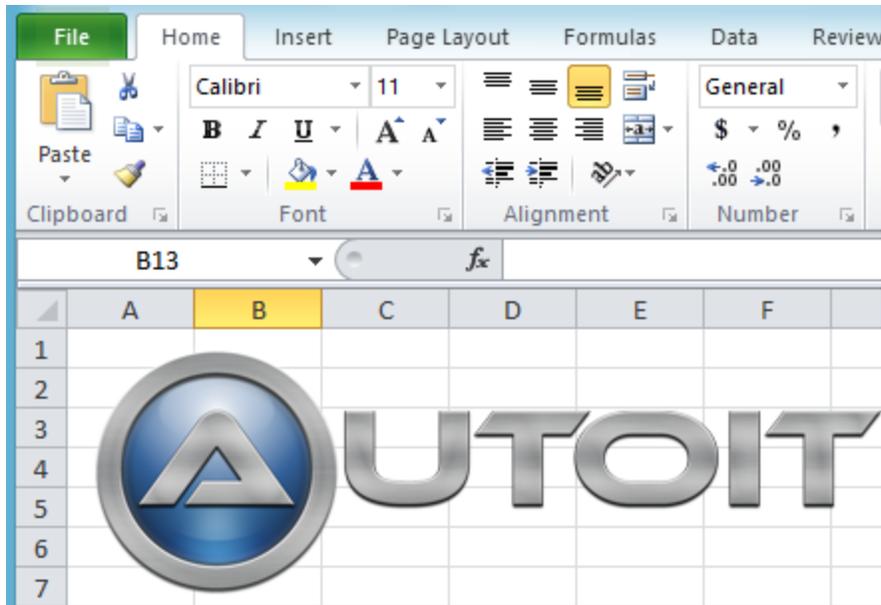
NOTE: Custom functions to automate other applications through an application programming interface (API) are an advanced topic. You will also need to study examples in the API documentation and convert the code provided to AutoIt code. The Excel COM API documentation can be found here: http://msdn.microsoft.com/en-us/library/office/ff194068%28v=office.15%29.aspx.

Chapter 16 Compiling: Making your programs into executables

To this point we have created many scripts covering a wide variety of functions. Let's say that we created a great script that we wanted to share with our friends so they could run it on their computers. How would we go about doing that? They would either have to have AutoIt installed on their machines or we would have to compile our script into an executable (a stand-alone program). That process transforms the code from human readable to machine readable instructions. It is what will allow others to run your program even if they have never heard of AutoIt. There are several different ways to do this.

First method:

AutoIt comes with a program called Aut2Exe that can be found from your start menu or within the folder where you installed AutoIt. The program allows you to select a script to compile, a location (path) to place the compiled program, and an icon to represent the program (if you don't have an icon or don't want to make one there is a default AutoIt icon). There is also an option to compile for x64. That option is used for newer Windows operating systems that are 64bits versions. However, most of the time you can compile to 32bits and it will run on both. The Aut2Exe program looks like this:
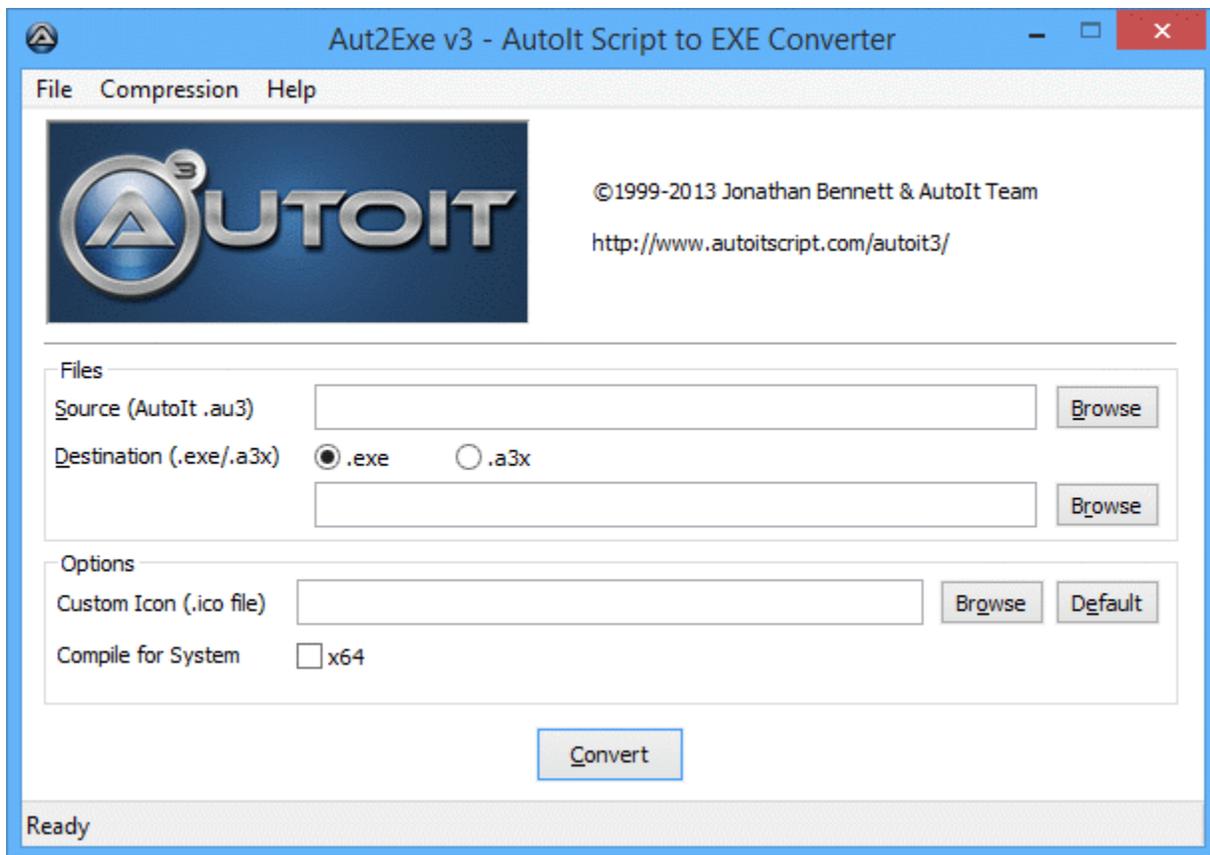
Above image from the AutoIt help file.

Just hit the convert button after you have made all your selections.

 NOTE: If your script has errors they won't be identified in the compiling process which compiles "as-is" – so run your script and check it before you compile.

Second Method:

You can simply right click on your script and compile it. When you do this the exe will have the same name as the script and the last used icon will be applied (because you don't have an opportunity to supply options / choices for these).

Third Method:

Users can also compile scripts from the command line. The command line can be seen when you run cmd.exe on your PC. You can also run command line functions from within your script (one way to do that is with the built in ShellExecute function (see the help file). This means that you could create your own script to compile scripts. A full list of the command line parameters can be found in the Autoit help file.

Chapter 17 AutoIt Forum Rules

The AutoIt forum is a fantastic place to get valuable assistance. The members are extremely helpful. You will more than likely get a very prompt response to just about any question – as long as it conforms to the forum rules. You can check the AutoIt forum for a complete listing but they don't tolerate questions about automating games (i.e. use of game "bots"), bypassing security, being disrespectful, or any discussions about decompiling code (the practice of taking a compiled program and trying to turn it back into a script / source code – which is usually indicative of someone trying to steal it).

You can find the forum here: https://www.autoitscript.com/forum/

There are various categories of discussion contained within the forum. Try to always post your question in the most appropriate category. This is usually, but not always, "AutoIt General Help and Support".

Chapter 18 Links To Source Code Examples Used In This Text

Source now included in the Appendix to this text. It may also be downloaded as a companion file on the AutoIt forum download page.

Chapter 19 Conclusion

Congratulations. If you made it this far you have learned the basic building blocks of programming. You learned about different types of data, variables, operators, conditional logic, loops, GUIs and more. At this point you may be thinking about writing your own program. You may even have an idea for something you want to build. Keep in mind; this was an introductory text for the basics. Your ideas may include concepts that weren't covered. However, knowing the information contained in this text will give the ability to figure out just about anything with a little research.

It is also important to note that these concepts don't necessarily gel into efficient programming overnight. It reminds me of the old joke: "How do you get to Carnegie Hall?" whose punch line is "Practice, practice, practice". I suppose it would be the same as someone asking "how do I learn how to program?"

As you approach your practice projects try to break them into pieces. Any large problem is easier to solve when you break it into pieces – coding is no exception. You can even test pieces of code outside your main script and then incorporate them into the larger program after you are confident they are working.

Best wishes in your programming endeavors.

Appendix

This appendix contains all the source code referenced in the book. It was auto-generated to a text file using the following script:

```autoit
; create appendix files with source code
#include <File.au3>
#include <Array.au3>

$codeDirectory=@ScriptDir&"\book code\" ; the directory containing the code samples
$dirArray=_FileListToArray($codeDirectory) ; a UDF that puts the chapter directories in an array listing

$appendixFile=FileOpen(@ScriptDir&"\appendix.txt",2) ; the text file that will contain all the samples

; This loops the chapters, adds the separators for chapters and code
; reads the file contents and writes them all to the new text file
For $a=1 to UBound($dirArray)-1
    $filesArray=_FileListToArray($codeDirectory&$dirArray[$a])
    FileWriteLine($appendixFile,"###########################################")
    FileWriteLine($appendixFile,$dirArray[$a])
    FileWriteLine($appendixFile,"###########################################")
    for $b=1 to ubound($filesArray)-1
        IF StringInStr($filesArray[$b],".au3")<> 0 then
            FileWriteLine($appendixFile,"*************************************")
            FileWriteLine($appendixFile,$filesArray[$b])
            FileWriteLine($appendixFile,"*************************************")
            $example=FileRead($codeDirectory&$dirArray[$a]&"\"&$filesArray[$b])
            FileWrite($appendixFile,$example&@crlf)
            FileWriteLine($appendixFile,@crlf) ; add some space between examples
        else
            ContinueLoop
        EndIf
    Next
Next
; close the file so it can be read
FileClose($appendixFile)
```

This example was not called out in the text – but it is the calculation I ran when discussing scientific notation.

```
###########################################
chapter 03 Hello Operator
###########################################
*************************************
Example1.au3
*************************************

$firstNumber = 1.826395e7
$secondNumber =23.34e3
$answer=$firstNumber*$secondNumber
ConsoleWrite(@crlf&$answer&@CRLF)
```

```
###########################################
Chapter 04 Lets Program Something
###########################################
**********************************
Example1.au3
**********************************
#include <MsgBoxConstants.au3>
MsgBox($MB_OK,"My first program","Hello World"); this is a comment - good job


**********************************
Example2.au3
**********************************
#include <MsgBoxConstants.au3>
$firstMessage = "Hello "
$secondMessage = "World"
MsgBox($MB_OK,"My second program",$firstMessage&$secondMessage)

###########################################
Chapter 05 conditional statements
###########################################
**********************************
Example1.au3
**********************************
#include <MsgBoxConstants.au3>
$age = 17
if $age>=18 Then
        MsgBox($MB_OK,"Voting Answer","The person can vote")
Else
        MsgBox($MB_OK,"Voting Answer","The person cannot vote")
EndIf

**********************************
Example2.au3
**********************************
#include <MsgBoxConstants.au3>
$age = 18
$nationality = "Canadian"

if $age>= 18 and $nationality = "US" then
                MsgBox($MB_OK,"Voting Answer","The person can vote")
Else
                MsgBox($MB_OK,"Voting Answer","The person cannot vote")
EndIf


**********************************
```

Example3.au3
****
```
#include <MsgBoxConstants.au3>
$age = 18
$nationality = "Canadian"

if $nationality = "US" then
        if $age>=18 then
                MsgBox($MB_OK,"Voting Answer","The person can vote")
        Else
                MsgBox($MB_OK,"Voting Answer","The person cannot vote")
        EndIf
Else
        MsgBox($MB_OK,"","Sorry. You are not a US citizen.")
EndIf
```

****
Example4.au3
****
```
#include <MsgBoxConstants.au3>
$expression = 5

Switch $expression
   Case 1
      $sMsg = "The value is 1"
   Case 2
      $sMsg = "The value is 2"
   Case 3
      $sMsg = "The value is 3"
   Case Else
      $sMsg = "The value is something other than 1,2,or 3"
EndSwitch

MsgBox($MB_OK, "", $sMsg)
```

##
Chapter 06 loops
##
****
Example1.au3
****
```
; ConsoleWrite example
ConsoleWrite("Hello World")
```

```
**********************************
Example2.au3
**********************************
; ConsoleWrite example
ConsoleWrite("Hello World"&&@CRLF)



**********************************
Example3.au3
**********************************
; ConsoleWrite example
for $a=0 to 9
ConsoleWrite("Hello World"&&@CRLF)
next


**********************************
Example4.au3
**********************************
; ConsoleWrite example
for $a=0 to 9
ConsoleWrite("This is a: " & $a &&@CRLF)
next


**********************************
Example5.au3
**********************************
; ConsoleWrite example
$loops = 1
for $a=0 to 1
        for $b = 0 to 9
                ConsoleWrite("This is a: " & $b &&@CRLF)
        next
        ConsoleWrite("Inner loop:"&$loops& " set finsished"&&@CRLF)
        $loops+=1
next


**********************************
Example6.au3
**********************************
; while loop example
$a=0

while $a<10
        consolewrite("$a is less then 10.  The value is: "&$a&&@CRLF)
        $a+=1
Wend
```

```
###########################################
Chapter 07 user functions
###########################################
************************************
Example1.au3
************************************
; An example of a user function
#include <MsgBoxConstants.au3>

Func _myFunction()
MsgBox($MB_OK,"User Function","Hello World")

EndFunc
_myFunction()


************************************
Example2.au3
************************************
; An example of a user function
#include <MsgBoxConstants.au3>

Func _mathWithAnswers($firstNum,$secondNum,$operator)

Switch $operator
        case $operator = "+"
                $answer=$firstNum+$secondNum
        case $operator = "-"
                $answer=$firstNum-$secondNum
        case $operator = "*"
                $answer=$firstNum*$secondNum
        case $operator = "/"
EndSwitch
MsgBox($MB_OK,"Answer",$answer)
EndFunc

_mathWithAnswers(5,3,"+")


************************************
Example3.au3
************************************
; An example of a user function
#include <MsgBoxConstants.au3>

Func _mathWithAnswers($firstNum,$secondNum,$operator)
```

```
Switch $operator
        case $operator = "+"
                $answer=$firstNum+$secondNum
        case $operator = "-"
                $answer=$firstNum-$secondNum
        case $operator = "*"
                $answer=$firstNum*$secondNum
        case $operator = "/"
                if $secondNum = 0 Then
                        MsgBox($MB_OK,"Error","You cannot divide by zero")
                else
                        $answer=$firstNum/$secondNum
                EndIf
EndSwitch
MsgBox($MB_OK,"Answer",$answer)
EndFunc

_mathWithAnswers(5,0,"/")
```

```
#############################################
Chapter 08 Arrays
#############################################
************************************
Example1.au3
************************************
#include <MsgBoxConstants.au3>
global $namesArray[3]=["John","Steve","Bob"]
MsgBox($MB_OK,"Who is second in the array?",  "Answer: "&$namesArray[1])
```

```
************************************
Example2.au3
************************************
#include <MsgBoxConstants.au3>
dim $people[3][2] = [["John",34],["Steve",23],["Bob",41]]
MsgBox($MB_OK,"",$people[1][0]&" is "&$people[1][1]&" years of age")
```

```
************************************
Example3.au3
************************************
#include <MsgBoxConstants.au3>
dim $people[3][2] = [["John",34],["Steve",23],["Bob",41]]

for $a=0 to UBound($people)-1
        MsgBox($MB_OK,"",$people[$a][0]&" is "&$people[$a][1]&" years of age")
Next
```

```
#############################################
Chapter 09 GUIs
#############################################
************************************
calc.au3
************************************
#include <MsgBoxConstants.au3>
; An example of a user function

Func _mathWithAnswers($firstNum,$secondNum,$operator)

Switch $operator
        case $operator = "+"
                $answer=$firstNum+$secondNum
        case $operator = "-"
                $answer=$firstNum-$secondNum
        case $operator = "*"
                $answer=$firstNum*$secondNum
        case $operator = "/"
                if $secondNum = 0 Then
                        MsgBox($MB_OK,"Error","You cannot divide by zero")
                else
                        $answer=$firstNum/$secondNum
                EndIf
EndSwitch

MsgBox($MB_OK,"Answer",$answer)
EndFunc

;_mathWithAnswers(5,0,"/")

************************************
Example1.au3
************************************
#include <GUIConstantsEx.au3>
#include <WindowsConstants.au3>
GUIcreate("Test GUI")
GUISetState(@SW_SHOW)
While 1
        $nMsg = GUIGetMsg()
        Switch $nMsg
                Case $GUI_EVENT_CLOSE
                        Exit

        EndSwitch
WEnd
```

```
**********************************
Example2.au3
**********************************
#include <GUIConstantsEx.au3>
#include <WindowsConstants.au3>
GUIcreate("Test GUI")
GUISetState(@SW_SHOW)
GUICtrlCreateButton( "Push ME", 175,200,75,50)
While 1
        $nMsg = GUIGetMsg()
        Switch $nMsg
                Case $GUI_EVENT_CLOSE
                        Exit

        EndSwitch
WEnd



**********************************
Example3.au3
**********************************
#include <MsgBoxConstants.au3>
#include <GUIConstantsEx.au3>
#include <WindowsConstants.au3>
GUIcreate("Test GUI")
GUISetState(@SW_SHOW)
$button=GUICtrlCreateButton( "Push ME", 175,200,75,50)
While 1
        $nMsg = GUIGetMsg()
        Switch $nMsg
                case $button
                        MsgBox($MB_OK,"Button Push","You pushed me!")
                Case $GUI_EVENT_CLOSE
                        Exit

        EndSwitch
WEnd



**********************************
Example4.au3
**********************************
#include <MsgBoxConstants.au3>
#include <GUIConstantsEx.au3>
#include <WindowsConstants.au3>
```

```
$guiWidth=300 ; the width of our GUI
$guiHeight=300 ; the height of our GUI

$guiBtnWidth=75 ; the width of our button
$guiBtnHeight=50 ; the heighr of our button

$inputWidth=150
$inputHeight=30

$buttonTop=($guiHeight/2)-($guiBtnHeight/2)
$buttonLeft=($guiWidth/2)-($guiBtnWidth/2)
$inputLeft=($guiWidth/2)-($inputWidth/2)
$inputTop=(($guiHeight/2)-($guiBtnHeight/2))-($inputHeight+10)


GUIcreate("Test GUI",$guiWidth,$guiHeight) ; using our variables to set height & width
GUISetState(@SW_SHOW)
;on the following line we are using some math and some variables to center the button by taking into
account the
; width and height of the GUI and button
$button = GUICtrlCreateButton ( "Push ME",$buttonLeft,$buttonTop,$guiBtnWidth,$guiBtnHeight)
$input=GUICtrlCreateInput("Enter text here",$inputLeft,$inputTop,$inputWidth,$inputHeight)

While 1
        $nMsg = GUIGetMsg()
        Switch $nMsg
                case $button
                        MsgBox($MB_OK,"Button Push","You pushed me!")
                Case $GUI_EVENT_CLOSE
                        Exit

        EndSwitch
WEnd


***********************************
Example5.au3
***********************************
#include <MsgBoxConstants.au3>
#include <GUIConstantsEx.au3>
#include <WindowsConstants.au3>

$guiWidth=300 ; the width of our GUI
$guiHeight=300 ; the height of our GUI

$guiBtnWidth=75 ; the width of our button
$guiBtnHeight=50 ; the height of our button
```

```
$inputWidth=150
$inputHeight=30

$buttonTop=($guiHeight/2)-($guiBtnHeight/2)
$buttonLeft=($guiWidth/2)-($guiBtnWidth/2)
$inputLeft=($guiWidth/2)-($inputWidth/2)
$inputTop= (($guiHeight/2)-($guiBtnHeight/2))-(($inputHeight*2)+10)
$input2Top=$inputTop+($inputHeight+10)


$comboWidth=50
$comboHeigh=30
$comboLeft=$inputLeft-($comboWidth+10)
$comboTop=(($inputTop+$input2Top)/2)

GUIcreate("Test GUI",$guiWidth,$guiHeight) ; using our variables to set height & width
GUISetState(@SW_SHOW)
; on the following line we are using some math and some variavles to center the button by taking into account the
; width and height of the GUI and the button
$button = GUICtrlCreateButton ( "Push ME",$buttonLeft,$buttonTop,$guiBtnWidth,$guiBtnHeight)
$input=GUICtrlCreateInput("Enter text here",$inputLeft,$inputTop,$inputWidth,$inputHeight)
$input2=GUICtrlCreateInput("Enter text here",$inputLeft,$input2Top,$inputWidth,$inputHeight)
$combo=GUICtrlCreateCombo("Oper",$comboLeft,$comboTop,$comboWidth,$comboHeigh)
While 1
        $nMsg = GUIGetMsg()
        Switch $nMsg
                case $button
                        $inputcontents=GUICtrlRead($input)
                        MsgBox($MB_OK,"","Text from input ",$inputcontents)
                Case $GUI_EVENT_CLOSE
                        Exit

        EndSwitch
WEnd


**********************************
Example6.au3
**********************************
#include <GUIConstantsEx.au3>
#include <WindowsConstants.au3>
#include <calc.au3>

$guiWidth=300 ; the width of our GUI
$guiHeight=300 ; the height of our GUI

$guiBtnWidth=75 ; the width of our button
```

```
$guiBtnHeight=50 ; the height of our button

$inputWidth=150
$inputHeight=30

$buttonTop=($guiHeight/2)-($guiBtnHeight/2)
$buttonLeft=($guiWidth/2)-($guiBtnWidth/2)
$inputLeft=($guiWidth/2)-($inputWidth/2)
$inputTop= (($guiHeight/2)-($guiBtnHeight/2))-(($inputHeight*2)+10)
$input2Top=$inputTop+($inputHeight+10)


$comboWidth=50
$comboHeigh=30
$comboLeft=$inputLeft-($comboWidth+10)
$comboTop=(($inputTop+$input2Top)/2)

GUIcreate("Test GUI",$guiWidth,$guiHeight) ; using our variables to set height & width
GUISetState(@SW_SHOW)
; on the following line we are using some math and some variavles to center the button by taking into account the
; width and height of the GUI and the button
$button = GUICtrlCreateButton ( "Push ME",$buttonLeft,$buttonTop,$guiBtnWidth,$guiBtnHeight)
$input=GUICtrlCreateInput("",$inputLeft,$inputTop,$inputWidth,$inputHeight)
$input2=GUICtrlCreateInput("",$inputLeft,$input2Top,$inputWidth,$inputHeight)
$combo=GUICtrlCreateCombo("Oper",$comboLeft,$comboTop,$comboWidth,$comboHeigh)
GUICtrlSetData(-1, "+|-|/|*", "+")

;GUICtrlCreateInput(
While 1
        $nMsg = GUIGetMsg()
        Switch $nMsg
              case $button
                     $firstNum=GUICtrlRead($input)
                     $secondNum=GUICtrlRead($input2)
                     $operator=GUICtrlRead($combo)
                     _mathWithAnswers($firstNum,$secondNum,$operator)
              Case $GUI_EVENT_CLOSE
                     Exit


        EndSwitch
WEnd


###############################################
Chapter 10 Koda
###############################################
***********************************
```

```
calc.au3
***********************************
#include <MsgBoxConstants.au3>
; An example of a user function

Func _mathWithAnswers($firstNum,$secondNum,$operator)

Switch $operator
        case $operator = "+"
                $answer=$firstNum+$secondNum
        case $operator = "-"
                $answer=$firstNum-$secondNum
        case $operator = "*"
                $answer=$firstNum*$secondNum
        case $operator = "/"
                if $secondNum = 0 Then
                        MsgBox($MB_OK,"Error","You cannot divide by zero")
                else
                        $answer=$firstNum/$secondNum
                EndIf
EndSwitch

MsgBox($MB_OK,"Answer",$answer)
EndFunc

;_mathWithAnswers(5,0,"/")


***********************************
Example1.au3
***********************************
#include <ButtonConstants.au3>
#include <ComboConstants.au3>
#include <EditConstants.au3>
#include <GUIConstantsEx.au3>
#include <WindowsConstants.au3>

#include <calc.au3>

#Region ### START Koda GUI section ### Form=c:\documents and settings\administrator\my
documents\book\learn to program book\learn to program book\programs\chepter
10\kodacalculatorgui.kxf
$Form1_1 = GUICreate("Form1", 301, 301, 188, 121)
$input = GUICtrlCreateInput("", 77, 55, 150, 21)
$button = GUICtrlCreateButton("Push ME", 113, 125, 75, 50)
$Input2 = GUICtrlCreateInput("", 77, 95, 150, 21)
$combo = GUICtrlCreateCombo("", 17, 75, 50, 30, BitOR($CBS_DROPDOWN,$CBS_AUTOHSCROLL))
GUICtrlSetData(-1, "+|-|/|*")
GUISetState(@SW_SHOW)
```

```
#EndRegion ### END Koda GUI section ###


;GUICtrlCreateInput(
While 1
        $nMsg = GUIGetMsg()
        Switch $nMsg
                case $button
                        $firstNum=GUICtrlRead($input)
                        $secondNum=GUICtrlRead($input2)
                        $operator=GUICtrlRead($combo)
                        _mathWithAnswers($firstNum,$secondNum,$operator)
                Case $GUI_EVENT_CLOSE
                        Exit


        EndSwitch
WEnd
```

```
################################################
Chapter 11 Strings
################################################
***********************************
Example1.au3
***********************************
#include <MsgBoxConstants.au3>
$myString = "The quick brown fox jumped over the log"
$foxPosition = StringInStr($myString,"fox")
MsgBox($MB_OK,"",$foxPosition)



***********************************
Example2.au3
***********************************
#include <MsgBoxConstants.au3>
$myString = "The quick brown fox jumped over the log"
$myStringLen = StringLen($myString)
MsgBox($MB_OK,"",$myStringLen)



***********************************
Example3.au3
***********************************
#include <MsgBoxConstants.au3>
$myString = "The quick brown fox jumped over the log"
$newString=StringReplace($myString,"fox","dog")
MsgBox($MB_OK,"",$newString)
```

```
***********************************
Example4.au3
***********************************
#include<Array.au3>
$names=" Bob, Jane, Sammy, David, Jonah, Billy, Duke, Lizzie, Greg"
$nameArray=StringSplit($names,",")
_ArrayDisplay($nameArray,"Names Array")



***********************************
Example5.au3
***********************************
#include<Array.au3>
$names=" Bob, Jane, Sammy, David, Jonah, Billy, Duke, Lizzie, Greg"
$nameArray=StringSplit($names,",")
;_ArrayDisplay($nameArray,"Names Array")

for $a=1 to UBound($nameArray)-1
        ConsoleWrite(@crlf&"This is a special message for"&$nameArray[$a]&&@crlf)
Next

***********************************
Example6.au3
***********************************
#include <MsgBoxConstants.au3>
$myString = "The quick brown fox jumped over the log"
$foxPosition = StringInStr($myString,"fox")
$trimmedString=StringTrimLeft($myString,$foxPosition+3)
MsgBox($MB_OK,"",$trimmedString)

***********************************
Example7.au3
***********************************
#include <MsgBoxConstants.au3>
$myString = "The quick brown fox jumped over the log"
$searchTerm="fox"
$searchtermLen=StringLen($searchTerm)
$foxPosition = StringInStr($myString,$searchTerm)
$trimmedString=StringTrimLeft($myString,$foxPosition+$searchtermLen)
MsgBox($MB_OK,"",$trimmedString)

***********************************
Example8.au3
***********************************
#include <MsgBoxConstants.au3>
$myString = "The quick brown fox jumped over the log"
$newString=StringTrimRight($myString,3)
MsgBox($MB_OK,"",$newString)
```

```
**********************************
Example9.au3
**********************************
#include <MsgBoxConstants.au3>
$myString = "The quick brown fox jumped over the log"
$trimmedString=StringLeft($myString,StringLen($myString)-3)
MsgBox($MB_OK,"",$trimmedString)


#############################################
Chapter 12 files and directories
#############################################
**********************************
Example01.au3
**********************************
$myFile = FileOpen(@ScriptDir&"\myfile.txt",2)



**********************************
Example02.au3
**********************************
$myFile = FileOpen(@ScriptDir&"\myfile.txt",2)
$myString = "The quick brown fox jumped over the log"
FileWrite($myFile,$myString)



**********************************
Example03.au3
**********************************
$myFile = FileOpen(@ScriptDir&"\myfile.txt",2)
$names=" Bob, Jane, Sammy, David, Jonah, Billy, Duke, Lizzie, Greg"
$nameArray=StringSplit($names,",")

for $a=1 to UBound($nameArray)-1
        FileWrite($myFile,@crlf&"This is a special message for"&$nameArray[$a]&@crlf)
Next
FileClose($myFile)

**********************************
Example04.au3
**********************************
#include <MsgBoxConstants.au3>
$fileIn = FileRead(@ScriptDir&"\myfile.txt")
MsgBox($MB_OK,"",$fileIn)

**********************************
Example05.au3
**********************************
```

```
FileCopy ( @ScriptDir&"\myfile.txt",@DesktopDir)


************************************
Example06.au3
************************************
$file = FileOpenDialog("Select File",@ScriptDir,"Text files (*.txt)")


************************************
Example07.au3
************************************
DirCreate(@ScriptDir&"\new folder\")


************************************
Example08.au3
************************************
#include <MsgBoxConstants.au3>
$size=DirGetSize(@ScriptDir&"\new folder\")
MsgBox($MB_OK,"",$size)


************************************
Example09.au3
************************************
#include <MsgBoxConstants.au3>
FileCopy ( @ScriptDir&"\myfile.txt",@ScriptDir&"\new folder\")
$size=DirGetSize(@ScriptDir&"\new folder\")
MsgBox($MB_OK,"",$size)


************************************
Example10.au3
************************************
DirMove(@ScriptDir&"\new folder\",@ScriptDir&"\new name\",1)


************************************
Example11.au3
************************************
DirRemove(@ScriptDir&"\new name\",1)


************************************
Example12.au3
************************************
#include <MsgBoxConstants.au3>
MsgBox($MB_OK,"",DriveSpaceFree("c:\"))


************************************
```

Example13.au3
```
***********************************
#include<Array.au3>
$driveArray=DriveGetDrive("ALL")
_ArrayDisplay($driveArray)
```

```
#############################################
Chapter 13 Macros
#############################################
***********************************
```
Example1.au3
```
***********************************
#include <MsgBoxConstants.au3>
MsgBox($MB_OK,"",@OSVersion)
```

```
***********************************
```
Example2.au3
```
***********************************
#include <MsgBoxConstants.au3>
MsgBox($MB_OK,"",@MON&"/"&@MDAY&"/"&@YEAR)
```

```
***********************************
```
Example3.au3
```
***********************************
#include <MsgBoxConstants.au3>
MsgBox($MB_OK,"",@MON&"/"&@MDAY&"/"&@YEAR&" Timestamp:
"&@HOUR&":"&@MIN&":"&@SEC)
```

```
#############################################
Chapter 14 User Defined Functions
#############################################
***********************************
```
Example1.au3
```
***********************************
#include<Array.au3>
dim $fruits[10]=["Grapefruit", "Banana", "Watermelon", "Grape", "Apple", "Guava","Star fruit",
"Mango", "Coconut", "Blueberry"]
_ArrayDisplay($fruits)
```

```
***********************************
```
Example2.au3
```
***********************************
#include<Array.au3>
#include <MsgBoxConstants.au3>
dim $fruits[10]=["Grapefruit", "Banana", "Watermelon", "Grape", "Apple", "Guava","Star fruit",
"Mango", "Coconut", "Blueberry"]
```

```
$grapePos=_ArraySearch($fruits,"grape")
MsgBox($MB_OK,"Position of grape",$grapePos)
```

****
Example3.au3
****

```
#include<Array.au3>
dim $fruits[10]=["Grapefruit", "Banana", "Watermelon", "Grape", "Apple", "Guava","Star fruit",
"Mango", "Coconut", "Blueberry"]
_ArraySort($fruits)
_ArrayDisplay($fruits)
```

****
Example4.au3
****

```
#include<Array.au3>
#include<File.au3>
$dirArray= _FileListToArray(@WindowsDir,"*",2)
_ArrayDisplay($dirArray,"Windows Directories")
```

****
Example5.au3
****

```
#include<File.au3>
_FilePrint(@ScriptDir&"\myfile.txt")
```

****
Example6.au3
****

```
#include <SQLite.au3>
#include <SQLite.dll.au3>

Local $hQuery, $aRow
_SQLite_Startup()
ConsoleWrite("_SQLite_LibVersion=" & _SQLite_LibVersion() & @CRLF)
_SQLite_Open()
; Without $sCallback it's a resultless statement
_SQLite_Exec(-1, "Create table tblTest (a,b int,c single not null);" & _
     "Insert into tblTest values ('1',2,3);" & _
     "Insert into tblTest values (Null,5,6);")

Local $d = _SQLite_Exec(-1, "Select rowid,* From tblTest", "_cb") ; _cb will be called for each row

Func _cb($aRow)
  For $s In $aRow
     ConsoleWrite($s & @TAB)
```

```
  Next
  ConsoleWrite(@CRLF)
  ; Return $SQLITE_ABORT ; Would Abort the process and trigger an @error in _SQLite_Exec()
EndFunc   ;==>_cb
_SQLite_Close()
_SQLite_Shutdown()

; Output:
; 1    1  2  3
; 2       5  6


**********************************
Example7.au3
**********************************
#include <SQLite.au3>
#include <SQLite.dll.au3>

Local $hQuery, $aRow
_SQLite_Startup()
ConsoleWrite("_SQLite_LibVersion=" & _SQLite_LibVersion() & @CRLF)
_SQLite_Open(@ScriptDir&"\example.db")
Local $d = _SQLite_Exec(-1, "Select rowid,* From tblTest", "_cb") ; _cb will be called for each row

Func _cb($aRow)
  For $s In $aRow
    ConsoleWrite($s & @TAB)
  Next
  ConsoleWrite(@CRLF)
  ; Return $SQLITE_ABORT ; Would Abort the process and trigger an @error in _SQLite_Exec()
EndFunc   ;==>_cb
_SQLite_Close()
_SQLite_Shutdown()

; Output:
; 1    1  2  3
; 2       5  6


#############################################
Chapter 15 Automating prgrams
#############################################
**********************************
Example01.au3
**********************************
$notepad=WinActivate("Untitled - Notepad")
sleep(2000)
$calculator=WinActivate("Calculator")
```

```
***********************************
Example02.au3
***********************************
$notepad=WinActivate("Untitled - Notepad")
sleep(2000)
$calculator=WinActivate("Calculator")

WinClose($notepad)

***********************************
Example03.au3
***********************************
$notepad=WinActivate("Untitled - Notepad")
WinWaitActive($notepad,"",3)
Send("Learning about automation is fun")



***********************************
Example04.au3
***********************************
run("notepad.exe")
WinWaitActive("Untitled - Notepad")
Send("Learning about automation is fun")



***********************************
Example05.au3
***********************************
run("notepad.exe")
WinWaitActive("Untitled - Notepad")
Send("!F")



***********************************
Example06.au3
***********************************
; First we opened an instance of calculator
run("calc.exe")
$calc=WinWaitActive("Calculator") ; activates the window

; Panes
$controlTopPane = "[CLASS:Static; INSTANCE:2]" ; pane above numbers
$controlNumberPane = "[CLASS:Static; INSTANCE:4]"; numbers pane
; update panes with our text
ControlSetText("Calculator","",$controlTopPane,"Automate with AutoIt!")
ControlSetText("Calculator","",$controlNumberPane,"Yeah!!!")
```

```
;Buttons
$controlOneBtn = "[CLASS:Button; INSTANCE:5]"; one button
$controlTwoBtn = "[CLASS:Button; INSTANCE:11]"; two button
$controlThreeBtn = "[CLASS:Button; INSTANCE:16]"; three button
$controlFourBtn = "[CLASS:Button; INSTANCE:4]"; four button
$controlFiveBtn = "[CLASS:Button; INSTANCE:10]"; five button
$controlSixBtn = "[CLASS:Button; INSTANCE:15]"; six button
$controlSevenBtn = "[CLASS:Button; INSTANCE:3]"; seven button
$controlEightBtn = "[CLASS:Button; INSTANCE:9]"; eight button
$controlNineBtn = "[CLASS:Button; INSTANCE:14]"; nine button
$controlZeroBtn = "[CLASS:Button; INSTANCE:6]"; nine button

; an array of the buttons that we can loop through to update them all
dim
$btnArray[10]=[$controlOneBtn,$controlTwoBtn,$controlThreeBtn,$controlFourBtn,$controlFiveBtn,$c
ontrolSixBtn,$controlSevenBtn,$controlEightBtn,$controlNineBtn,$controlZeroBtn]

$text="A" ; used to update all buttons
; The loop that sets all the buttons to "A"
for $a=0 to ubound($btnArray)-1
ControlSetText("Calculator","",$btnArray[$a],$text)
next


***********************************
Example07.au3
***********************************
; First we opened an instance of calculator
run("calc.exe")
$calc=WinWaitActive("Calculator") ; activates the window

; Panes
$controlTopPane = "[CLASS:Static; INSTANCE:2]" ; pane above numbers
$controlNumberPane = "[CLASS:Static; INSTANCE:4]"; numbers pane
; update panes with our text
ControlSetText("Calculator","",$controlTopPane,"Automate with AutoIt!")
ControlSetText("Calculator","",$controlNumberPane,"Yeah!!!")

;Buttons
$controlOneBtn = "[CLASS:Button; INSTANCE:5]"; one button
$controlTwoBtn = "[CLASS:Button; INSTANCE:11]"; two button
$controlThreeBtn = "[CLASS:Button; INSTANCE:16]"; three button
$controlFourBtn = "[CLASS:Button; INSTANCE:4]"; four button
$controlFiveBtn = "[CLASS:Button; INSTANCE:10]"; five button
$controlSixBtn = "[CLASS:Button; INSTANCE:15]"; six button
$controlSevenBtn = "[CLASS:Button; INSTANCE:3]"; seven button
$controlEightBtn = "[CLASS:Button; INSTANCE:9]"; eight button
$controlNineBtn = "[CLASS:Button; INSTANCE:14]"; nine button
```

```
$controlZeroBtn = "[CLASS:Button; INSTANCE:6]"; nine button

; an array of the buttons that we can loop through to update them all
dim
$btnArray[10]=[$controlOneBtn,$controlTwoBtn,$controlThreeBtn,$controlFourBtn,$controlFiveBtn,$c
ontrolSixBtn,$controlSevenBtn,$controlEightBtn,$controlNineBtn,$controlZeroBtn]

$text="A" ; used to update all buttons
; The loop that sets all the buttons to "A"
for $a=0 to ubound($btnArray)-1
ControlSetText("Calculator","",$btnArray[$a],$text)
next

sleep (2000) ; small delay to view original results
send("!2") ; invoke scientific calc mode ALT + 2 shortcut
$radiansRadio="[CLASS:Button; INSTANCE:30]" ; the radians radio button Au3Info
WinWaitActive("Calculator"); wait for new mode to become active
ControlCommand("Calculator","",$radiansRadio,"check"); select radian radio


**********************************
Example08.au3
**********************************
Sleep (2000)
$mousePos=MouseGetPos()
$mouseX=$mousePos[0]
$mouseY=$mousePos[1]
mousemove($mouseX+100,$mouseY+100)



**********************************
Example09.au3
**********************************
#include<Excel.au3>
$oExcel = _Excel_Open()



**********************************
Example10.au3
**********************************
#include<Excel.au3>
$oExcel = _Excel_Open()
$workbook=_Excel_BookNew($oExcel, 3)



**********************************
Example11.au3
**********************************
```

```
#include<Excel.au3>
#include<Array.au3>
$oExcel = _Excel_Open()
$workbook=_Excel_BookNew($oExcel, 3)

global $people[3][2] = [["John",34],["Steve",23],["Bob",41]]
_Excel_RangeWrite($workbook,1,$people)



************************************
Example12.au3
************************************
#include<Excel.au3>
#include<Array.au3>
$oExcel = _Excel_Open()
$workbook=_Excel_BookNew($oExcel, 3)

global $people[3][2] = [["John",34],["Steve",23],["Bob",41]]
_Excel_RangeWrite($workbook,1,$people)

_Excel_BookSaveAs($workbook,@ScriptDir&"\names.xlsx",default,True)
_Excel_BookClose($workbook)



************************************
Example13.au3
************************************
#include <MsgBoxConstants.au3>
#include<Excel.au3>
#include<Array.au3>
$oExcel = _Excel_Open()
$workbook=_Excel_BookNew($oExcel, 3)

global $people[3][2] = [["John",34],["Steve",23],["Bob",41]]
_Excel_RangeWrite($workbook,1,$people)

_Excel_BookSaveAs($workbook,@ScriptDir&"\names.xlsx",default,True)
_Excel_BookClose($workbook)
MsgBox($MB_OK,"","Workbook closed click okay to reopen")
_Excel_BookOpen($oExcel,@ScriptDir&"\names.xlsx")

************************************
Example14.au3
************************************
#include <MsgBoxConstants.au3>
#include<Excel.au3>
#include<Array.au3>
$oExcel = _Excel_Open()
```

```autoit
$workbook=_Excel_BookNew($oExcel, 3)

global $people[3][2] = [["John",34],["Steve",23],["Bob",41]]
_Excel_RangeWrite($workbook,1,$people)

_Excel_BookSaveAs($workbook,@ScriptDir&"\names.xlsx",default,True)
_Excel_BookClose($workbook)
MsgBox($MB_OK,"","Workbook closed click okay to reopen")
$newworkbook=_Excel_BookOpen($oExcel,@ScriptDir&"\names.xlsx")
sleep(2000)
$data=_Excel_RangeRead($newworkbook)
_ArrayDisplay($data)
```

This example uses an image from the AutoIt website that you will need to store in the same directory.
You can find it here: http://www.autoitscript.com/images/logo_autoit_210x72@2x.png
****
Example15.au3
****

```autoit
#include<Excel.au3>
$oExcelApp=_Excel_Open()
$oExcel = _Excel_BookNew($oExcelApp)
$path = @ScriptDir&"\au3logo.png"
$targetRange=$oExcel.ActiveSheet.Range("A1:D10")

func _InsertPictureInRange($path, $targetRange)
; inserts a picture and resizes it to fit the TargetCells range
Dim $p, $t, $l, $w, $h
  ; import picture
  $p = $oExcel.ActiveSheet.Pictures.Insert($path)
  ; determine positions
  With $targetRange
    $t = .Top
    $l = .Left
    $w = .Offset(0, .Columns.Count).Left - .Left
    $h = .Offset(.Rows.Count, 0).Top - .Top
  EndWith
  ; position picture
  With $p
    .Top = $t
    .Left = $l
    .Width = $w
    .Height = $h
  EndWith
  $p = "Nothing"
EndFunc
_InsertPictureInRange($path, $targetRange)
```

---

[i] "Jon" runs the AutoIt forum referenced in the text and does not include his last name in his posts.