



Small Visual Basic

3.0

The language reference book

Released in the 60th BASIC anniversary, 1 May 2024

Mohammad Hamdy Ghanem

To
John G. Kemeny, Thomas E. Kurtz,
Bill Gates and Alan Cooper:
We will keep your legacy alive!

In memory of:

My father **Hamdy Kamel Ghanem** (1950, 2015)
My grandmother **Om Wahba** (1938-2021)

Special thanks to:

My mother Zinab Fouad Othman
Mr. Attef Al Hedaby
Mr. Samir Mostafa
Prof. Mohsen Rashwan
Eng. Mohammad Galal
Dr. Mohamed Attia
Anthony D. Green

Note

This reference book may not be the easiest way for kids and beginners to start learning sVB with, as it serves as the language full documentation. Instead, they can start with the "Small Visual Basic Kid Programmer" book series. The Level 1 book is [published on Amazon](#).

Small Visual Basic Kid Programmer



Level 1

Hello Programming

Mohammad Hamdy

Index

[About The Author](#)

[Acknowledgment](#)

- [In memory of my grandmother](#)
- [In memory of my father](#)
- [The lady who taught me everything I know](#)
- [The Book that changed the world](#)
- [My guides along the journey](#)
- [And the list can go on, but..!](#)

Hello Small Visual Basic!

[What is Small Visual Basic?](#)

[Download the language](#)

[Try the samples](#)

[Using the sVB source code](#)

[Why do we need sVB?](#)

[It is also a Small Visual Basic .NET!](#)

- [sVB and Small Basic](#)
- [sVB and Visual Basic](#)
- [sVB and Visual Basic .NET](#)
- [sVB IDE and VS.NET IDE](#)
- [sVB and WPF](#)

[sVB for kids](#)

[Conclusions](#)

How To

[Try the samples](#)

[Fix SB variable issues in sVB](#)

[Design forms](#)

[Design menus](#)

[Animate controls](#)

[Handle control events](#)

[Evaluate math expressions at runtime](#)

[Create a project](#)

[Add a global file to the project](#)

[Create a spherical button](#)

[Use sVB to create a code library](#)

[Debug the project](#)

[Create a multi-thread program in sVB](#)

[Write and run test functions](#)

Small Visual Basic IDE

[The sVB Form Designer:](#)

- [The upper toolbar](#)
- [The side panel](#)
- [The form-surface canvas](#)
- [The Font Dialog](#)
- [The Color dialog](#)
- [The Properties Window](#)
- [The Menu Designer](#)

The sVB Code Editor:

- [Opening and saving documents](#)
- [The code document](#)
- [Code formatting and pretty listing](#)
- [Intellisense](#)
- [Auto completion](#)
- [The program toolbar](#)

Small Visual Basic Syntax

Tokens

Expressions:

- [Identifier Expressions](#)
- [Literal Expressions](#)
- [Array Expressions](#)
- [Array Initializer Expressions](#)
- [Binary Operator Expressions](#)
- [Negate Expressions](#)
- [Function and Method Call Expressions](#)
- [Property Expressions](#)
- [Dynamic property expressions](#)

Statements:

- [Assignment statements](#)
- [Label and Goto Statements](#)
- [ExitLoop Statements](#)
- [ContinueLoop Statements](#)
- [Return Statements](#)
- [Subroutine Call Statements](#)
- [If Statements](#)

- [For Statements](#)
- [ForEach Statements](#)
- [While Statements](#)
- [Sub Statements](#)
- [Function Statements](#)

sVB Projects

[Create a project](#)

[The form Files](#)

[Open an existing project](#)

[Generating the executable file](#)

[The startup form](#)

[Show another form](#)

[The Global File](#)

[Using sVB to create a code library!](#)

[Using VB.NET and C# to Create a library for sVB](#)

The sVB Debugger

[Debugging sVB projects](#)

[Types of errors](#)

[Debugger Commands](#)

[Break at errors](#)

[Debugging Test Functions](#)

[How does the debugger work?](#)

[Debugging best practice](#)

The sVB Library

Note that the LitDev external library is now installed with sVB. See [LitDev documentation](#) for more details. You can also download [litDev samples](#).

Array	ImageList
Button	Keyboard
Chars	Keys
CheckBox	Label
Clock	ListBox
Color	MainMenu
Colors	Math
ComboBox	MenuItem
Control	Mouse
Controls	Program
ControlTypes	ProgressBar
Date	RadioButton
DatePicker	ScrollBar
DemoLib	Shapes
Desktop	Slider
DialogResults	Sound
Dialogs	Stack
Dictionary	Text
Evaluator	TextBox
Event	TextWindow
File	Thread
Form	Timer
Forms	ToggleButton
GeometricPath	Turtle
Geometrics	UnitTest
GraphicsWindow	WinTimer

About The Author



- Eng. Mohammad Hamdy Ghanem.
- Born in Damietta governorate, Misr (Egypt), 1977.
- Holds a B.Sc. of Electrical Communications & Electronics from Cairo university.
- Started programming using classic Visual Basic in Jan. 1998, to write a program that analyzes the prosody of Arabic poems.
- A .NET developer since its release up to the moment.
- Published several VB.NET and C# books since 2008, to introduce the .NET platform in Arabic to beginners and professionals.
- Published a book about the Ring programming language in 2021.
- Arabic poet, and has published two poetry collections and many other poetry e-books.
- Published 10 e-book novels, most of them are science fiction, and has many other unpublished novels written before programming took over!

- Started developing Small Visual Basic since Jan. 2021.
- <https://github.com/VBAndCs>.
- msvbnet@hotmail.com

Every help is appreciated:

This book contains 750 pages and took about 6 months to be written. I only made an auto check to correct typos, but I have no enough time to thoroughly review the whole book right now, so, if you find any typo, grammatical mistakes, or anything wrong in the book, please don't hesitate to report it to [my email](#) or on the [sVB issues page](#). Also, the issues page is the right place to report any sVB bug or suggest any new feature.

Thank you.

Acknowledgment

* In memory of my grandmother (1938-2021):

A month after my grandmother **Om Wahba** passed away, may Allah rest her soul in peace, I published the first release of the Small Visual Basic language, to honor her memory.

She was a simple country lady, that had only a few years of education, but she was smart, clever and resourceful.



I spent most of my childhood with her, enjoying her kindness, and learning how to deal with hardships and use whatever simple tools available to solve problems.

She raised a big family, helped in raising her grandsons, and saw many of their sons. She lived a long fruitful life, and we will never forget her.

* In memory of my father (1950, 2015):

I have been writing novels and books for more than 30 years, some of them are free e-books, like this one and the language it introduces. In fact, my father **Hamdy Kamel Ghanem**, may Allah rest his soul in peace, is the hero who made this possible, as he provided me with all I needed to learn what I like and write what I want, which gave me the time and freedom to experiment with new ideas.

My father was a social person and a school social worker. He had many friends and kept strong social ties with everyone around him.



In a letter he sent me 30 years ago from Qatar - where he used to work - he asked me to mention my mother and acknowledge her role in my life in one of my books. I was just a high school student, and there was no Internet in Egypt then, so he was obviously encouraging me to continue, and it seems that he had succeeded. So, it is about time to finally fulfill his wish, right now, right here!

* **The lady who taught me everything I know:**

As a child, my mother, **Zinab Othman**, may Allah bless her, used an abacus and marbles to teach me basic arithmetic operations, while I was thinking we are playing!

She also helped me to read, write, and memorize the Glorious Quran. She even helped me studying English and French years later. I owe her everything I know in a direct or an indirect way, so neither sVB nor this book would exist without her.

Furthermore, I still live with her, and the time and effort she put in preparing our meals, gave me more free time to program sVB and write this book!

May Allah heal her and bless her life.

* **The Book that changed the world:**

Mentioning the Glorious Quran, I wouldn't forget to mention Sheikh **Mosaad Shorisher**, may Allah rest his soul in peace, who taught me how to recite Quran, which made me an eloquent Arabic writer and a poet.

But the Glorious Quran is not just a linguistic book. It is also full of logical arguments and scientific evidence, so it always guides the mind to look and think, which is a very solid foundation to build an engineer's and a programmer's mind:

(Say [O Muhammad]: "Travel in the land and see how Allah originated creation, and then Allah will resurrect the creation of the Hereafter. Verily, Allah is able to do all things").

You may think this is just a personal impression, however it is exactly what happened to countless number of Muslim scientists who evolved Algebra, Trigonometry, Chemistry, Medicine, Optics, Geography and Astronomy, and paved the way to the age of European Renaissance. For instance, once [Abbas Ibn Firnas](#) heard this Quran verse:

(O assembly of jinn and mankind, if you are able to pass beyond the zones of the heavens and the earth, then pass. You will not pass unless you have the authority [power]).

Surah Ar-Rahman, 33

He asked whether anyone ever tried to do that, then decided to be the first, so he made two wings and tried to fly. He actually succeeded and flew for 10 minutes, but he didn't have a tail, so he failed to land safely and broke his legs! You can see [this video](#) for more info.

Abbas Ibn Firnas was the first man to fly, and he inspired others to keep trying until we finally got the planes! NASA honored him by naming a lunar crater after his name: [Ibn Firnas Crater](#).

In fact, the verse mentioned above was the first invitation to humans to go to outer space, and the invitation is still valid to go out of our planetary system, or even our galaxy, or as far as our power can take us through the zones of the heavens!

Note that the verse above implies that heavens and earth have sides/directions/zones, and after the Quran started the Islamic scientific revolution through the fastly and vastly growing Islamic empire, the Arabic word "أقطار" (aqtar)" - that is translated to "zones" in the verse - has been used in geometry for circle and sphere diameters!

This is not the only sign in Quran that tells us that earth is a

globe! For example:

**(He wraps the night over the day,
and wraps the day over the night) Surah Az-Zumar, 5**

The literal translation for the Arabic word "يُكْوِر" (Yokawwir)" is to wrap a thing around a spherical shape. The sphere and the ball are called "الكُوره" (Alkorah)" in Arabic.

So, my point is: the Glorious Quran is full of wonders that Muslims spent centuries in discovering and still do. It is, literally, the book that took the Middle East (then Europe a few centuries later) out of the dark ages, and led the mankind to advance all the modern science and technology that we use today. We particularly honor that in a daily basis, every time we use **logarithms** in mathematics, and **algorithms** in programming, since both are named after [Al-Khwarizmi](#), the Muslim scientist who is the father of Algebra and the inventor of zero and the decimal system as we know today.

Note that "Al" in Arabic means "The" in English and "Lo" in Italic. Note also that the Arabic word "صفر" (Siphro)" became Cero in Spanish and zero in French, Italic and English. Also other words related to encryption are taken from that Arabic word such as cipher in English and chiffre in French.

So, all thanks to Allah, my true and only God, for giving the mankind such a great book.

To get a translation of the meanings of the Glorious Quran, you can install this [Android app](#) or [IOS app](#), or you can [read online](#), or [get a paper copy](#). But, be aware that what you will read is a translation of just one possible meaning of each verse, while in Arabic, the verse is loaded with many unconflicted meanings that are all true, and each of them suits the knowledge of the reader and the era he lives in. So, no translation can possibly reveal all the Quran secrets to you, but it can be sufficient to give you a glimpse of its magnificence.

* My guides along the journey:

I have written many books in Arabic; some on computer programming, and others are novels and poetry collections. But this is my first book in English where I introduce the first programming language I ever evolved, and I hope it will not be the last!

It was a long journey. I already explained above how it started, and here I will say a few words about some of many persons who made it possible for me to keep going, and to whom I offer my sincere thanks and appreciation to:

- My brothers Hamdy and Mohsen, who always support me. I actually got involved in this work while trying to make their sons learn programming.
- All my teachers, who have all my respect and admiration. I need to particularly mention **Mr. Gamal Abd Alwahed**, **Mr. Ahmad Abdo Algamal** (may Allah rest their souls in peace), **Mr. Samir Mustafa** and **Mr. Attef Al Hedaby**, who were not just my teachers that made me love Arabic, English and math, but they were also my moral compass, my elder brothers and, if I may say with due respect, my friends.
- **Prof. Dr. Mohnsen Rashwan** who made me wrote my first Visual Basic program as an assignment in his collage course about humanities, so I became a programmer thanks to him! He also was the supervisor of my graduation project, and I learnt a lot from him through humanities, signals and control courses in faculty of engineering, and a lot more later in a summer seminar in his software company, RDI.
- **Eng. Mohammad Galal**: He taught me Visual Basic in January 1998.

- **Dr. Mohamed Attia:** I learnt a lot from him about software engineering and AI, among many other things.
- **Eng. Mahmoud Fayed:** the creator of the [Ring programming language](#), that inspired me to evolve Small Basic to sVB.
- The Visual Basic community and all Visual Basic lovers who are fighting to keep it alive for next generations. I'd like to particularly mention **Vijaye Raji**, the Small Basic creator, and [Anthony D. Green](#), the former VB.NET project manager at Microsoft, who is currently evolving VB.NET to ModVB, which we expect to be the future of VB.NET with its amazing new features, particularly in web and mobile development. I crated sVB to be an easy attractive introduction to VB.NET (and ModVB) for a new generation of developers.
Besides, his [article about the anatomy of the VB.NET compiler](#) helped me to understand the Small Basic compiler source code, which was the first step to create sVB.

* **And the list can go on, but..!**

So, thank you all, the few I named and the majority I didn't, not of less gratitude, but simply because this book will never be big enough to mention you all. I am sure you know what you mean to me.

And as a small payback, I hope this work be helpful for your younger siblings, may Allah bless them all.

With all my gratitude,
Eng. Mohammad Hamdy Ghanem,
Misr (Egypt), July 2023.

Hello Small Visual Basic!

* What is Small Visual Basic?

Small Visual Basic (sVB) is an educational programming language, created by Eng. Mohammad Hamdy as an evolved version of Microsoft Small Basic (SB). It is meant to be easier and more powerful at the same time, to introduce programming basics to kids and beginners of any age, provided that they can use the English keyboard on the Windows operating system.

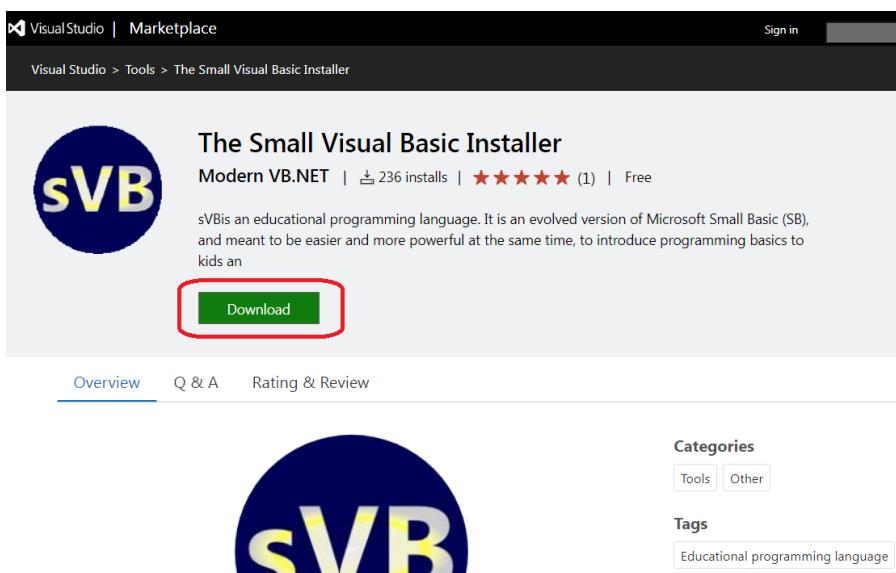
To achieve this goal, sVB has many new parts that don't exist in Small Basic, like the form designer, the small Windows forms library, the debugger, and many new syntax features and compiler and code editor improvements. For more details, see

[Why do we need sVB?](#)

In short, you can say that sVB is the Visual version of Small Basic, and the small version of Visual Basic.

* Download the language:

You can download the sVB installer from the [VS marketplace](#).



Just click the download button, then double-click the downloaded zip file, unzip it, and double-click the svb installer.msi file.

When the setup program starts, it will make sure that your PC has the .NET Framework 4.5, and if not, it will ask you to download and setup it from a provided link.

After that you can press next on each setup page to install sVB with the default options. Wait until the setup finishes in a few seconds, and close the installer when its last page confirms that the setup has completed successfully.

Now you have 4 icons related to sVB on the desktop and in the Small Visual Basic folder in the start menu, so you can double-click the sVB icon to start the sVB integrated development environment (IDE).

Congratulations, you can start programming now.

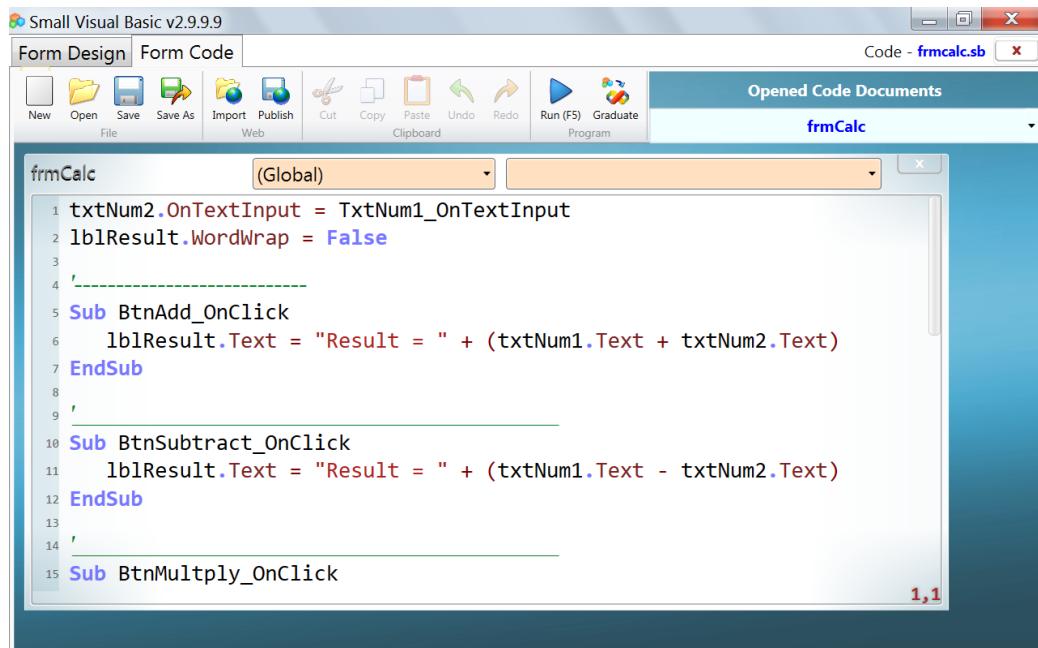
* Try the samples:

After installing Small Visual Basic, you can double-click the "sVB samples" icon on the desktop to open the sVB samples folder, which is installed on your documents folder with about 90 samples, including the Ball, Tom and Jerry, Cars, and Tetris games, and many other interesting samples.

To try any sample, open its folder and just double-click any .sb file (the one that has the sVB icon) to open it in the sVB IDE, which will show the code of this file in the code editor.

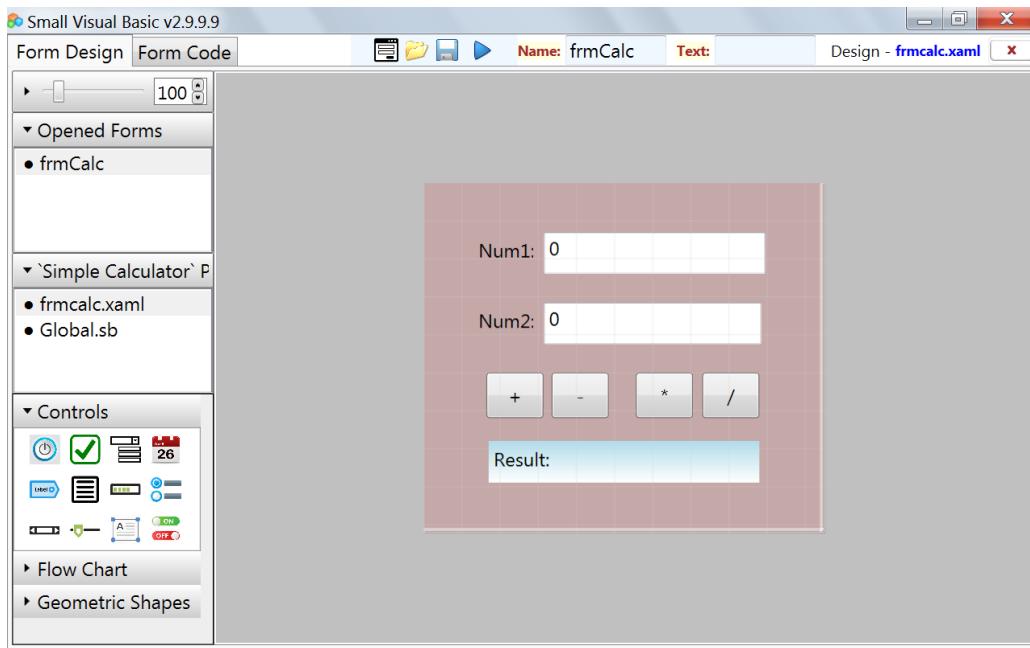
If the file is related to a form, it will also be loaded in the form designer, and you can switch to by clicking the "Form Design" tab at the top of the IDE.

The following 2 pictures shows you how the code editor and the form designer will look like when you open the Simple Calculator application from the samples folder.



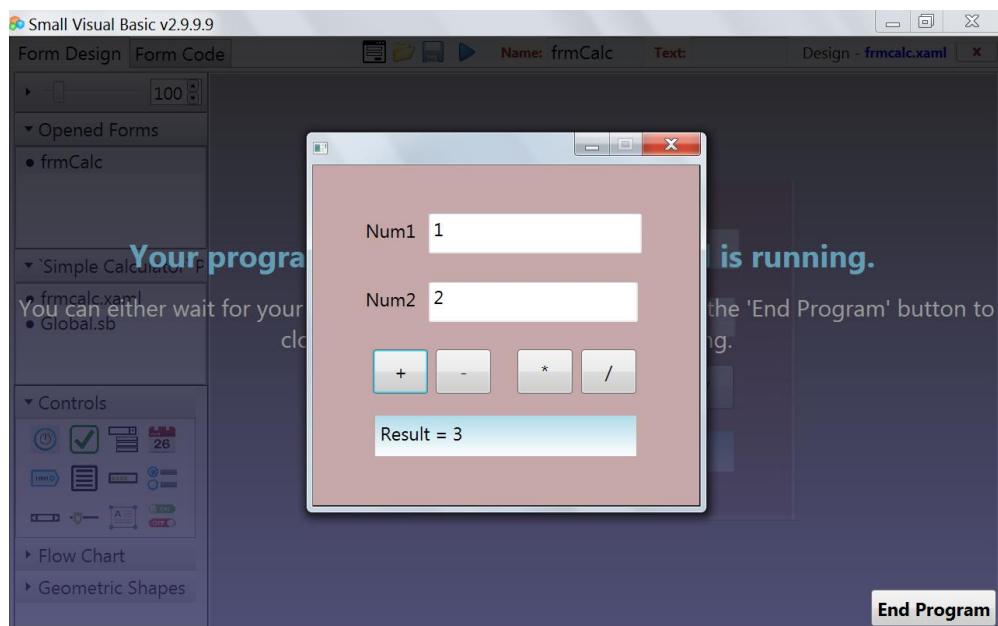
The screenshot shows the Small Visual Basic v2.9.9.9 IDE interface. The title bar reads "Small Visual Basic v2.9.9.9". The menu bar has "File", "Web", "Clipboard", "Program", and tabs for "Form Design" and "Form Code". The "Form Code" tab is selected, showing the code for the "frmCalc" form. The code editor displays the following VBScript code:

```
frmCalc (Global)
1 txtNum2.OnTextInput = TTxtNum1_OnTextInput
2 lblResult.WordWrap = False
3 '
4 -----
5 Sub BtnAdd_OnClick
6     lblResult.Text = "Result = " + (txtNum1.Text + txtNum2.Text)
7 EndSub
8 '
9 -----
10 Sub BtnSubtract_OnClick
11     lblResult.Text = "Result = " + (txtNum1.Text - txtNum2.Text)
12 EndSub
13 '
14 Sub BtnMultiply_OnClick
```



You can also open any code or form files from the IDE by clicking the Open button from the upper toolbar or pressing Ctrl+F5 from the keyboard.

Now, to run the project, click the "Run" button from the toolbar (or hit F5 from keyboard). This will display the form on the screen, where you can interact with its controls to input data and get the results as the form is programmed to do.



Note that closing the main form or clicking the "End program" button will close the program and take you back to the IDE.

* **Using the sVB source code:**

sVB is an open source project, that is published as a [GitHub repo](#).

The source code of MS Small Basic is written in C#, but I converted a copy of it to VB.NET and uses it to create sVB.

I am a professional C# programmer, but it seemed inappropriate to me to write a BASIC-family compiler with a C-family language (Not to mention that VB.NET is my favorite language).

Furthermore, I wanted to use sVB as a training yard for understanding and working with compilers, both for me and for the VB.NET community. This small compiler with small tools, provides a very easy way to understand how compilers are built and how they work, which can make it easier in a next step to work with the VB.NET compiler (Which is a part of [Roslyn](#)).

This is a necessary step for the VB community, after MS neglected VB.NET since 2017, and stopped evolving it since 2020. And this is why Anthony D. Green forked Roslyn and start evolving VB.NET under the name [ModVB](#) since 2022.

Anthony had participated in designing and creating Roslyn and VB.NET in the first place, but he is still one man, and the community must help him maintain and evolve ModVB, if they really want it to live long and prosper.

So, here is a small programming language written in VB.NET for you. You can fork it, play with it, and evolve it as you want, and when you feel comfortable with compilers, try to take a look at Roslyn and ModVB.

Note that all sVB projects are WPF projects, that target the .NET framework 4.5. You can run the source code in VS.NET 2019 and

later. But before running the code, please copy the "Lib" and "Toolbar" folders from the "SmallBasicIDE\SB.Lib" folder to both "SmallBasicIDE\bin\Debug" and "SmallBasicIDE\bin\Release" folders, as obviously "Git" excludes these folders, and I prefer it this way.

* Why do we need sVB:

"BASIC used to be on every computer a child touched, but today there's no easy way for kids to get hooked on programming."

David Brin, "Why Johnny can't code", 2006.

One day, Vijaye Raji has read the above article, so he decided to do something to solve that problem. In his spare time in Microsoft, he created the Small Basic as an educational programming language for 7 years old kids and above, and Microsoft had released it in 2008.

I haven't read this article, nor even knew that Small Basic exists, when I tried to teach VB.NET to my 13 years old nephew, which was not easy for him!

By the way, his name is Mohammad Hamdy Ghanem too, as my brother is named after our father, and his son is named after me! So, I found myself asking a similar question: Why can't Mohammad code?

There are tons of fundamental facts about VB.NET that he needs to learn before producing any thing useful.

He kept asking me about who would use such simple samples in the real world, and he was absolutely right!

I had to build a simple car racing game to get his attention, but things weren't that easy for him! User controls, objects, and inheritance?! He's absolutely got overwhelmed!

I recall that the classic Visual Basic was much easier to start with, even for someone without any programming background, but VB.NET got complicated over years while it was getting more powerful.

When I discussed this with the VB community, one developer suggested to teach him Small Basic instead, and that was the first time I hear about.

I tried to do that in the next year with my younger nephews, Ali and Omar, and they already got interested with Small Basic. Small Basic is really small, containing only 14 keywords to perform the basic programming instructions like "**Sub**", "**If**", "**For**", "**While**" and "**Goto**" statements.

Small basic is a dynamic language, as it doesn't declare variables of certain types. You just assign a value to a valid identifier and SB will declare it as a Primitive variable, which can hold a string, a number, or an array.

This makes the language very easy to learn and use for kids. So, Ali and Omar were happy with SB, until they reached the graphics window drawings that depend on the sine and cosine functions, which were hard to them, because they were still in the 4th grade without any trigonometry background!

The PDF book that comes with SB makes it hard by focusing on drawing shapes by using trigonometric functions (it even contains a fractals sample!)

This is not the best way to introduce programming to kids. A black command window (the Text Window) is easy but boring, while using vector graphics or drawing using the turtle on the Graphics Window is amazing but can be quite hard.

The good news is that the Controls class allows you to draw a TextBox, and a Button on the Graphics Window, deal with their properties and handle their events. But, unfortunately, the kid has

to design the form blindly while adding controls by code. Furthermore, the code used to communicate with these controls is verbose, because SB doesn't have objects, so, you can only store the name of the control in a variable, then send it to static/shared methods to change its properties or perform its tasks. For example:

```
Btn = Controls.AddButton("Enable", 100, 100)
Controls.ButtonClicked = OnClick

Sub OnClick
    If Controls.GetButtonCaption(Btn) = "Enable" Then
        Controls.SetButtonCaption(Btn, "Disable")
    Else
        Controls.SetButtonCaption(Btn, "Enable")
    EndIf
EndSub
```

This is not the kind of code you want to show to a kid! In fact it will be easier to teach him Visual Basic, so he can drag a button from the toolbox, drop it on the window, set its name and caption from the properties window, double-click it to go to its click event handler in the code editor, and just write:

```
If btn.Text = "Enable"
    btn.Text = "Disable"
Else
    btn.Text = "Enable"
EndIf
```

And that's it. A fast, clean, easy and short code, that made us love programming!

It is unbelievable that SB complicated such an easy task, in the name of being simple and easy to learn for kids!

I tried some SB alternative IDEs, but they are either:

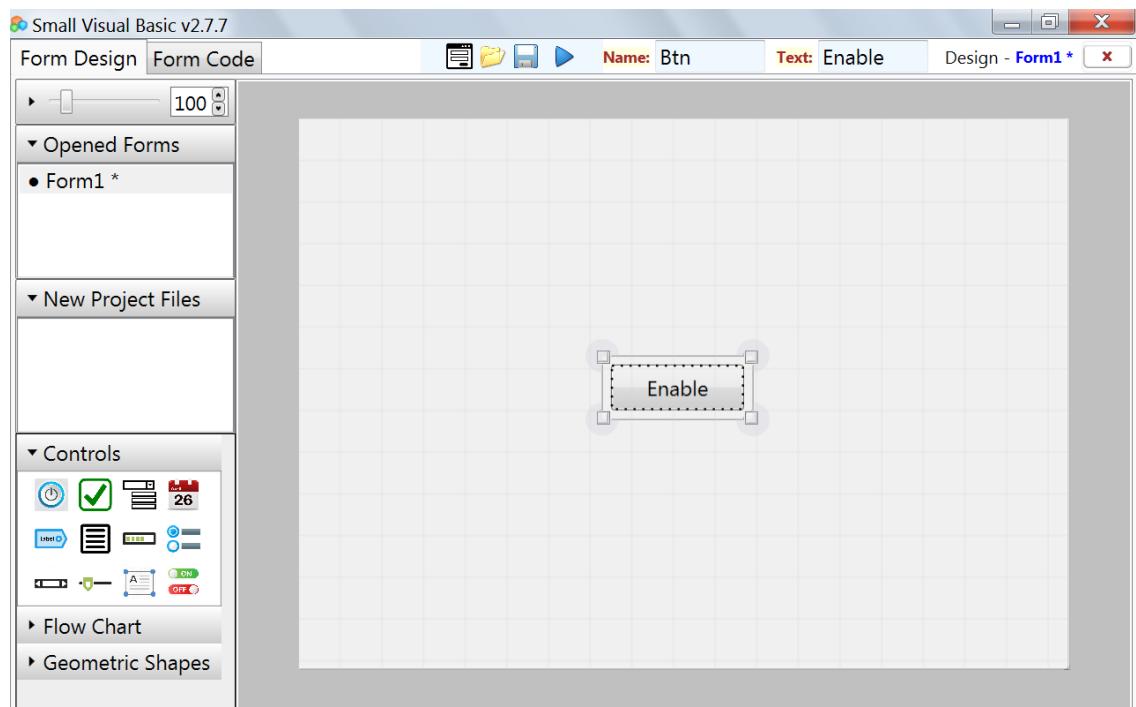
- more complex (too advanced to do nothing important with a language meant to be a learning toy),
- or simple enough to draw the controls and generate some code for them, but still can't overcome the SB syntax

limitations when dealing with objects.

This is why I asked my self: Why does SB have only two windows? Can't SB have a form designer and a simple syntax to program the controls?

I actually asked for this on the SB repo on GitHub, but got no attention from the team, so, I decided to prove the concept myself.

I added a form designer to SB, which I built using a tool called "Diagram Helper" I created back in 2014 to design flow charts, but found it can be easily modified to work as a form designer. I also added a small WinForms library to SB and wrote a pre-compiler to lower the object syntax to the normal syntax that the SB compiler understands. Using this simple trick, I allowed using control names as objects to make the code shorter and easier. So, I ended up with a **Visual Small Basic** which was too close to Visual Basic or in fact a small version of it, hence I decided to name it **Small Visual Basic**.



The screenshot shows the Small Visual Basic v2.7.7 interface. The title bar reads "Small Visual Basic v2.7.7". The menu bar has "Form Design" and "Form Code" tabs, with "Form Code" selected. The toolbar includes icons for New, Open, Save, Save As, Import, Publish, Cut, Copy, Paste, Undo, Redo, Run (F5), and Graduate. The main window displays the code for Form1.ssb:

```
1 Sub Btn_OnClick()
2     If Btn.Text = "Enable"
3         Btn.Text = "Disable"
4     Else
5         Btn.Text = "Enable"
6     EndIf
7 EndSub
```

I published the a sVB prerelease on GitHub in January 2021, and invited the SB team to use the idea to evolve SB, but they didn't seem interested!

For about 5 months, I had no intention to do anything further, but in June 2021, [my beloved grandmother Om Wahba](#) died, God rest her soul, so, I decided to complete sVB to honor her.

Also in that time, my nephew Mohammad Mohsen started to see my YouTube videos about SB, and I wanted to spare him the SB issues I explained above, so I worked hard to release the first sVB stable version in July 2021, which had only 3 controls in the toolbox!

I kept updating the language over the next two years, until it hit v2.8 with the release of this book.

That was the story of the three years journey from Small Basic to Small Visual Basic.

* It is also a Small Visual Basic .NET!

You can say that:

Small Visual Basic = Small Basic + Visual Basic:

but this is not the whole truth!

sVB is meant to be a middle stage in between SB and VB.NET, so, its syntax combines the best of SB, VB, and VB.NET languages!

It is also as easy as Windows Forms, but also has some advanced WPF graphical features!

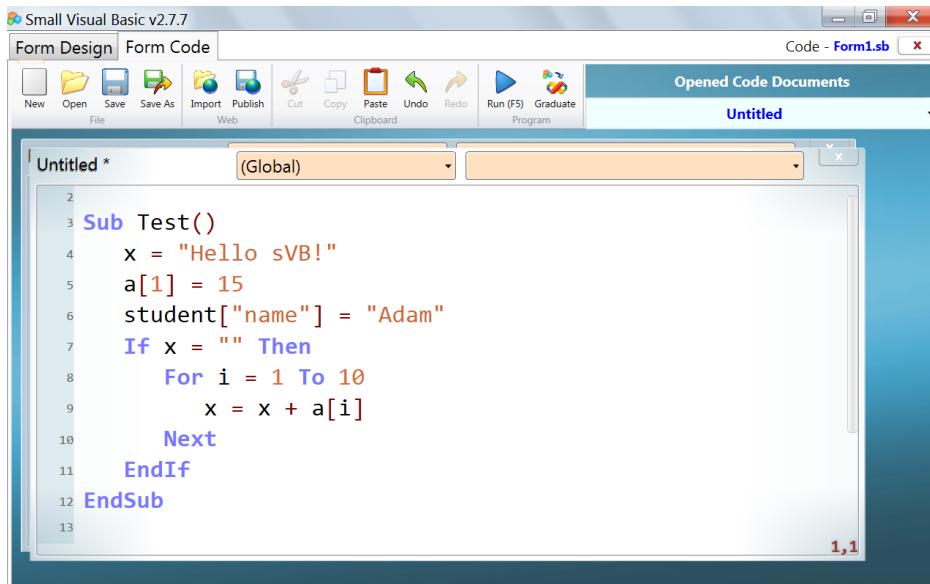
Let's look a little deeper into that:

* sVB and Small Basic:

sVB is built on top the Small Basic IDE, compiler and library.

In most cases, SB code works in sVB, but there are a few breaking changes that we will explore in next paragraphs. See also: [How to fix SB variable issues in sVB](#).

sVB also introduces some improvements to SB, such as:



1. SB uses the **EndFor** and **EndWhile** keywords, and they are still valid in sVB but it allows (and recommends) you to use

the **Next** and **Wend** keywords instead, to be more consistent with VB.

2. sVB added many methods to the SB types like Array, Text and File, and made changes to a few of the old methods like making the file methods return True or False instead of "SUCCESS" or "FAILED". You can learn more about this in [the sVB Library](#) part of this book.
3. SB allows to use spaces as a separator between arguments, but sVB doesn't and only allows using the comma.
4. There is a parseing error in SB that allows you not to add the closing quote of the string literal, but it is fixed in sVB.
5. There is a aprsing error in SB that prevents you from using double quotes to embed a wuote inside a string literal, but it is fixed in sVB.
6. sVB can't use the external libraries created for SB, unless they are rewritten for sVB to target the SmallVisualBasicLiberary.dll. So, SB programs that use such libraries will not work in sVB. The good news is that the kargest and most famous SB external library LitDev has created a version valid for sVB, and it is now installed with the sVB, so you can use it directly if you want.
7. sVB added the TextWindow.WriteLine method to allow you to directly output an array of items each at a line, which can be very handy when used with the sVB array initializer:

```
TextWindow.WriteLine({  
    "Line1",  
    "Line2",  
    "Line3"  
})
```

8. You can use TW as an alternative name for TextWindow.
9. You can also use these shortcuts for the TW methods, and the code editor will convert them to the full name with the

correct syntax as soon you leave the line:

Shortcut	method	example
?	TW.WriteLine	? "Hello"
	TW.WriteLine	? x, y
	TW.Read	x = ?
?:	TW.Write	?: "Enter x: "
#?	TW.ReadNumber	x = #?

For more details, see the samples provided for these methods in the [TextWindow](#).

10. sVB adds many enhancements to the Graphics Window. For example:

- a. You can use GW as an alternative shorter name for GraphicsWindow.
- b. You can use MsgBox as a shortcut name to show the message box with the default title "Message". Ex:
MsgBox "Hello!"

When you leave the line or save the file, the above code will be converted to:

Forms.ShowMessage("Hello!", "Message")

which is similar to:

GW.ShowMessage("Hello!", "Message")

but the Forms.ShowMessage doesn't enforce the GraphicsWindow to show, which makes it suitable to use within any form in the project.

You can also provide the title for the MsgBox shortcut syntax like this:

MsgBox "How are you?", "Hello"

Which will be converted to:

Forms.ShowMessage("How are you?", "Hello")

- c. In SB you can add only buttons and textBoxes on the graphics window, but in sVB you can add all the controls

you see in the toolbox of the form designer, by using the Controls.AddX methods like [Controls.AddCheckBox](#), [Controls.AddComboBox](#), [Controls.AddDatePicker](#), ... etc.

These methods return the keys of the added controls, where sVB can deal with them as objects of these control types, so it is easy to access their properties and methods and add handlers for their various events (See examples provided with each method).

- d. In SB, there is only one Timer objet, which complicates things if you need to take different actions in defferent intervals. In sVB this is not an issue, as you can add as many timers as you neeed to the Graphics Window using the [Controls.AddTimer](#) method.
- e. The GW got some new drawing methods like DrawArc, DrawBezierCurve, DrawQuadraticBezierCurve, DrawPolygon and FillPolygon methods. The Shapes library also got the AddPolygon method.
- f. You can create a composite shape using the [geometric path](#) and draw it on the graphics window by calling the [Shapes.AddGeometricPath](#) method.
- g. You can filll the graphics window background with a gradient brush by calling the [FillGradientBackground](#) method. You can also use the [GradientEndColor](#) property to create a gradient brush theat will be used to fill the shapes drawn on the graphics window using the GW.DrawX methods or Shapes.AddX methods.
- h. You can show the GW in full screen mode by setting the FullScreen property to True.
- i. You can show your graphics on the desktop directly by setting the [BackgroundColor](#) to Colors.None or Colors.Transparent.

- j. You can keep the graphics window on top of all other windows by setting the [Topmost](#) property to True.
 - k. Since the graphics window is in fact a normal form like any other form you add to your sVB project, you can use the [GraphicsWindow.AsForm](#) method to get the form object that represents it, so you can access more of its properties and methods.
 - l. The turtle can fill an area, by calling the [CreateFigure](#) and [FillFigure](#) methods. You can also change the turtle size by changing its Width or Height properties, increase its speed up to 50, and choose to disable the animation to get instant motion by setting the UseAnimation property to False
 - m. Some SB types got some love from the sVB compiler, like the Sound and Shape types. For example, when you assign the value returned by Sound.Load method to a variable, you can use it as an object of the Sound type, so you can access the sound methods directly from it. For an example, see the [Sound.Pause](#) method.
Likewise, the return value from the Shapes.AddX methods can be treated as objects of the Shape type. For an example, see the [Shapes.AddLine](#) method.
 - n. You can use the sVB [Thread library](#) to run subroutines in their own threads. This can be useful when you draw complex graphics on the GW specially when you draw using the SetPixel method.
11. The sVB code editor makes it easy to compose a multi-line string literal. All you need is to press Enter within the string literal, and the code editor will split it over two lines, and add the necessary code to embed the new line string. For example, type:

? "Line 1

then press Enter and type:

Line 2

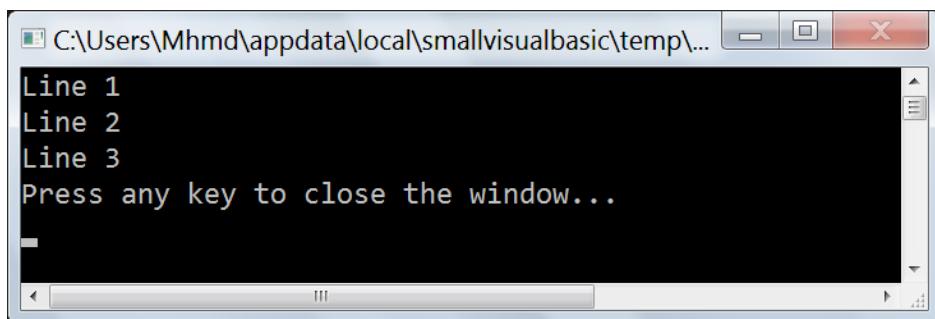
then press Enter and type:

Line 3

This is what you get on the code editor:

```
TW.WriteLine("Line 1" & Text.NewLine &
    "Line 2" & Text.NewLine &
    "Line 3")
```

And this is what you get when you run the code:



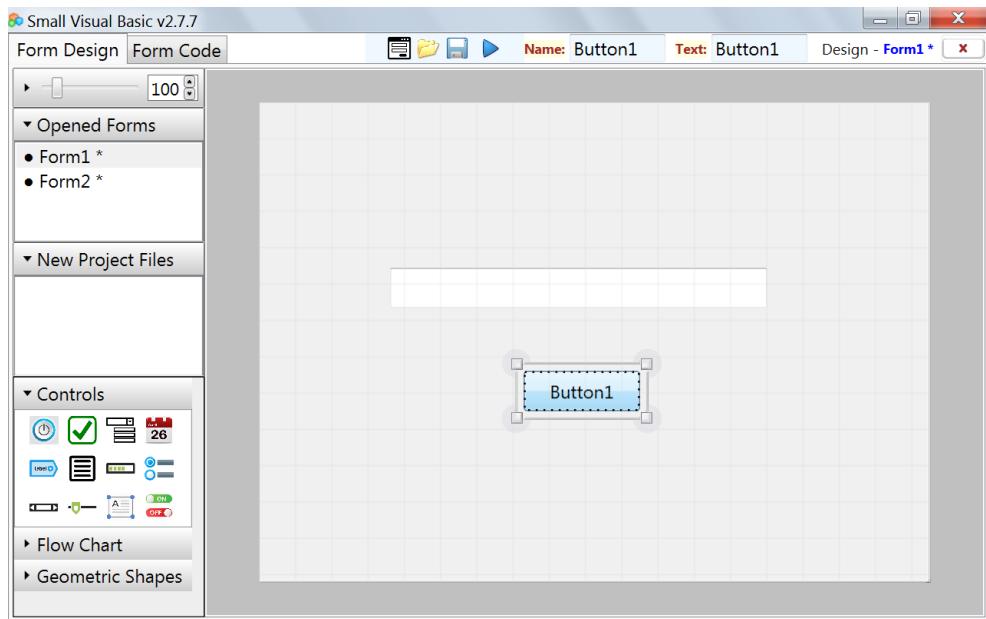
* **sVB and Visual Basic:**

sVB has many enhancements over SB to make writing apps fast and easy with a little code. It brings back the joy and excitement of using vb6 to write RAD applications.

To do that, sVB contains a small WinForms library and a form designer that allows you to add many forms to the project and only one global module.

You can drag controls from the toolbox, drop them on the form, and use the Properties Window to change some of its properties. You can use the Menu Designer to add a main menu to the form and design its submenus and their shortcuts.

You can double-click any control to get its default event handler created for you, where the ControlName_EventName naming convention is used to recognize the event handlers.



You can also add event handlers from the upper dropdown lists of the code editor, by choosing the control name from the left list (say "Button1"), and click the event name from the right list (say "OnClick"), so this sub will be added for you in the code editor:

```
Sub Button1_OnClick()
```

```
EndSub
```

```
1
2 Sub Button1_OnClick()
3     TextBox1.Text = "Hello sVB!"
4 EndSub
5
6
7
```

Like SB, sVB is a dynamic language, that uses the same syntax rules of Visual Basic syntax when you add these two statements at the top of the VB code file (this code is not used in sVB):

Option Strict Off

Option Explicit Off

sVB also has some other VB syntax features that are not supported by SB, like:

1. sVB has some OOP features like using control names as objects to access their properties and methods directly using the dot, like:

TextBox1.Text = "Hello sVB!"

TextBox1 is still a dynamic variable that contains the string key of the control (which is "form1.textBox1"), and you can read it, or change it to any value at any time (which of course is not recommended), but the editor auto-completion and intellisense make you feel it is an object, and the sVB compiler translates this object-like syntax to the normal SB syntax like:

TextBox.SetText(TextBox1, "Hello sVB!")

2. sVB allows you to use **Me** to refer the current form.
3. sVB allows you to use the line continuity symbol **_** to split a long command line at any position you want (except around the dot and !). For example:

**TextBox1.Text = "Hello sVB! " + _
"This is my first program."**

4. In SB, all variables are global, which means that the whole file has only one variable scope. sVB acts like VB, where variables defined in a subroutine are local to it, while variables defined at the file level (outside subroutines and functions) are global to the whole file. See [How to fix SB variable issues in sVB](#).
5. sVB allows you to declare Functions that return values.
6. sVB allows subroutines and functions to declare parameters.
7. sVB has a **ForEach** statement to loop through arrays, which is similar to VB **For Each**.
8. sVB uses **ExitLoop** to exit For and While loops, which is similar to **Exit For** and **Exit While** in VB.

But note that there are a few syntax differences between SB/sVB and VB, such as:

1. sVB uses **array[i]** to index the array instead of **array(i)** in VB.
2. sVB array is actually a dictionary, so you can use string keys instead of numeric indexes, and this is why array indexes don't need to be continuous nor ordered, and can even be negative such as **array[-1]**, because the compiler actually treats -1 as the string key "-1", so you actually deal with **array["-1"]**.
3. The two keyword statements in VB (like **For Each**, **End If**, **End While**, **End Sub**, and **End Function**) are combined into one word in sVB: (**ForEach**, **EndIf**, **EndWhile**, **EndSub** and **EndFunction**).

* sVB and Visual Basic .NET:

sVB in fact can be called sVB.NET, because it supports some VB.NET syntax features like:

1. sVB allows implicit line continuity before or after some symbols like comma, parentheses and arithmetic operators:

```
TextBox1.Text = "Hello sVB! "
    + "This is my first program."
```

2. sVB uses array initializers {} to add elements to the array:

```
a = {1, 2, 3, 4}
```

3. sVB supports date and time literals like:

```
d = #1/31/2023 13:00:00#
```

Note that the above literal uses the date format of the English culture.

Date literals are more flexible in sVB than in VB.NET, since sVB supports any valid English culture date format that can be parsed with the .NET Framework DateTime.Parse method, like **#31 Jan 2023#**.

sVB even has a time span literal like `#+3:15:00#` which is not supported by VB.NET!

4. sVB uses some famous .NET predefined enums like Colors and Keys with auto-completion support:

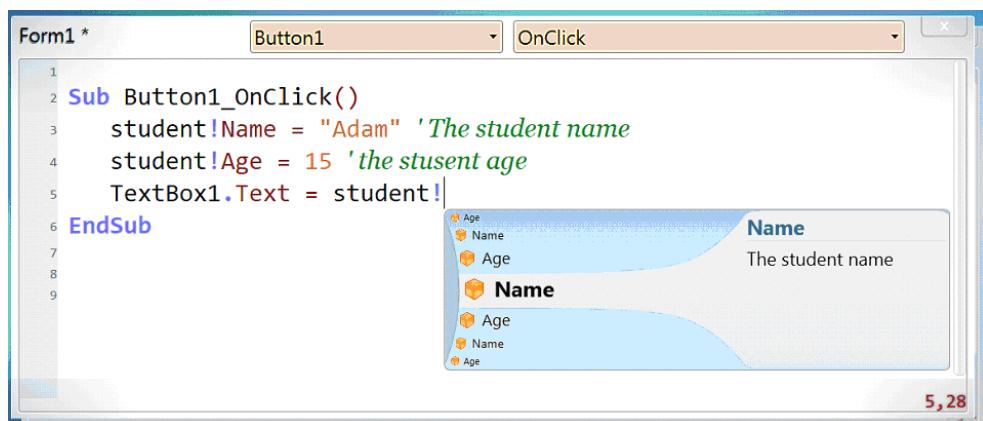
```
me.BackColor = Colors.AliceBlue
```

5. sVB uses **ContinueLoop** in For and While loops, which is similar to **Continue For** and **Continue While** in VB.NET.

6. sVB uses the dictionary lockup operator **!** to access array keys, which is similar to dictionaries in VB.NET:

```
student["ID"] = 1  
TextBox1.Text = student!ID
```

Also, this operator can be used to add dynamic properties to arrays, like the Expando Object in VB.NET, but sVB provides auto-completion dynamic properties, and you can even use comments to provide intellisense info about them:



7. sVB uses a VB.NET-like type inference, to detect the variable type from its initial value. This doesn't change the fact that sVB variables are dynamic types, and can hold any value of any type at any time, but inferring the initial variable type allows the compiler to treat it as an object, so you can access its properties and methods directly like you do in VB.NET. For example:

```
n = 2  
x = " Test "
```

```
TextBox1.AppendLine(n.Power(3))
```

```
TextBox1.AppendLine(x.Trim())
```

The above code is the short form of this code:

```
n = 2
```

```
x = " Test "
```

```
TextBox1.AppendLine(Math.Power(n, 3))
```

```
TextBox1.AppendLine(Text.Trim(x))
```

The above example may not be so important to you, but it can be when sVB infers the variable type from an expression or a method return value, specially when the method returns a form or a control, so that sVB can treat the variable as a control, which makes it easier to deal with its methods and properties directly:

```
list = Me.AddListBox("lstTest", 10, 10, 200, 300)  
list.AddItem({1, 2, 3, 4})
```

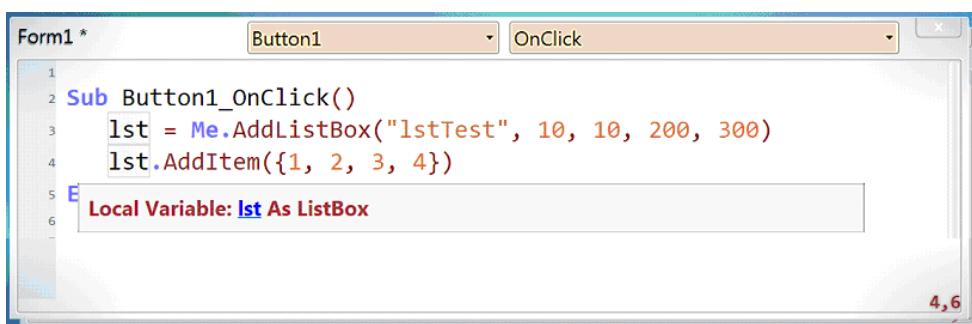
This somehow puts sVB in between of dynamically-typed and statically-typed languages!

8. The source code of sVB itself is written in VB.NET, and sVB projects are compiled to IL code, similar to VB.NET and C# projects! This means you can use any reflector to decompile any exe file created by sVB back to a C# project!
9. The sVB WinForms controls and their methods are so close to those of the VB.NET WinForms.
10. sVB has a [small UnitTest framework](#) that allows you to [write and run test functions](#) to test your project.
11. sVB can [create its own class libraries](#), in addition to the ability of [using VB.NET and C# to create libraries for sVB](#).

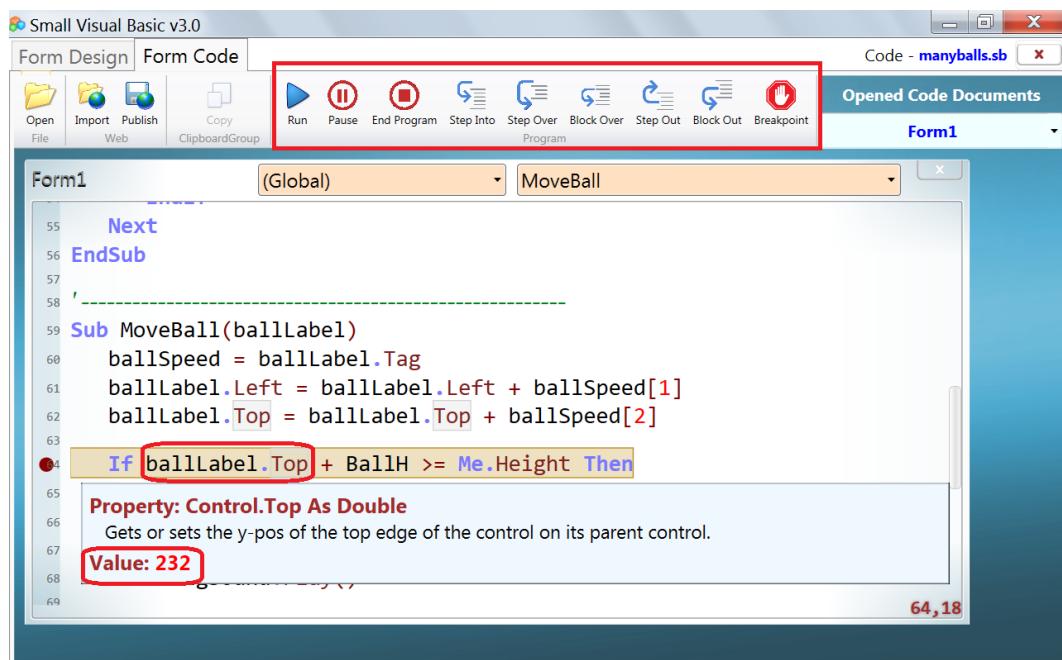
* sVB IDE and VS.NET IDE:

sVB editor is built on top of SB editor, with many enhancements that make it more like a small VS.NET editor, such as:

1. Code blocks auto completion, like adding **Then** and **EndIf** after writing **If**.
2. Identifiers highlighting and navigation using **Ctrl+Shift+Up** or **Ctrl+Shift+Down**.
3. Intellisense info about focused words with a link to go to definition.



4. A fully functional debugger to allow you trace the execution of your code at runtime.



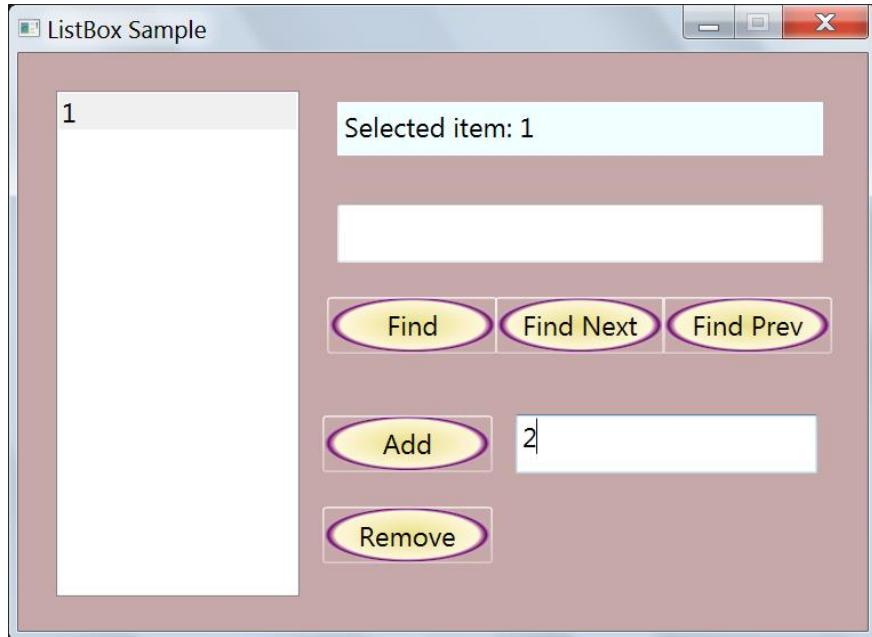
* sVB and WPF:

sVB is built using the Window Presentation Foundation (WPF) framework, to have advanced IDE features. For example, the form designer allows you to easily create advanced visual effects, like rotating and skewing controls, and the color dialog allows you to use solid brushes, linear and radial gradient brushes, and tile and image brushes to draw the control's border, background and foreground, which allows you to change the form and controls shapes.

For example, when you run the FormShape project from the samples folder, you will see this elliptical form, where its transparent edges really don't exist, so you can click any other windows through them!



Also, when you run the ListBox2 project from the samples folder, you will see these elliptical buttons:



Besides, you can use the [Control.SetResourceDictionary](#) and [Control.SetStyle](#) methods to load a resource dictionary from an external xaml file, to apply advanced styles on the form and any other controls you want to target!



* Fixing SB variable issues in sVB:

The most reason to break Small Basic codes in Small Visual Basic is variable domains. In SB, all variables are global to the whole file, which is a bad practice for kids and will make them suffer when move on to real programming languages. sVB has fixed this, so variables defined inside a subroutine are private to this subroutine and are invisible to other subroutines.

So, when you port any SB code to sVB, variables that sVB complains about should be initialized at the beginning of the file, by setting them to 0 for numbers, "" for strings, and {} for arrays. It is also recommended to convert variables that are used to send data to subroutines to parameters and convert variables that receive data from subroutines to function return values. For example, this SB code:

```
X = 10  
Y = 20  
Sum()  
TextWindow.WriteLine(z)
```

```
Sub Sum  
    z = X + Y  
EndSub
```

should be rewritten like this in sVB:

```
Z = Sum(10, 20)  
TextWindow.WriteLine(Z)
```

```
Function Sum(x, y)  
    Return x + y  
EndFunction
```

Or you can even get rid of z:

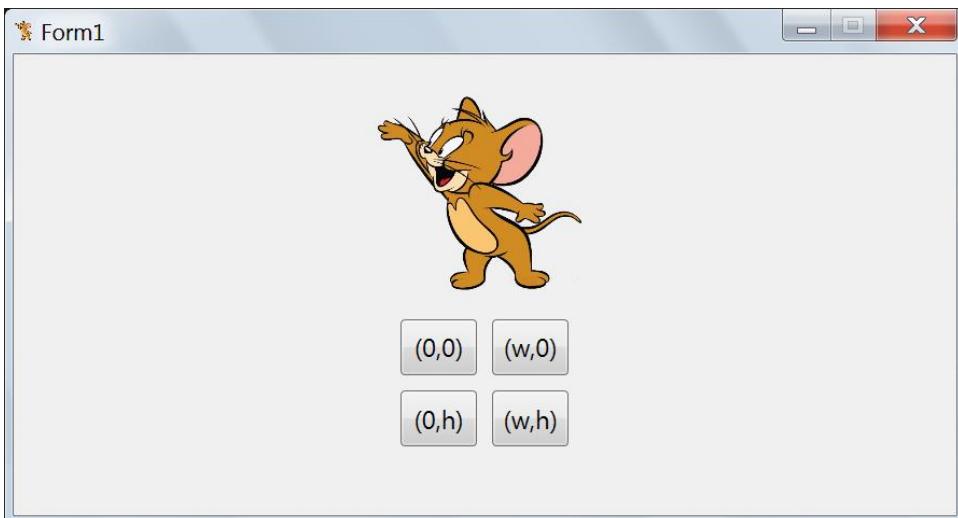
```
TextWindow.WriteLine(Sum(10, 20))
```

This is the correct way to organize variables in your program, which makes the code cleaner, shorter and more efficient.

* sVB for kids:

It is fair to wonder about how to introduce sVB to kids, while it seems very big compared to SB.

In fact sVB can be very easy for kids. All you need is to prepare some attractive images, use the [color dialog to display them](#) in labels, and use the [animation methods](#) to write some short and funny applications that interests small kids (see the animation and Jerry projects in the samples folder).



You can start with codeless apps, use the [form designer](#) to display some shapes and images, use the color dialog to [change their back colors](#). In the next step you can add some buttons to the app, each with a single code line that can animate the shape location, color or transparency.

In the next step you can introduce variables, but avoid If statements and loops until the kid gets comfortable with the language.

In fact I started writing the "Small Visual Basic Kid Programmer" book series to apply those guidelines. The Level 1 book of this series is already [published on Amazon](#).

I published [a course to teach sVB for kids on YouTube](#), but it is in Arabic. It contains more than 60 videos (10 minutes in average), and I think it is easy for developers and teachers to watch them even they don't understand Arabic, as the visual design and the little sVB code used, with the aid of this book, will be easy to comprehend, so, they can get the idea and build upon.

I encourage youtubers to introduce such a course in English and other languages, and they are free to reuse any ideas from my course without worrying about copyrights.

* **Conclusions:**

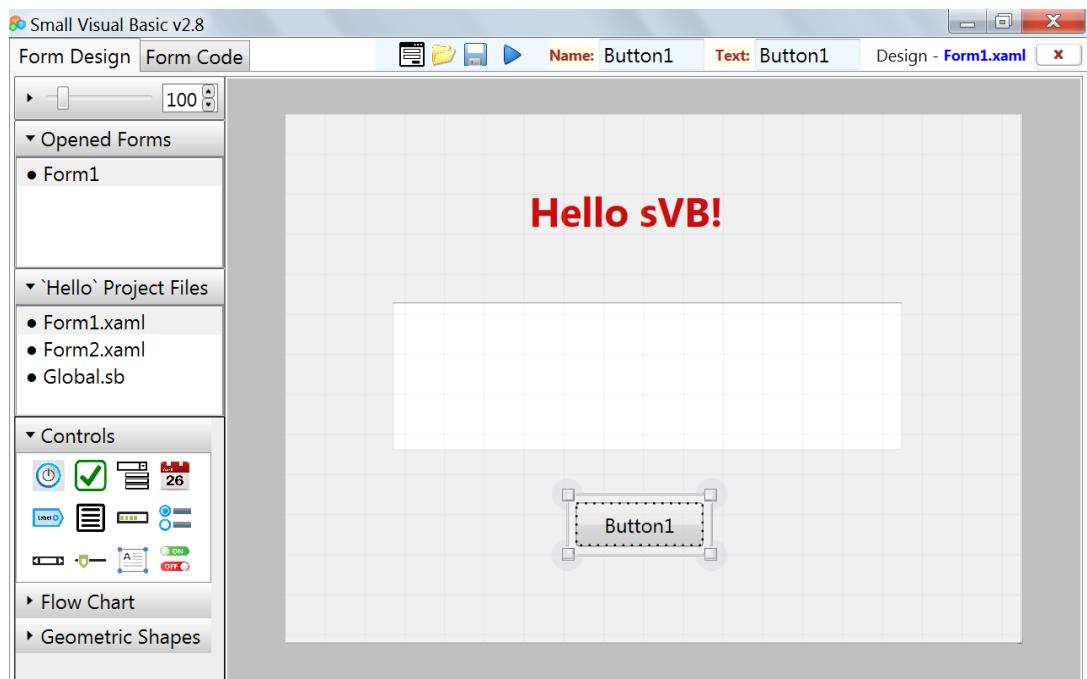
1. Small visual Basic is not really SMALL, as you can use it to create large projects containing tens of forms. The Small part of the name is just to reflect that it is built on Small Basic.
2. sVB combines the best of SB, VB, VB.NET, VS.NET IDE, and WinForms and WPF frameworks.
3. sVB is an easy lightweight version of the .NET platform, with only 20 MB installer to download, and minimum hardware requirements to run!
4. sVB purpose is to be attractive for kids and beginners to learn programming, then easily move on to the .NET platform.
5. By learning sVB, you are only one step forward to learn VB.NET.
6. sVB trains you on using the Windows Forms framework (WinForms), and it also opens a small window to take a look at some amazing WPF capabilities, so anytime later, you may find yourself interested in learning one of the XAML family technologies like WPF, MAUI or WinUI.

Small Visual Basic IDE

The sVB Integrated development Environment (IDE) is easy and simple, and looks like the VB6 IDE. It consists of two tabs:

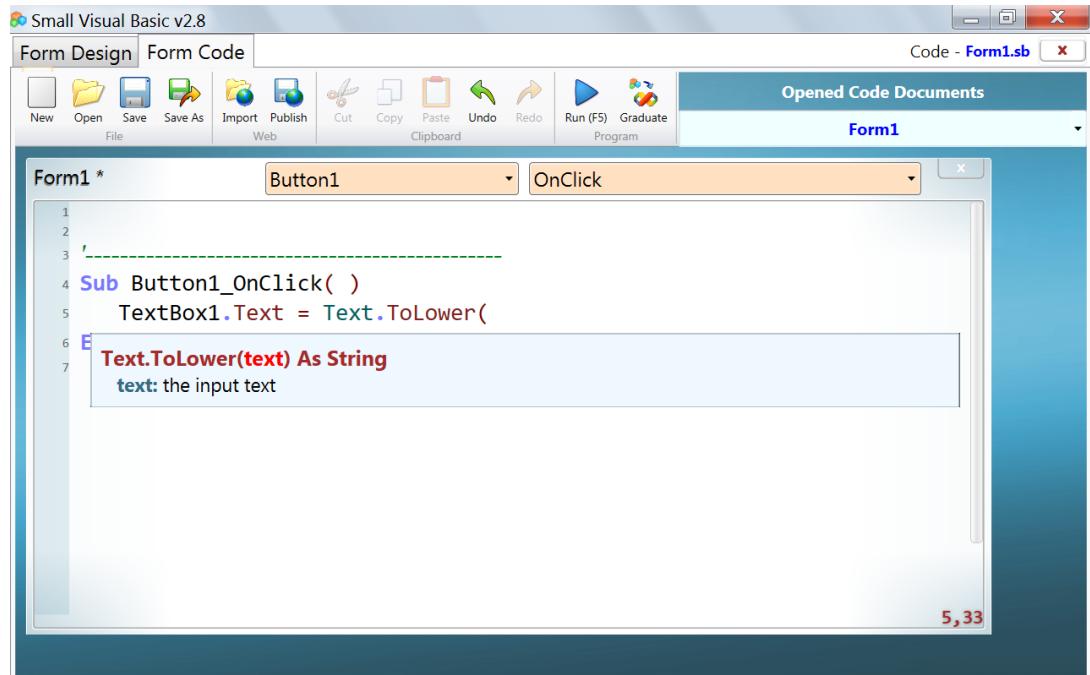
1. The Form design tab:

Switches you to the form designer, which is selected by default when you run sVB:



2. The Form code tab:

Switches you to the sVB code editor, which is built on the SB code editor with many enhancements:



Let's learn more about these two components.

The sVB Form Designer

The form designer consists of 4 parts:

1. The upper toolbar:

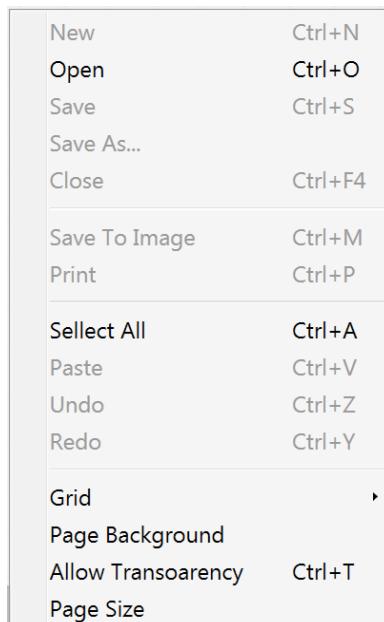
It shows the 6 buttons that you see in the following picture:



These buttons do the following tasks respectively:

- a. add a new form.
- b. open a saved form.
- c. save the current form.
- d. show the properties window for the form or the selected control(s).
- e. run the project.
- f. debug the project.

Those commands (and more) appear in the context menu that pops up when you right-click the form designer surface:



The upper toolbar also shows two text boxes to allow you to change the name and title of the form or the currently selected control on the form surface:



It also shows the name of the currently opened form, and a button to close it.



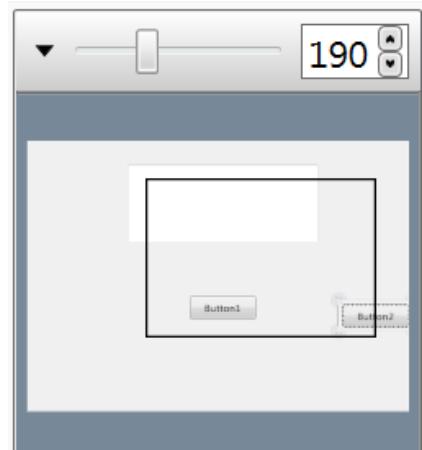
2. The side panel:

It shows many collapsible tabs that you can click to collapse or expand. These tabs are:

- **The zoom tab:**

It shows a slider that decreases and increases the zoom ratio of the form designer canvas. This ratio is displayed in the adjacent numeric textbox, where you can also change it manually or by clicking the up and down arrow buttons.

Note that the zoom tap is collapsed, and you can click it down arrow to expand, to show its zoom preview screen, which shows you the whole view of the form and its controls, an a view port rectangle that shows you the currently visible part of the form on the designer canvas. You can drag this rectangle to change the view port, which allows you to show any part of the form in an easy way.



- **The opened forms tab:**

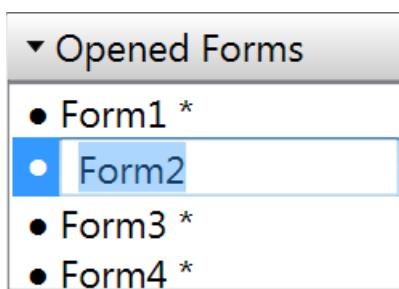
Shows a list of the names of the currently opened forms. The selected item in the list is the form that is currently viewed in the design canvas. You can close the current form by any of these ways:

- Clicking the close button on the upper toolbar.
- Clicking the Close command from the context menu.
- Pressing the Ctrl+F4 shortcut from the keyboard.
- Selecting the form name in the opened forms list and pressing the delete key from the keyboard.

If the form is not saved, or has any unsaved changes, sVB will ask you to save them.

Note that closing a saved form doesn't remove its files from the project folder. It just unloads it from the form designer, and you can open it again whenever you need.

Note also that other forms in the list are still opened in memory but just not active and don't appear on the design canvas at the moment, so they may contain changes that are not saved yet, and you will be asked to save each of them when you close sVB.



The name of the form is its programmable name that you use in code, and it can differ than the name of the saved file of that contains the form data. You can change the form name by selecting the canvas surface (outside any control drawn on it) then write the new form name in the

Name textbox on the upper toolbar. You can also select the form name in the opened forms list, and click it once or press F2 to switch to the editing mode, so you can write the new name and press enter or select another item or other part of the designer to commit the changes and end the editing mode.

The form name should be a valid identifier name and should be a unique all over the project, so no other form in the project folder can have the same name, otherwise you will get an error message, and the new name will be rejected, and you will not be able to end the editing mode unless you enter a valid name or press Escape to cancel the process and go back to the old name.

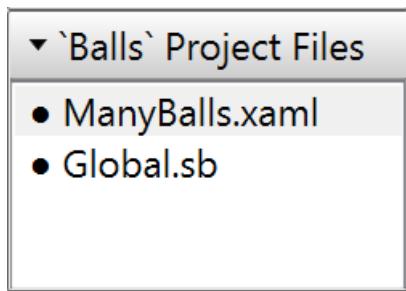
The opened forms list can contain saved and unsaved forms, where an asterisk * suffix will appear at the end of the unsaved form names. You can add a new form to the list by clicking the Add New Form button from the upper toolbar or by pressing Ctrl+N.

You can click the Open existing Form or press the Ctrl+O to open an existing form, and its name will appear in the opened forms list. This means that you can open forms from different projects at the same time, and they will appear together in the opened forms list, which means that it can contain more than one form with the same name (like Form1) because they belong to different folders (projects)! This allows you to switch from one project to another, by just selecting one form of the later project, where a list of all xaml files in this project folder will be shown in the project files tab. But if you select an unsaved form is in the opened forms list, the project files list will be empty.

Finally, you can switch to the code file of any form by double-clicking it in the opened forms list, or selecting it and pressing Enter.

- **The project files tab:**

Shows a list of xaml file names that exists in the folder of the currently opened form. If the form is not saved yet, this list will be empty.



When you select an item from this list, the corresponding form will be opened in the design canvas and its name will appear in the opened forms list. You can also double-click any item, or select it and press Enter, to open the code file of the form in the code editor.

To change a file name, you can click the selected file once or press F2 to switch to the editing mode, so you can write a new file name, then press enter or select another item or other part of the designer to commit the changes and end the editing mode. You should write the form name only without the extension, and you should use a unique name for the file, not to conflict with other file names in the folder. If the name is accepted, sVB will use it to rename the three files of the form (the .xaml, .sb, and .sb.gen files), otherwise, you will get an error message, and the new name will be rejected, and you will not be able to end the

editing mode unless you enter a valid name or press Escape to cancel the process and go back to the old name. Note that you should always use the project files list to rename the files, and never rename them manually from the folder, because sVB uses the form name in the generated code written in the form.sb.gen file, and it will regenerate this file when you change the form file name from the project files list, which will not happen when you do this manually, unless you are paying attention and remembered to modify the form.sb.gen file manually. You can also press delete to remove the selected item in the project files list from the project. This will remove all the three files of the form from the project folder to the recycle bin, which is a dangerous operation, so sVB will show you a message box to confirm that this is really what you want.

The project files will also show the Global.sb item at the end of the list. This is the only single sb file allowed in the project and it has no form. You can't modify the name of this item but you can delete it to remove it to the recycle bin, but when you reopen the project or switch back to one of its forms in the opened forms list, you will see the Global.sb item again. In fact this item appears even if the Global.sb file doesn't exist in the project folder, so you can double-click it to create the file and open it in the code editor.

- **The controls tab:**

This is the sVB basic controls toolbox. It contains 12 controls:



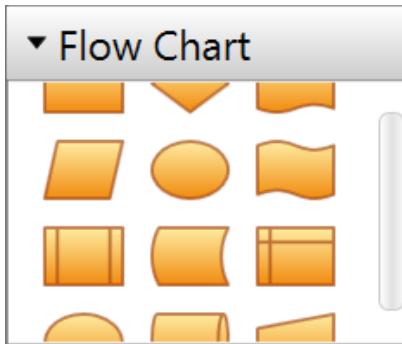
1. [The Button Control.](#)
2. [The CheckBox Control.](#)
3. [The ComboBox Control.](#)
4. [The DatePicker Control.](#)
5. [The Label Control.](#)
6. [The ListBox Control.](#)
7. [The ProgressBar Control.](#)
8. [The RadioButton Control.](#)
9. [The ScrollBar Control.](#)
10. [The Slider Control.](#)
11. [The TextBox Control.](#)
12. [The ToggleButton Control.](#)

You can draw these controls on the design canvas as explained in the [form surface canvas](#) section.

Note that there are 3 more controls that can be added to the form at runtime but are not supported by the designer yet, which are the [MainMenu](#), the [MenuItem](#), and the [WinTimer](#) controls.

- **The flow chart tab:**

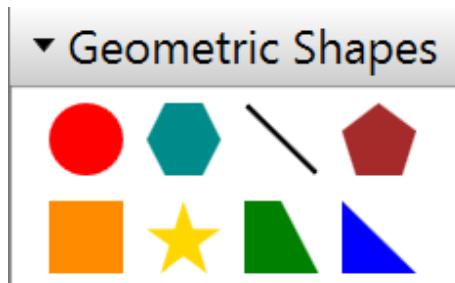
This is the flow chart toolbox. When you add any of these shapes on the form, it will be drawn on a label, so, you will deal with it programmatically as [a Label control!](#)



You can draw these items on the design canvas as explained in the [form surface canvas](#) section.

- **The geometric shapes tab:**

This is the geometric shapes toolbox. When you add any of these shapes on the form, it will be drawn on a label, so, you will deal with it programmatically as [a Label control!](#)



You can draw these shapes on the design canvas as explained in the [form surface canvas](#) section.

Remember that the form designer allows you to resize, rotate and skew any control, which allows you to reshape those shapes to get more geometric figures, like ellipse, rectangle, diamond, etc.

Modify the Toolboxes:

The three toolboxes explained above are created from the data existed in the sVB\Bin\Toolbox folder. Basically, this folder contains the three folders (Controls, Flow Chart and Geometric Shapes) where their names are used as the tap names in sVB.

The controls folder contains a xaml file and an image file for each of the 12 controls you see in the Controls toolbox. So, you can change the controls icons, and it is possible to change the default property values of any control to be used when it is drawn on the form, but you should be familiar with the xaml language and the WPF controls before doing that, and have a backup copy of the toolbox folder so you can restore it if you damaged something!

You can also add new controls to this folder, but they will not be useful without creating a class for each control in the sVB library so you can deal with its properties and methods from sVB code.

The Flow Chart and Geometric Shapes show you two ways to create vector graphics shapes to be added to the toolbox. You can add more shapes to the geometric shapes toolbox if you want.

You can also **add new folders** in the toolbox folder and add shapes to them as you want. It will be nice if you add some vector graphics for famous cartoon figures like Tom and Jerry to create a Kids toolbox. This doesn't need any change in sVB library, as all shapes other than controls are drawn on the form inside labels, and you will just code them as labels.

3. The form-surface canvas:

This is the working area of the form, where you can draw controls and adjust their appearance. What you see on this canvas is what you will see on the form when you run the program.

*** The active form:**

sVB starts with an empty form, which will be closed silently if it has no changes and you opened an existing project.

You can use the toolbar, the context menu or the keyboard shortcut keys to add new forms or open saved ones. The form designer views only one active form, but opened forms still loaded in memory preserving their last states until you bring them back into view, by selecting their names from the Opened Forms or the project files tabs.

You can close any form by clicking the x button on the upper toolbar, clicking the Close command from the context menu, or selecting the form name in the opened forms list and pressing delete. If there any changes in the form, sVB will ask you to save it.

After closing the active form, the previously displayed form (if any) will be the active form and the design canvas will show it.

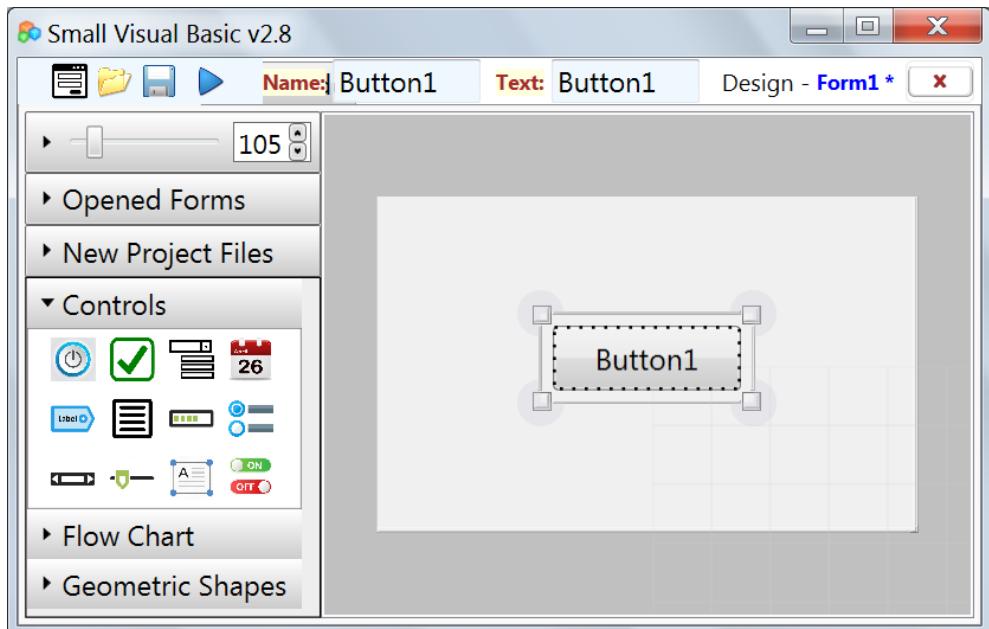
Not that there will always be at least one form opened in sVB, so, you can't close the unsaved unchanged form, and when you close the last saved or changed form, sVB will open a new empty form!

*** Drawing controls on the form:**

The controls tabs contains 12 important controls that you can draw on the form surface canvas in design time. Other toolbox tabs like the flow chart tab and the geometric shapes

tab contain different vector graphics shapes, that you can also draw on the form, where sVB will wrap each of them inside a label control, so they are actually just labels. You can add as many as toolbox tabs as you need and populate them with any shapes you want as explained earlier.

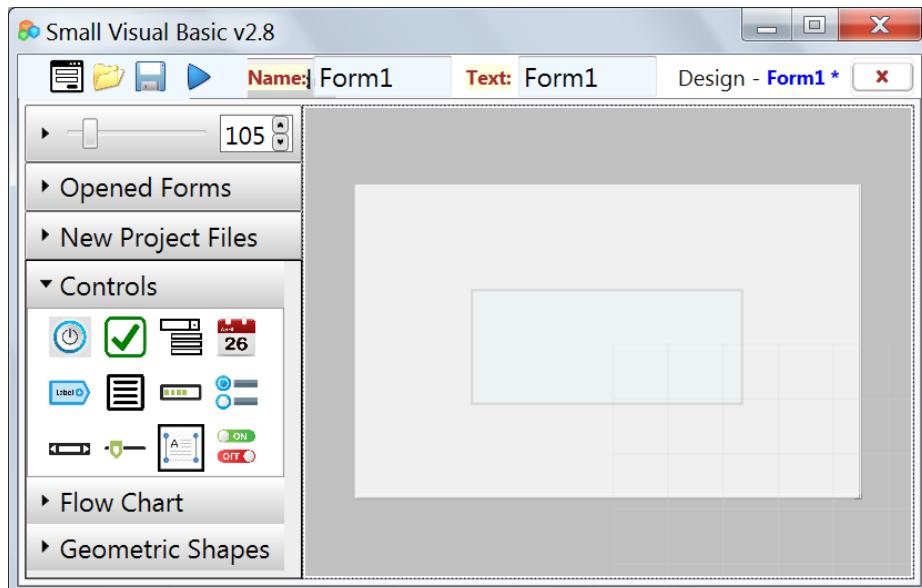
To add a control from any toolbox to the design canvas, just left-click its icon and keep the left mouse button down while you drag it to the canvas. Keep moving until you reach the position you want to draw the control at, then release the mouse button to drop it. The left top corner of the control will be at the mouse pointer position, and it will be drawn with its default width and height (as defined in its xaml file in the toolbox folder).



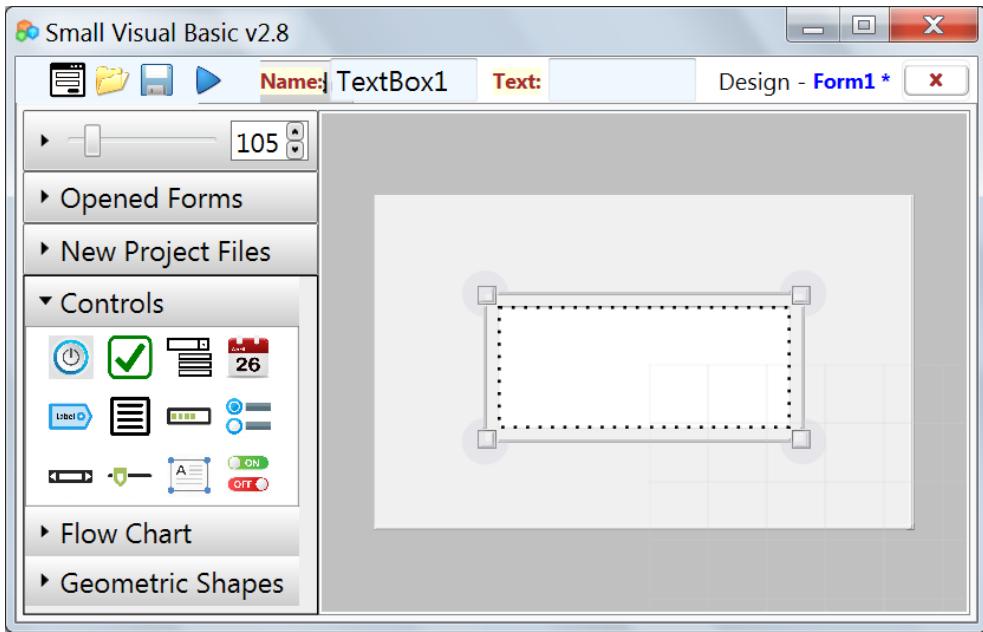
There is also another way to draw a control, by left-clicking its icon in the toolbox but instead on dragging it, you can release the mouse button to select this icon where a selection border will appear around it.

The icons in each toolbox work as radio buttons, where you can select only one icon at a time, but they also work as toggle buttons, as you can select the icon by one click, then deselect it by another click.

When an icon is selected in the toolbox, the mouse pointer turns into an pen cursor over the design canvas, which mean you can draw a rectangle to drop the controls inside. To do that, left-click the canvas at any point and move the mouse around. This will draw a selection rectangle that is resized while the mouse moves. The following picture shows you the selection rectangle being drawn on the canvas while the TextBox icon is selected in the Controls toolbox:



Now you can release the mouse button to draw the control to fill this rectangle. This way is better than the first one in that you control the width and the height of the control, not just its position. But you shouldn't worry, as you can easily modify the control position and size at ant time, as we will explain in the next section.



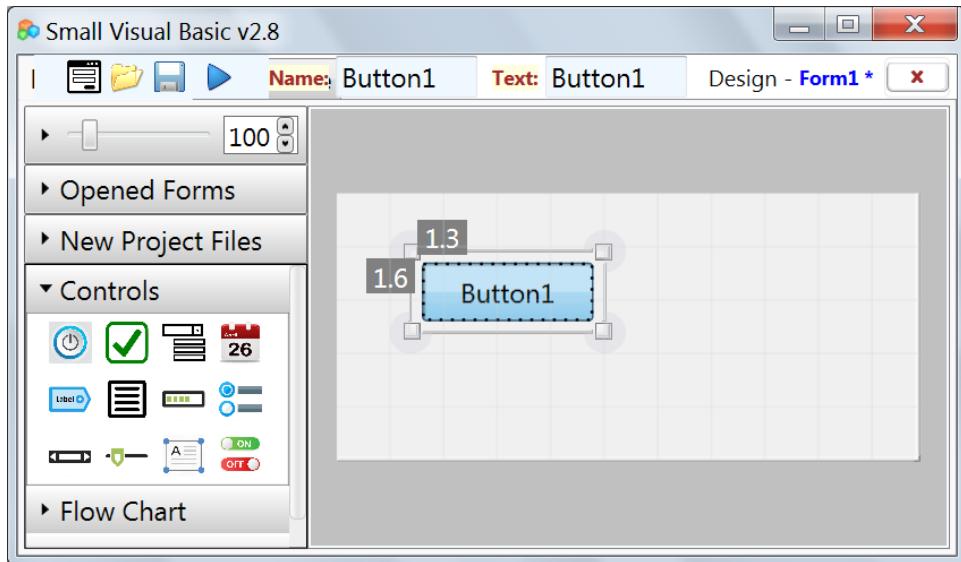
Note that you can cancel the selection rectangle before drawing the control by pressing the Escape key from the keyboard. Either way, the selected icon in the toolbox will be deselected.

Finally, you can delete the control by selecting it by a mouse left-click, then pressing the delete button from the keyboard.

* **Moving controls on the form:**

After you draw controls on the form you can drag them by mouse (left-click and move) to drop them anywhere you want. While dragging, the control will display two labels that show you the top and left positions in centimeters.

The design canvas also displays horizontal and vertical grey grid lines which divide the form surface to 1cm x 1cm squares, which helps you to align controls.



You can also click the control to select it, then press one of the arrow keys from the keyboard to adjust its position:

- Left arrow: decreases the left position by 0.1cm.
- Right arrow: increases the left position by 0.1cm.
- Up arrow: decreases the top position by 0.1cm.
- Down arrow: increases the top position by 0.1cm.

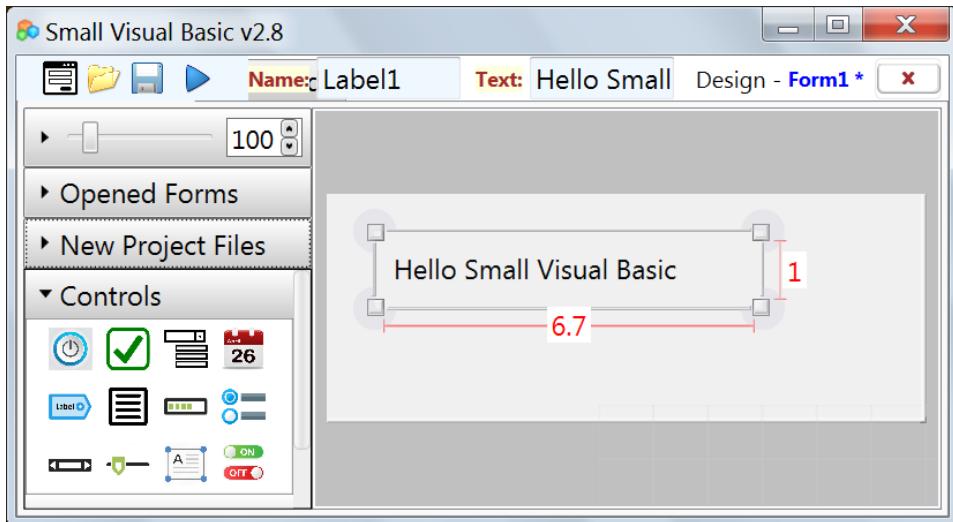
You can also press the Control key with the above keys to move the control by 1cm not just 0.1cm.

You can also use the page up and page down keys:

- Ctrl+ Page up: moves the left position of the control to the left edge of the form.
- Ctrl+ Page down: moves the right position of the control to the right edge of the form.
- Page up: moves the top position of the control to the top edge of the form.
- Page down: moves the bottom position of the control to the bottom edge of the form.

* Resizing controls:

When you click any control, it becomes the active (selected) control on the form, and a selection border appears around it. This border has 8 resizing thumbs, four of them are the rectangle sides, and the other four are the small square thumbs at the rectangle corners.



When you hover by mouse over any of these resizing thumbs, the mouse cursor turns into a bi-directional arrow, that points to the two resizing directions of this thumb (back and forth). You can left-click the thumb and drag it along these directions to resize the control, which will display two labels that show the control's width and height in centimeters. You can release the mouse when the control has the size you prefer.

Note that you can double-click any of the horizontal resizing thumbs to make the control width equal its height. Likewise, you can double-click any of the vertical resizing thumbs to make the control height equal its width. This is the fast precise way to get a square control.

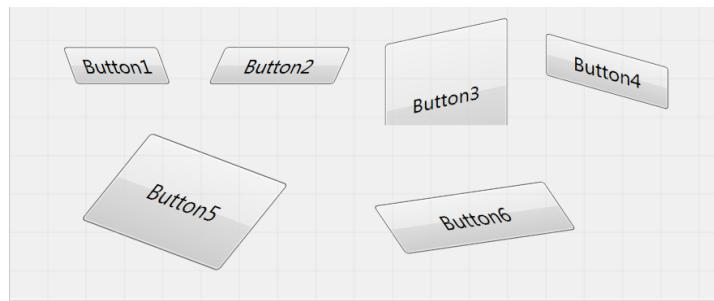
* Resizing the form:

Use the designer resize thumbs to change the form size.

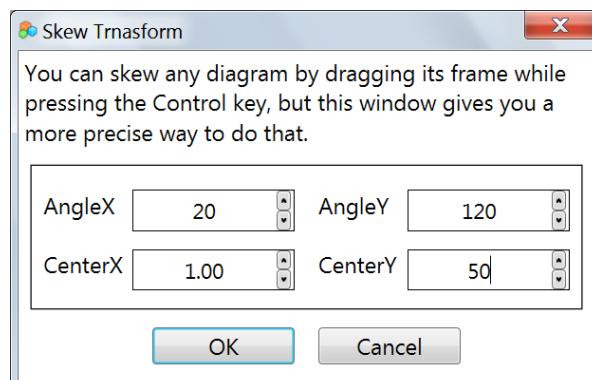
There are 3 thumbs, two at the right and bottom sides and one at the right-bottom corner, so you can drag the canvas at these locations to increase or decrease its size.

* Skewing controls:

Skew a control means pulling its horizontal or vertical edges by a positive or a negative angle.

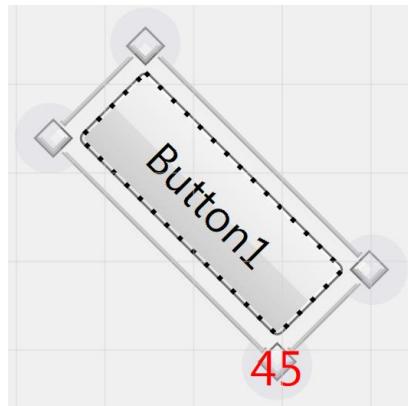


You can skew a control around its center by selecting it and dragging one of its resize thumbs while pressing the Control key. You can also use the skew window to have a better control of the skewing center and angles. To do that, right-click the control to show its context menu, expand the "Rotation & Skew" submenu, and click the Skew command, or just simply press Ctrl+K from the keyboard. This will show the skew transform window, where you can enter the horizontal and vertical values of the skew center and angles.



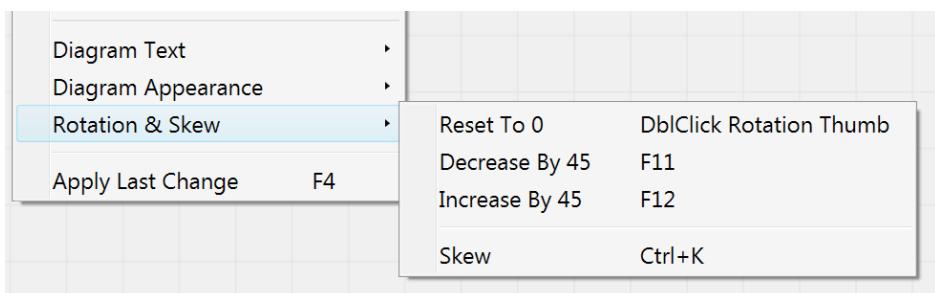
* Rotating controls:

You can rotate any control around its center. To do that, select the control to show its selection border. You can notice a gray circle around each corner of the selection border. These are the rotation thumbs, and you can drag them around the control to change its rotation angle, which increase from 0 to 359 in the clock direction. While dragging it, the rotation thumb will display the angle in degrees.



Note that you can double-click any rotation thumb to reset the rotation angle to zero, which is the fast precise way to cancel any applied rotation.

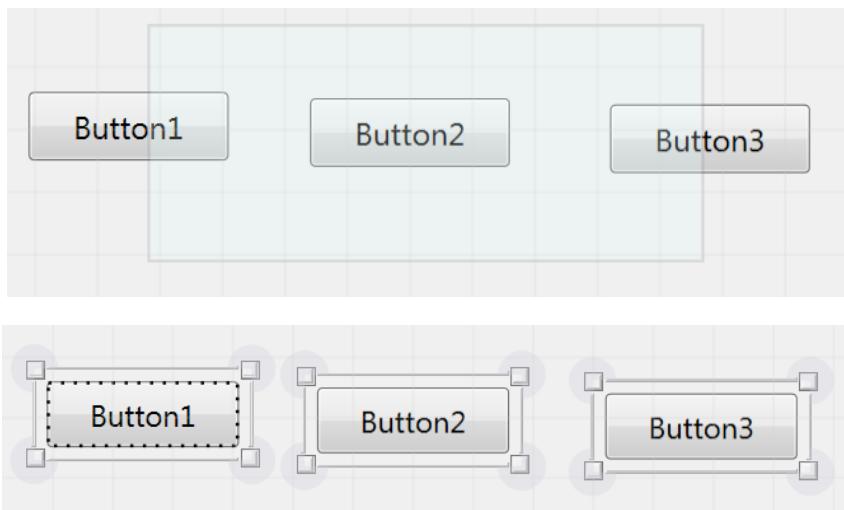
You can also right-click the control to show its context menu, then expand the "Rotation & Skew" submenu, which gives you some useful rotation commands to reset the rotation angle to zero or increase or decrease it by 45 degrees, which you can do directly by double clicking the rotation thumb and pressing F11 and F12 respectively.



* Selecting and grouping controls:

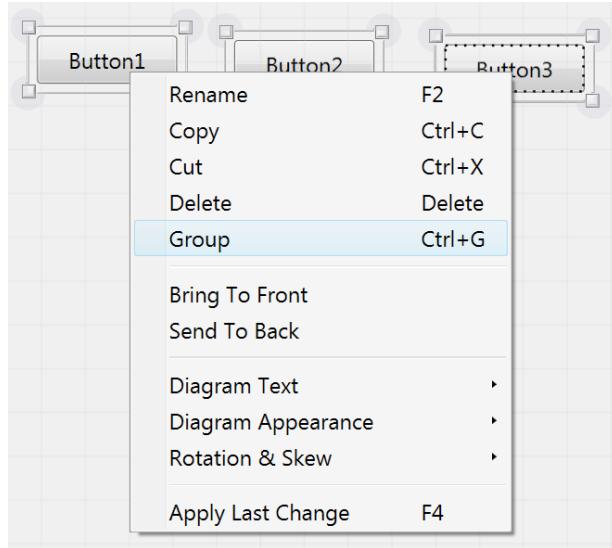
You can select one control by a single mouse click, and you can move to the next control by pressing the Tab key, or to the previous control by pressing Ctrl+Tab.

You can also select multiple controls by a left-click on the design canvas (out of any control), and moving the mouse to draw a selection rectangle. Any control is fully or partially contained in this rectangle will be selected when you release the mouse button.

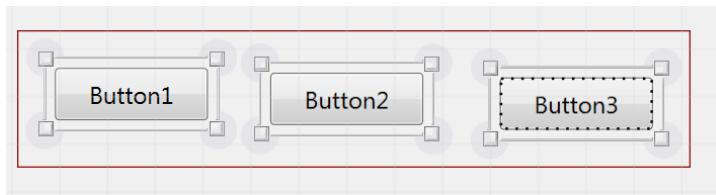


The selected controls will be affected by the editing commands, For example, you can press Delete to remove them all, Ctrl+C to copy them to the clipboard, or Ctrl+X to cut them. You can also change their position on the form by dragging any of them.

You can also group the selected controls together by pressing Ctrl+G, or right-clicking one of them and clicking the Group command.



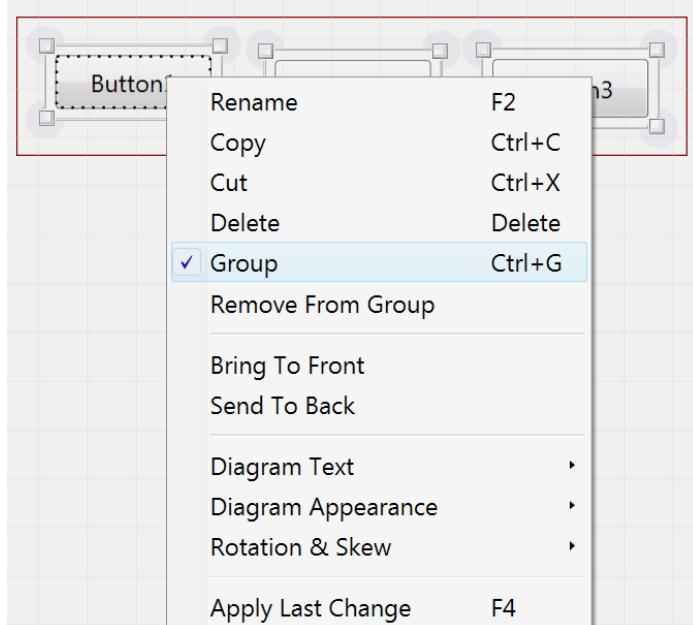
A grouping rectangle will be shown around the grouped controls, which will be hidden when you deselect them.



When you select back any of these controls, all of them will be selected and the grouping rectangle appear again. This allow you to keep these controls together to maintain their relative positions on the form while you are dragging them. But if you want to move a single control inside the group, press the Control key whey you are dragging it, to relocate it alone. This will change the grouping rectangle area to keep all of controls contained within.

You can also take a control out of the group, by right-clicking it and clicking the "Remove From Group" command from the context menu. If you want to ungroup the controls, right-click any of them to show the context menu. You will notice that the Group command is checked. Click it again to

uncheck it and free the controls, or just press the Ctrl+G to do it.



For all controls, grouping is useful only in design time, except for the radio button controls, because grouping them changes their [GroupName property](#), which affects how they work in runtime.

* **Changing form and control names:**

Each form in the project must have a unique name. Also each control must have a unique name within the form that displays it, so you can't have two buttons with the same name "button1" both drawn on the same form, but it is OK if they belong to two different forms, because sVB defines a variable for the form and its controls in the form.sb.gen file. For example, this is the how generated code looks like for a form named "form1" with a button named "button1":

```
Form1 = "form1"  
Me = "form1"  
Button1 = "form1.button1"
```

And this is the how generated code looks like for a form named "form2" with a button named "button1":

```
Form2 = "form2"  
Me = "form2"  
Button1 = "form2.button1"
```

Note that each variable defined in the form code is private to the form, so sVB doesn't have any problem to have a variable named Button1 in both forms. What really matters is the value that each of them holds, because it is the button key in the controls collection. You can see that the generated key is unique for each button because it is qualified with the form name that it belongs to, hence you can say they have the same name "button1" but they have two different keys: "form1.button1" and "form2.button1".

In fact you don't need to worry about all of that, because sVB does the job for you. All you need to worry about is to choose a unique name for each form and control, which should also follow [the rules of a valid identifier](#) so that sVB can define a valid variable with this name that you can use in your code.

But, where to set this name?

When you click the form or any control to select it, you can notice that its name appears in the Name textbox that is displayed on the upper toolbar, so you can simply change this name and press Enter or move out of this textbox to commit the new name. sVB will check the name, and will refuse it if it is not valid, preventing you from leaving the textbox unless you enter a valid name or press Escape to cancel the change and keep the old name.

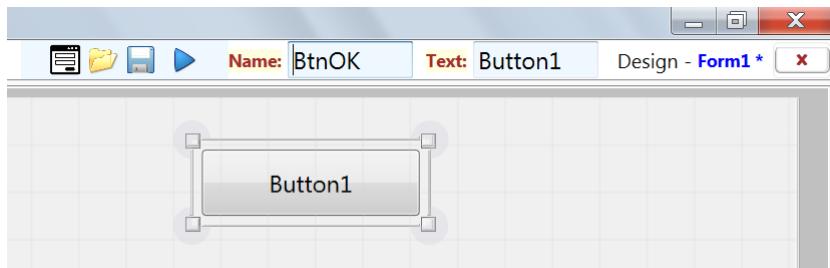
Note that you are totally free to give the control any name you want as long as it is valid, but as a good practice, you should follow an extra rule in naming your controls, to make

you code easy to understand and maintain, which is to choose a name that tells you what job it performs. For example, "Cancel" and "OK" are good names for the cancel and OK buttons.

It is also recommended that you add a prefix to the control name that refers to its type. It is popular to use a 3 letters as a control prefix like Frm, Btn, Lbl, Txt, Lst, Cmb, Rdo, Chk ... etc. For example: "FrmMain", "BtnOK", "TxtUserName", "LblResult", ... etc.

Others may prefer to add the whole control name as a suffix, like " MainForm", "OKButton", "UserNameTextBox", "ResultLabel", ... etc.

You are free to use the pattern you prefer, or mix them together as you want.



* **Changing form and control titles:**

The form can show a title on its title bar, and some controls can also display a title like Label, Button, CheckBox, ComboBox, RadioButton, and ToggleButton. Besides, the TextBox can input and output text.

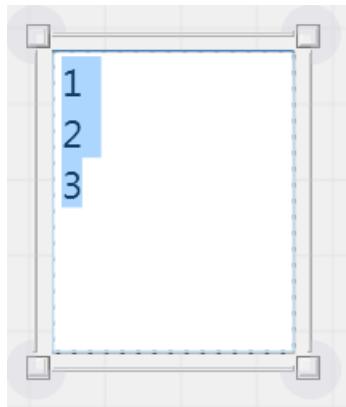
All those controls have a Text property that you can use to set or get the text they display. In addition, you can set their Text property in design time, by selecting a control on the form designer and using one of the following two ways:

1. Use the Text textbox displayed on the upper toolbar to change the title.

2. Right-click the control to show the context menu, expand the "Diagram Text" and click the "Edit Text" command submenu, or just select the control and press Enter from the keyboard. This will show an editing textbox over the control surface, where you can enter the text you want. You can press Ctrl+Enter to add a new line in this editing textbox, which you can't do in the first way.

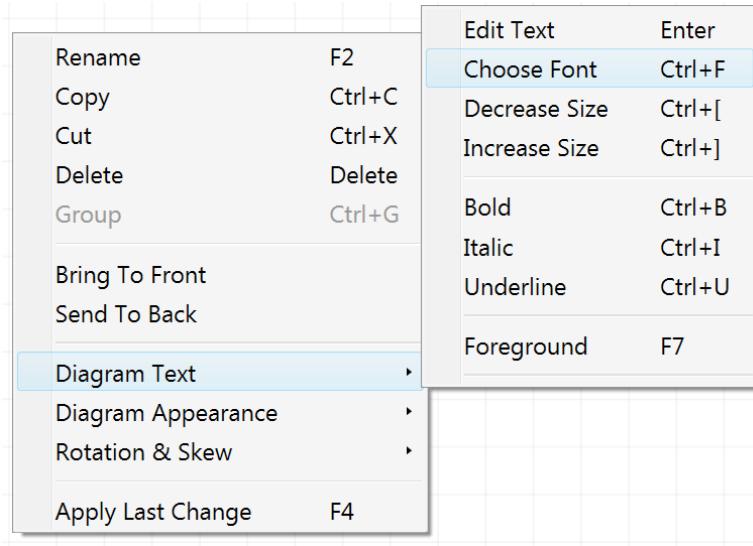
In both ways, you can press Enter to commit the change, or Escape to cancel it and keep the current title.

Note that changing the title of a control that doesn't display a text (like the ProgressBar) will have no effect, but to make things easier when you design the ListBox and the ComboBox, the sVB designer allows you to enter the list items as a text property, each item at a line:



* **Changing form and control Fonts:**

When you right-click the control to show its context menu, you can see the "Diagram Text" submenu:



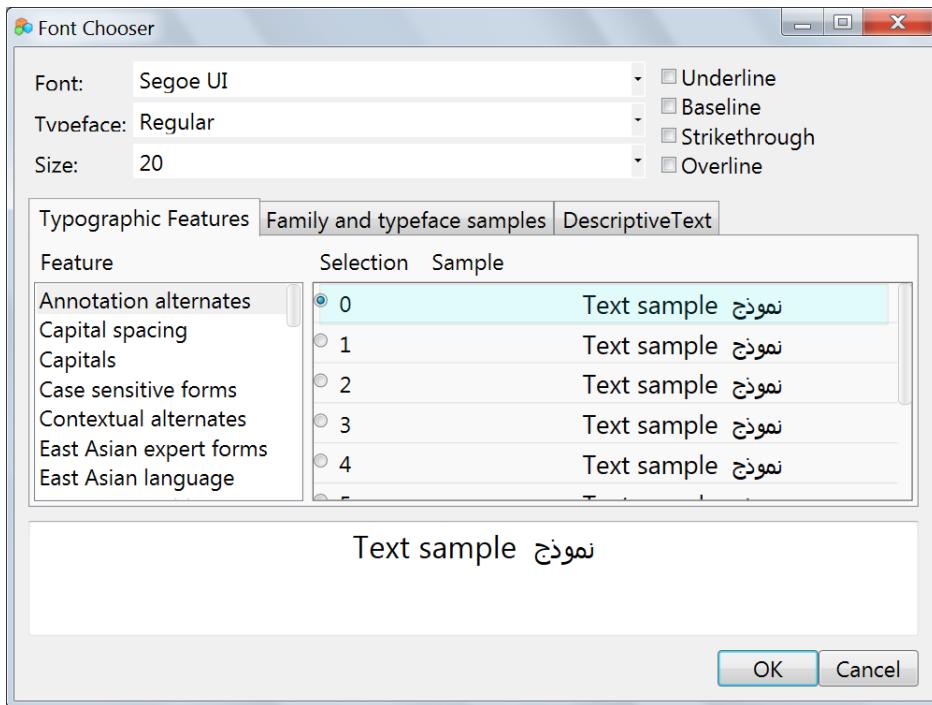
You can expand this submenu to see its commands and their keyboard shortcuts, where you can notice some standard commands with famous shortcuts used in text editors like MS Office, such as:

Command	Shortcut	Preview
Bold	CTRL+B	Sample Text
Italic	CTRL+I	<i>Sample Text</i>
Underline	CTRL+U	<u>Sample Text</u>
Decrease Size	CTRL+[Sample Text
Increase Size	CTRL+]	Sample Text

The submenu contains two more commands:

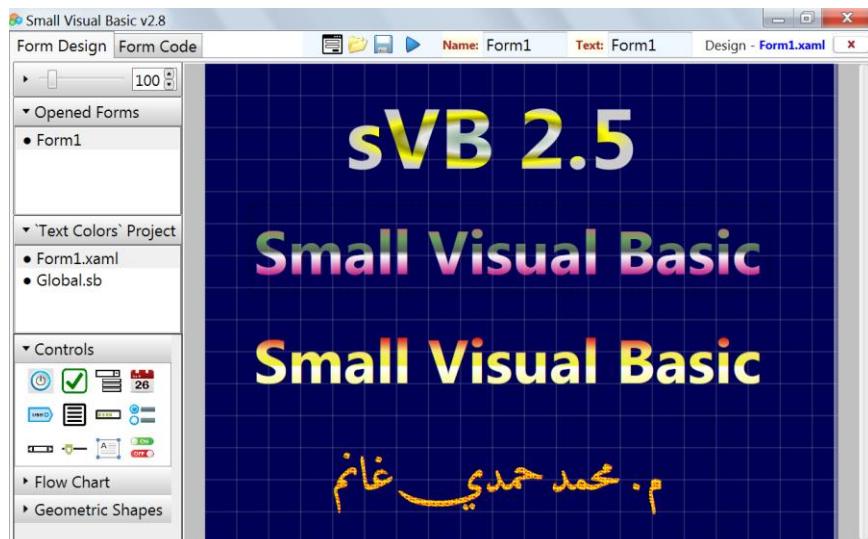
- Choose Font (Ctrl+F):

This command shows an advanced font dialog window, where you can choose the font name, size, style, effects and many other font properties that can't be covered in this book, but you can experiment with them as you want and see their effect of the preview sample text.



- Foreground (F7):

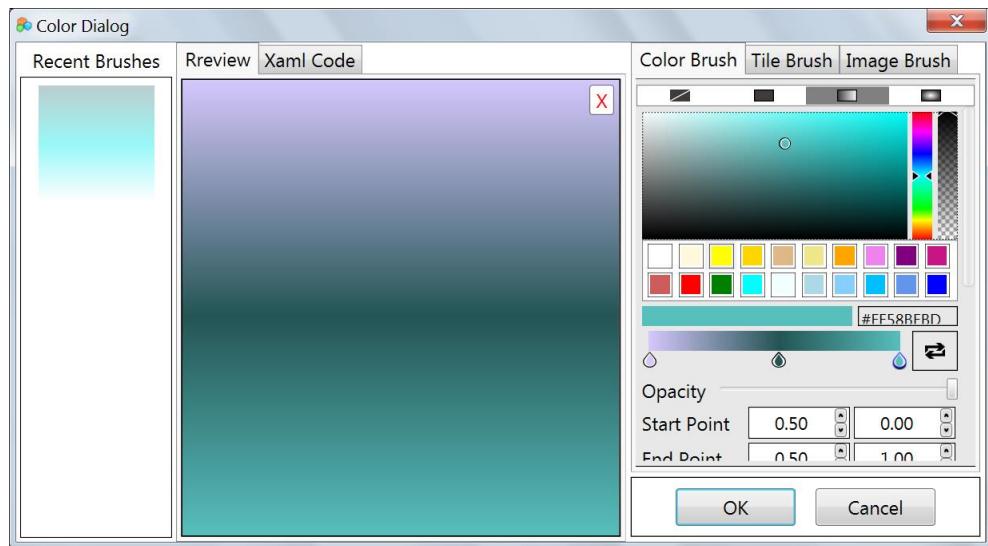
This command shows the color dialog window to allow you to compose and advanced color or image brush to be used in drawing the text, which means you can have amazing texts, like what you see in this picture:



The color window is explained in the next paragraph.

* Changing form and control colors:

You can use the color dialog to create an advanced brush to paint the form and controls with:



You can open this window, by clicking one of these commands from the context menu of the form and controls:

- Page Background: To change the form background.
- Diagram Appearance\ Diagram Background: To change the control background.
- Diagram Appearance\ Border Brush: To change the control border.
- Diagram Text\ Foreground: To change the control Foreground.

For more info, see the color dialog section below.

* Changing the control border:

Right-click the control to show its context menu, and expand the "Diagram Appearance" submenu. You will see 3 commands to deal with the control border:

- Border Brush (F3): Shows the color dialog to change the control border.
- Decrease Border Thickness (Shift+F6): Decreases the border thickness by 0.1.
- Increase Border Thickness (F6): Increases the border thickness by 0.1.

Note that the default button style may prevent you from changing its border thickness, but other controls will be OK.

* **Apply last change:**

You may know this amazing command from the MS Office. It allows you to apply the last property change you made in one control on the selected controls.

You can execute this command from the context menu or by pressing F1 from keyboard.

For example, add three buttons on the form, select the first one and change its back color, select the other two buttons and press F1 to apply the same back color on them.

You can also use this feature to give the same size for the three buttons. Use a corner resizing thumb of one of the buttons to change its width and height, then select the other two buttons and press F1 to give them the same size.

You can do the same with the location, but if you dragged the button by mouse to a new top and left position and applied that of the other buttons, the three buttons will be stacked over each other! The best way to use this, is to select a button, use the keyboard arrows to move it in one direction (press Ctrl to jump), then apply the change on the other buttons to give them the same top or the same left, which is a fast and precise way to align controls.

* Undo and Redo:

The context menu of the form contains the Undo and Redo commands, which you can also apply by the Ctrl+Z and Ctrl+Y keyboard shortcuts. As long as the form is opened in the designer, you can undo and redo any kind of change you applied on it or on any of its controls.

* The Color dialog:

Let's learn more about the color dialog window:

The window consists of three panes:

- The recent brushes pane:

Shows a list of recent brushes you have created. When there is no brush selected in the window, this list will show all recent brushes regardless their types, but when you select a brush, or switch to a brush type, only recent brushes of this type will be shown in the list.

You can click any brush in this list to preview it, or double-click it to apply it and close the window.

- The preview pane:

This pane shows a label to apply the brush you are composing on its background. You can double-click this label to apply the brush and close the window, or you can click the x button on its upper right corner to delete the selected brush (which is useful when you want to create no brush to remove the previous brush from the control).

Note the preview label is displayed on the preview tab, and when you switch to the XAML code tab, you will see a read only textbox that displays the XAML code of the selected brush. You can copy this code and reuse it as you want (for example, in style files that you use with the

[Control.SetResourceDictionary](#) and [Control.SetStyle](#) methods.

- The brush composer pane:

This pane allows you to compose the brush as you want. It shows three tabs for three different types of bushes:

- The color brush tab:

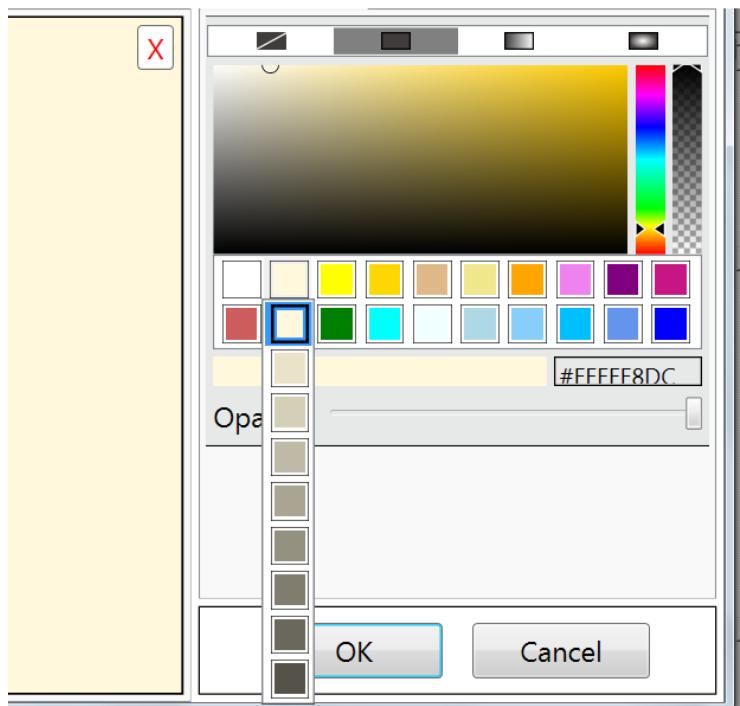
There are 4 tabs under this brush type:

- a. No brush:

The selected brush will be nothing. It is equivalent to clicking the x button on the preview pane, but the button is easier to reach, besides you may need to select another color brush type then reselect the "No Brush" tab to force removing the brush.

- b. Solid color brush:

This brush uses only one color to paint the area it is applied on.



There are 20 basic colors you can choose from, and when you click any of them, it will expand a 9 shades list, where you can choose the color you like.

But you are not confined to these 90 colors, as you can compose any custom color of your choice. To do that, click the colors palette on any position to select this point color. You can also drag the mouse over this palette to preview the selected color while dragging.

Note that there are two vertical sliders on the right of the palette. The first one changes the brightness ratio of the colors displayed in the palette, which allows to change the displayed color range, and the second one changes the selected color opacity, where the color transparency increases as you slide down.

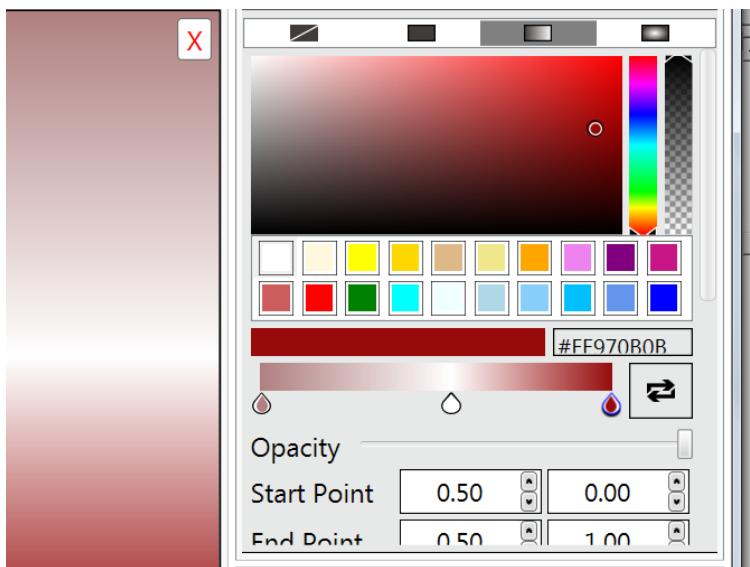
There is also a horizontal opacity slider under the palette. It seems redundant here, but it affects the whole brush while the vertical one affects only the selected color. The difference appears only in linear and radial gradient brushes, because they contain at least two color stops, so, you can individually change each color transparency.

Finally, There is a textbox that displays the hexadecimal representation of the color like #FFFFF8DC. You can copy this string to use it anywhere else. You can also change this string manually or past a new valid color string, then

press Enter to apply the new color and see its preview.

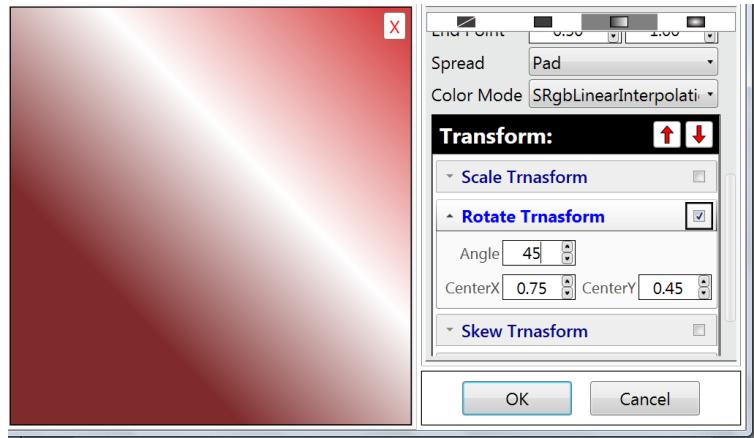
c. Linear gradient brush:

A linear gradient brush paints the area it is applied on with a color that is gradually changes to another one in the vertical direction. The colors are called stops, and you can add more stops between them to have many gradient colors.



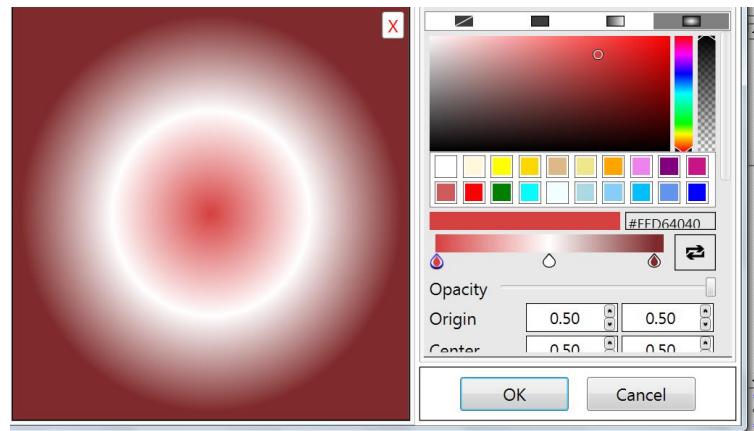
The linear gradient tab looks like the solid color tab, with an extra stops slider, that essentially has 2 stop colors. You can click any stop color to select it, drag it to the left and right, or press Delete to remove it (if there are still two more stops left, otherwise it will not be removed). You can change the color of the selected stop color as usual, and see the effect in the preview label. There are also more options that you can explore yourself. For example, you can rotate the brush

by checking the Rotate transform expander, and change the rotation angle and center.

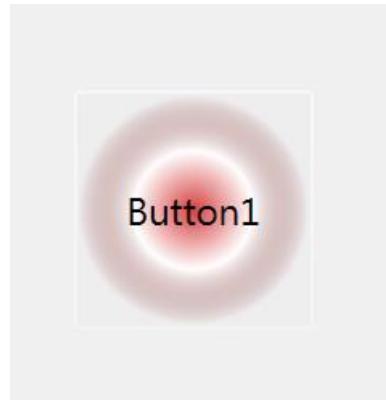


d. Radial gradient brush:

The linear and radial gradient brushes are similar, except that last one paints the area with gradient circles around the given center.

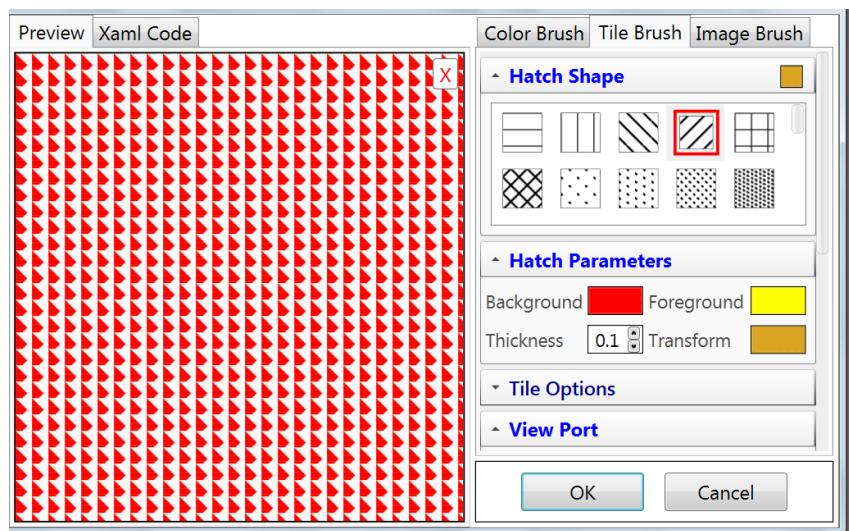


Note that if you make the last (outer) stop color fully transparent, this will change the shape of the control that uses this brush to paint its background to be elliptical (or spherical if its width equals its height), but you should also remove the control border by setting its color to nothing.



- The tile brush tab:

This brush tiles the area it is applied on with a hatch shape. There are 55 predefined hatch shapes you can choose from, and you can expand the hatch properties to change its foreground and background colors and thickness. You can also apply any transform on the this shape. Just click the rectangle next to each property name to view a popup window that gives you more options. The following picture show you the backward diagonal hatch shape after applying a 45 degrees rotation on it:



- The image brush tab:

This brush tiles the area it is applied on with the given image. If you didn't change any property of this brush, the image will be stretched to fill the entire area.

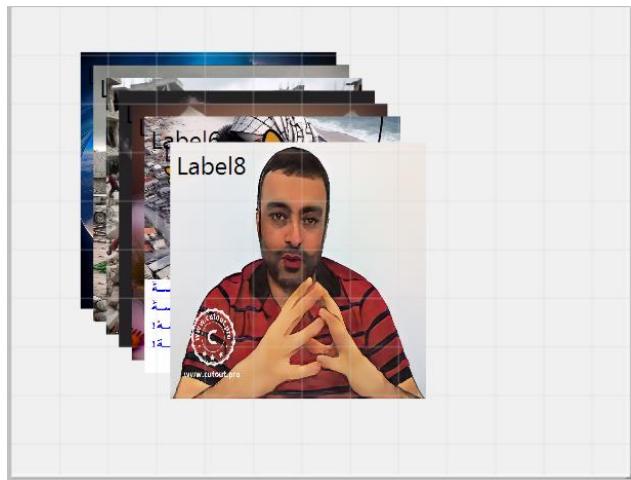
You can select the image from your file system by clicking the button with the three dots title to show the open file dialog.



There are many options that allows you to control how the image appears on the control. For example, you can mirror the image by expanding the Scale transformation by checking it and set the ScaleX to -1 to invert the image horizontally.



Note that you can drag one or more image files directly from the Windows File Explorer and drop them on the form designer. This will add labels to display each of these images, and you can still open the color dialog and use the image brush tab to change the properties of any of these images.

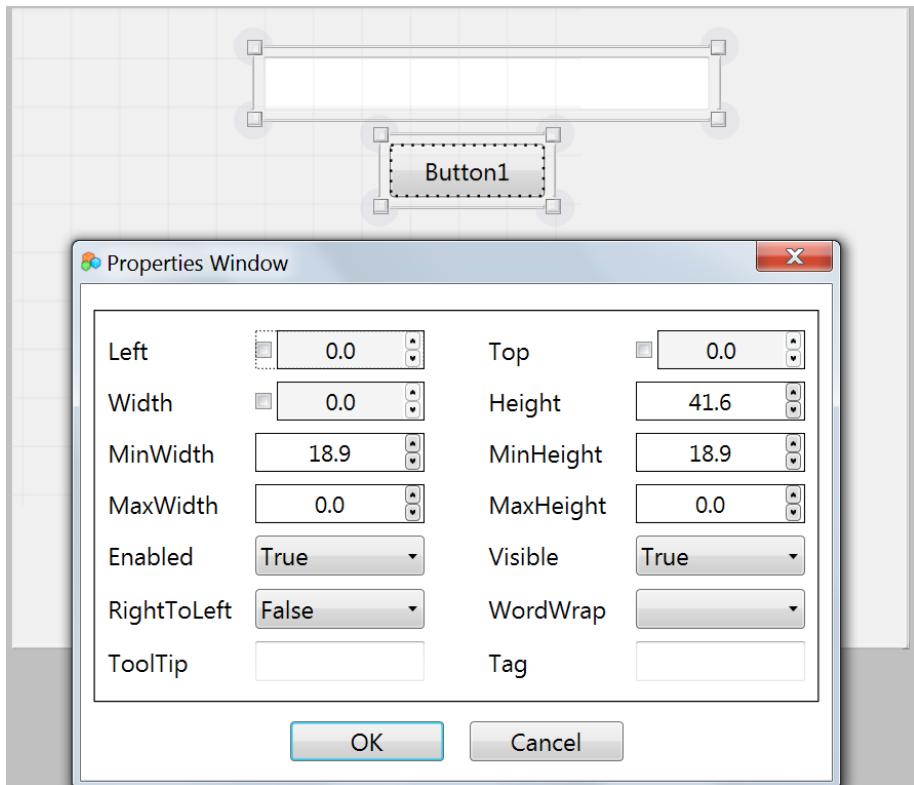


* **Changing form and control Properties:**

The form designer allows you to use the properties window to change a few more properties that are common to the form and controls.

To show the properties window, click the properties button from the upper toolbar, or use the Properties command from the context window, or press F4 from your keyboard.

The properties window can view and change the properties of the selected item, which can be the form, a single control, or many controls. In the following picture, you can see that textbox and the button are selected, hence the properties window shows their properties together:



Note that:

- a. The properties window shows 14 properties, that are common to the form and all control types. You can find more details about these properties in the [Control Type](#).
- b. When showing the properties of more than one control, it will show the properties that have the same value (like Enabled and Visible in the above picture, because both the textbox and the button have these both properties set to True).
- c. The properties window will disable any property that have different values for the selected control (like left, top and width properties). In this case, it will show a checkbox beside the disabled property to allow you to enable it if you want to enter a value for the property, which will be set to all selected controls. This can be

useful if you want to give the same size to all controls, or align their lefts or tops.

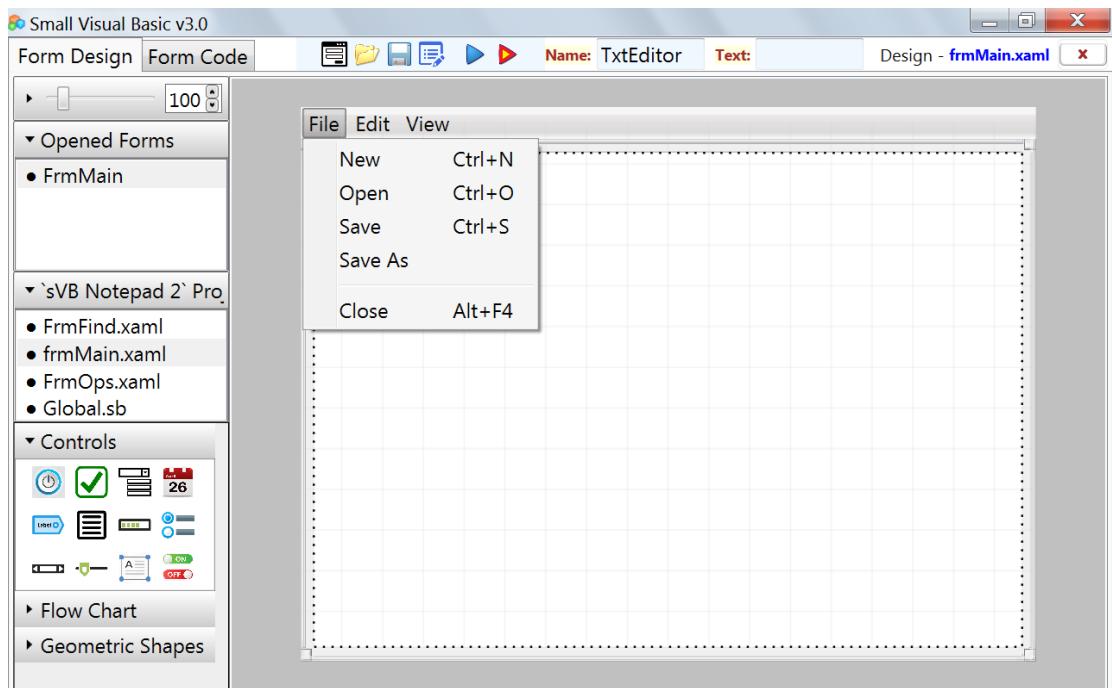
- d. The properties that offer a dropdown list to choose the value from (like Enabled and Visible) will not be disabled not show a checkbox, but it will have no selected item when the controls have different values (like the WordWrap in the previous picture), so if you want to set a single value for all selected controls, just choose it from the list, but note you can't remove this value and the only way to discard it is to press the cancel button.
- e. If you want to use an auto width or height, just use 0 as the value for these properties.
- f. If you want to use unlimited MaxWidth or MaxHeight, just use 0 as the value for these properties to set them to infinity.
- g. When you show the from properties, the left and top properties will be disabled to indicate that the form will be shown at the center of the screen. If you checked and enabled at least one of the two properties, the form will be shown at the given location, where (0, 0) is the upper-left corner of the screen.
- h. The Visible property of the form will always be disabled and will not have a checkbox to enable it. This is because the form is hidden until the Show method is called to display it, hence the Visible property is set to True. So setting the Visible property at design time will have no effect on how the form is displayed, but you can set it to False at runtime to hide the form.
- i. The Enabled property of the form will always be disabled and will not have a checkbox to enable it. This is because

showing a disabled form will make it useless because the user can't interact with it.

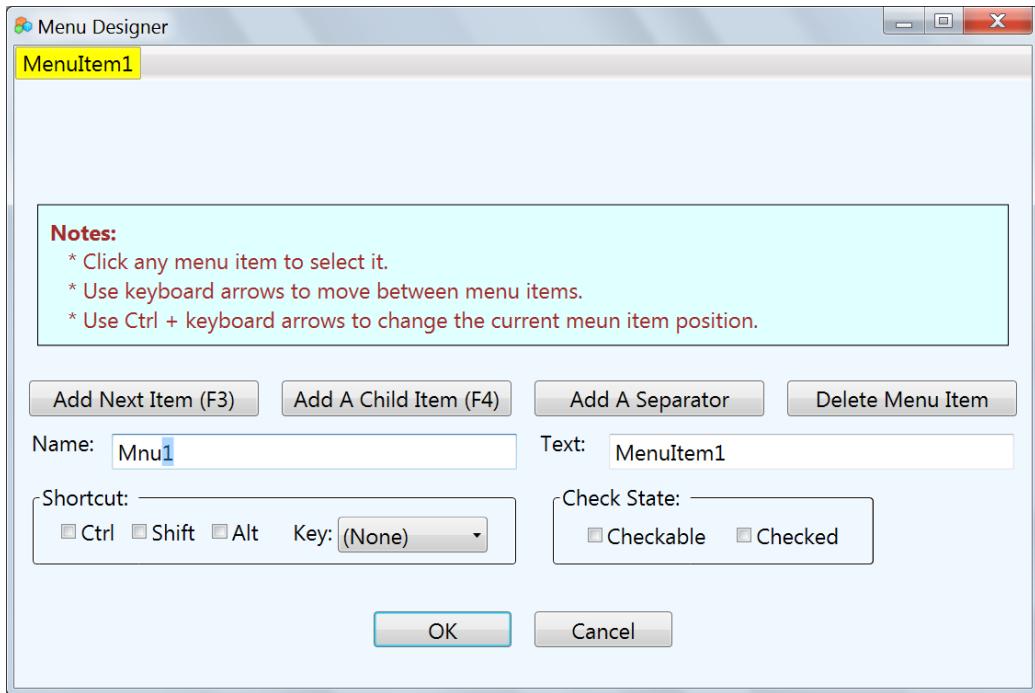
- j. The WordWrap property of the form and some other controls like the ListBox, ProgressBar and DatePicker will always be disabled and will not have a checkbox to enable it. This is because this property is not common for all controls, and available only of the Label, TextBox and Button controls (like Button, CheckBox and RadioButton). But if at least one of the selected controls has the WordWrap property, it will be enabled and you can set its value, which will be applied only on valid controls, and be ignored for invalid controls.

4. The Menu bar:

This is part of the form designer appears above the design canvas, but will not be visible until you create a main menu for the form.



To add a main menu to the form, right-click the form surface, then click the "Menu Designer" command from the context menu, to show the menu designer window:

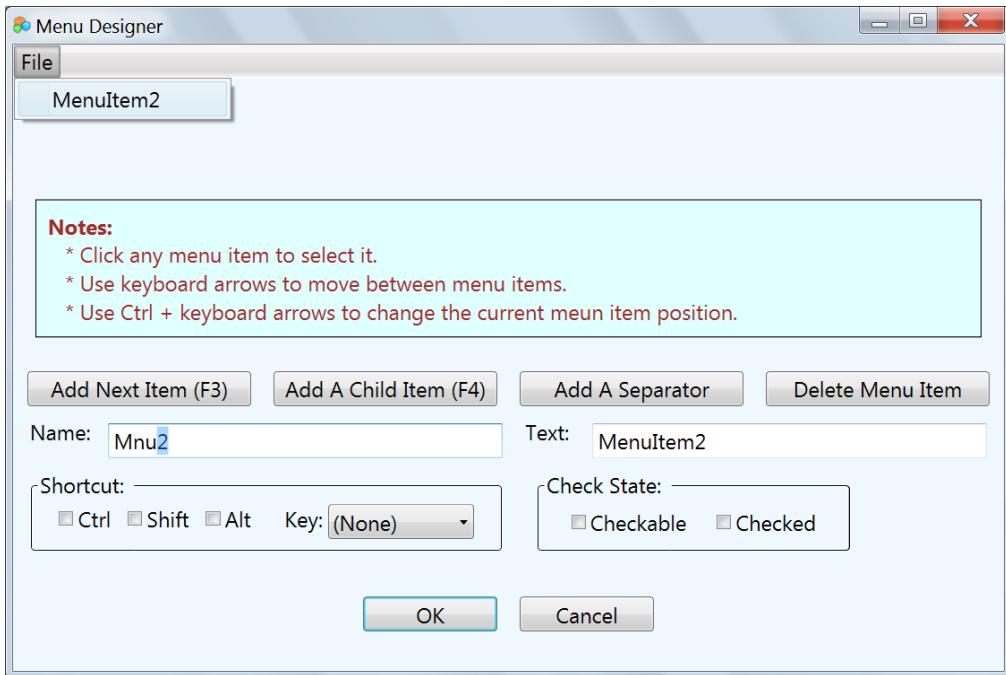


The menu designer allows you to easily and quickly create [menu items](#) and set their properties.

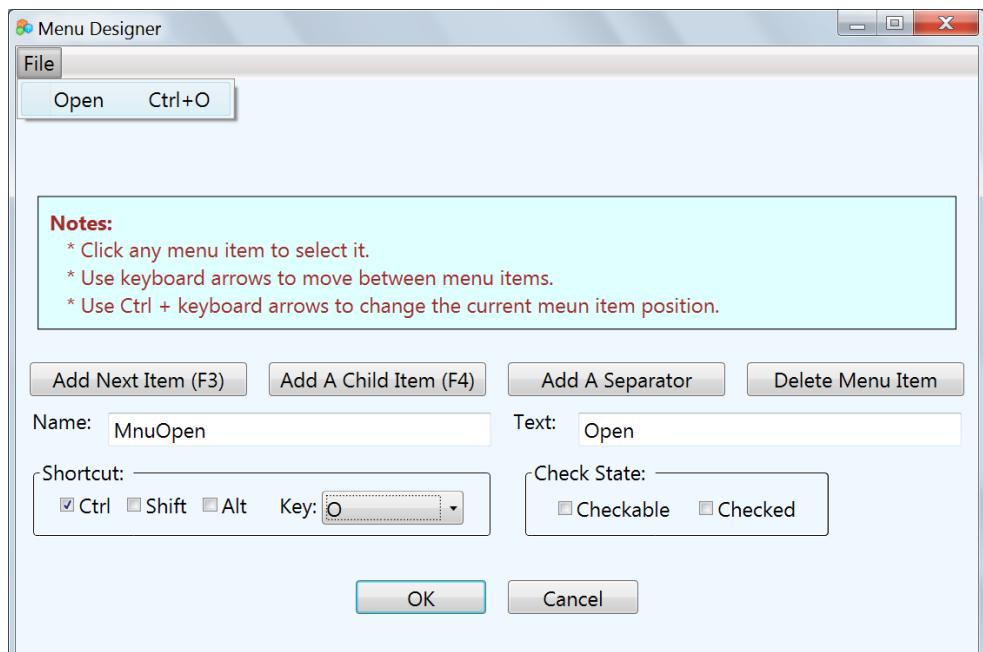
By default, a `menuItem1` is added, where you can change its name and text. It is recommended that menu names start with the "Mnu" prefix, and the name field helps you with that by suggesting names like `Mnu1`, `Mnu2`... etc. As you can see, the numeric part is auto selected, so you can directly type the name of menu (that reflects its job, like File and Edit).

Furthermore, the text field will be changed to the name you are typing (like File and Edit).

So, let's type "File" to have the name "`MnuFile`", then click the "Add child Item" button or press F4 from keyboard, to add a sub menu under the File menu:

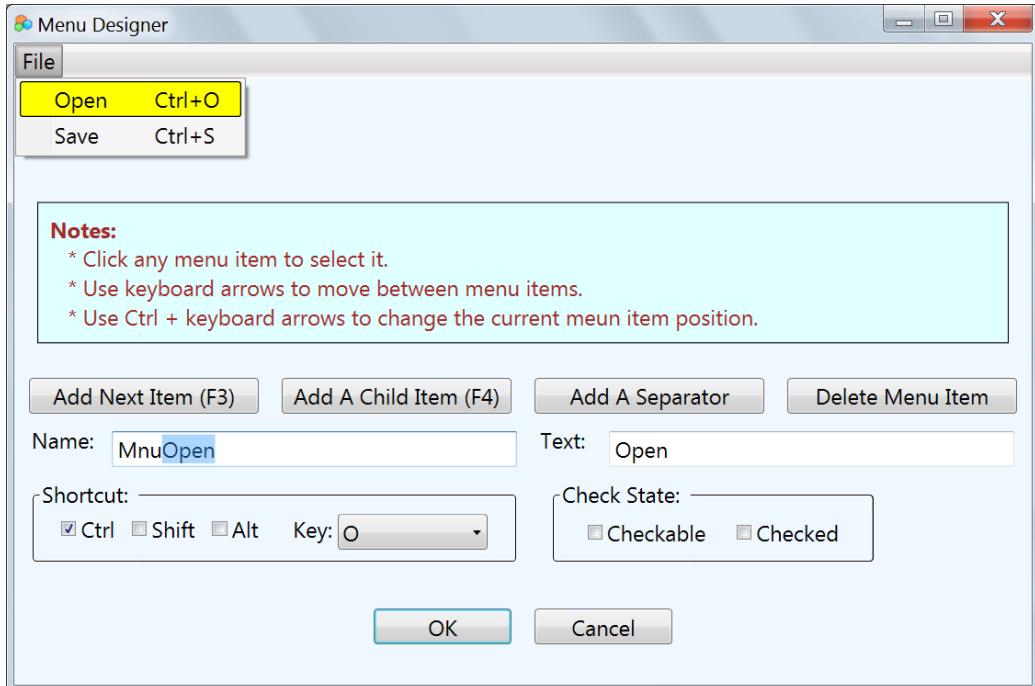


Type "Open" to change the name of this new menu item to MnuOpen and its text to Open, then check the Ctrl and choose the O letter from the Key dropdown list (you can select the list and press o to find it). You can now see the Ctrl+O shortcut on the text of the menu item that is displayed at the top of the window:



Now to add another child items under the File menu, click the "Add next item" button or press F3 from the keyboard. Don't use the "Add Child Item" button, because it will add the new item as a child of the MnuOpen, but if you did, you can use the "Delete Menu Item" button to delete it.

Note that you can click any menu item to select it, so you can modify its properties or add child items under it.



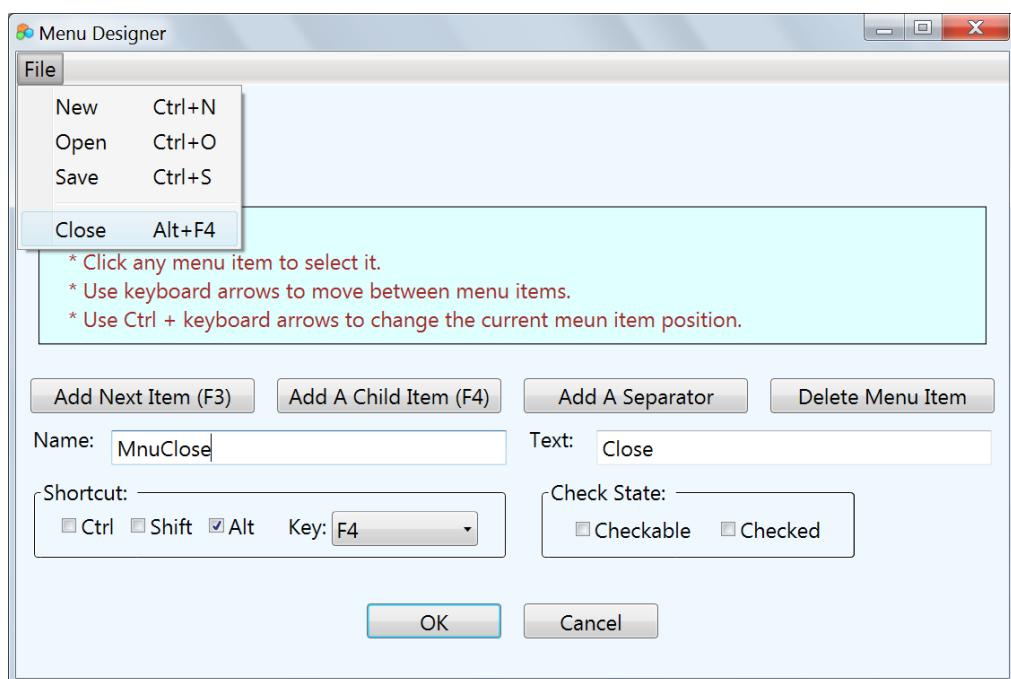
This also allows you to control the order of the child items. For example, when you select the Open menu as in the above picture as click the "Add Next item" button, then new item will be added between the Open and Save menus. Let's name it New and give it the Ctrl+N shortcut.

But, what if you want to move the New menu one step up to appear before the Open menu? That's easy, by just pressing the Ctrl+Up from your keyboard! Here are some useful shortcuts that helps you reorder menu items:

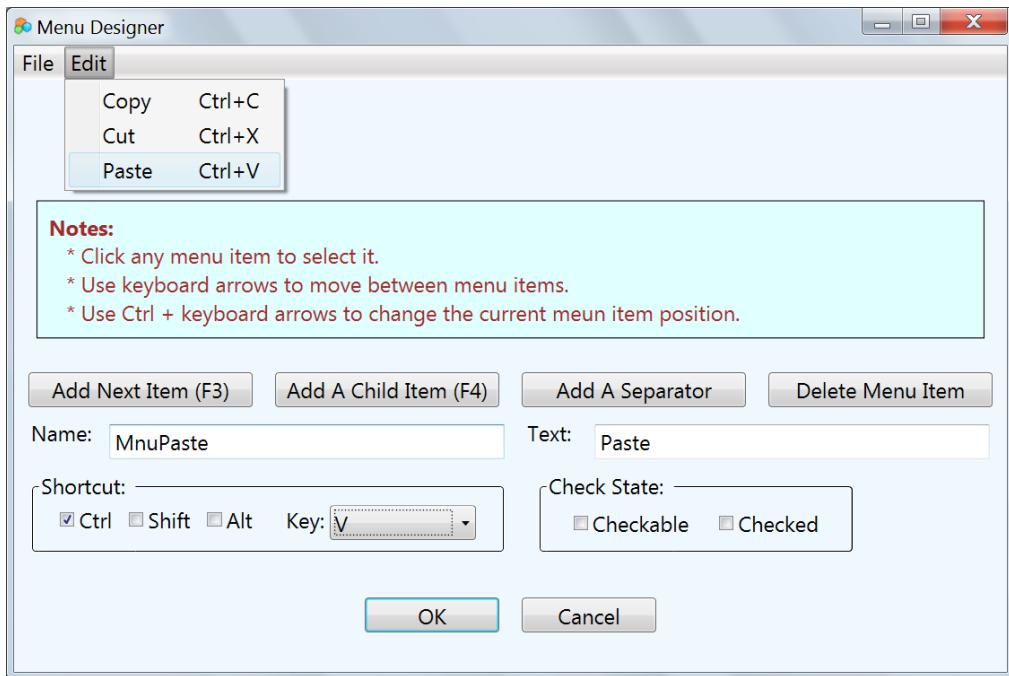
- Ctrl+Up: Move the menu item up.

- Ctrl+Down: Move the item down.
- Ctrl+Left: Move a child menu item out to appear next to its parent menu item.
- Ctrl+Right: Move a menu item in to appear as a child of its previous sibling menu item.

Now, let's add a separator under the Save menu. Click the Save menu to select it, then click the "Add A separator" button. This will add a separator line under the Save menu. Now click "Add Next Item", name it Close, check Alt and select the F4 from the key list.



Now, let's add the Edit menu. To do that, select the File menu, then click the "Add Next Item" button, to add a new item on the main menu bar. Name it Edit, and add some child items under it, like Copy (Ctrl+C), Cut(Ctrl+X) and Paste(Ctrl+V).



Be careful that all the work we are doing is not committed yet, and you can lose it all if you click the cancel button! You need to click the OK button to add the main menu to your form, and you can reopen the menu designer at any time to modify it as you like.

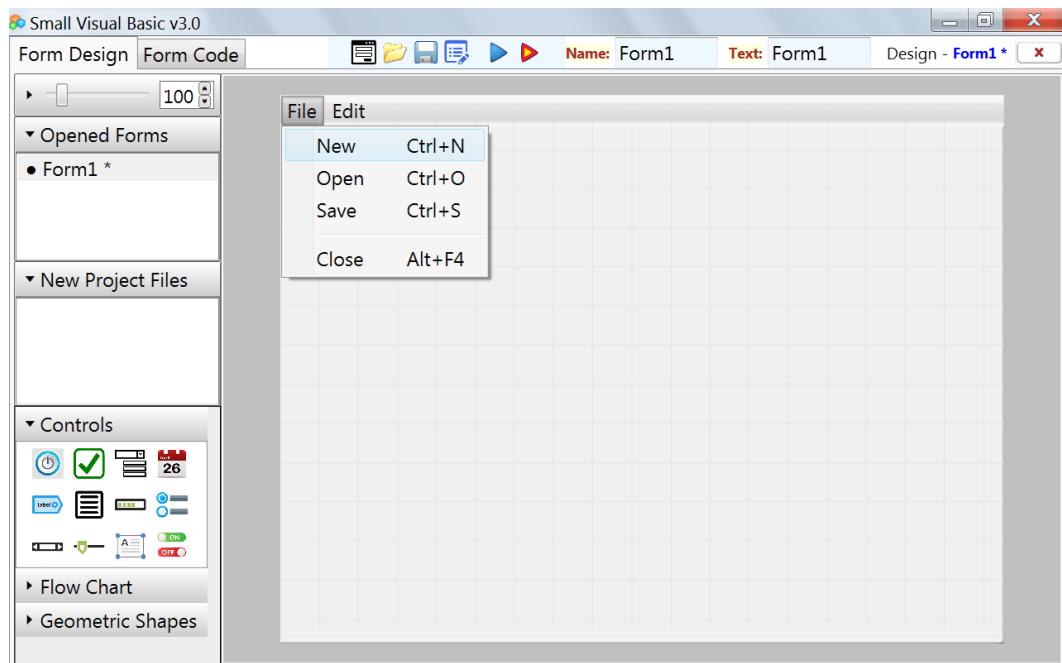
But before you click OK, let's understand what the Checkable and Checked checkboxes do:

- MenuItem Checkable: When you check this box, the user can click the menu item (at runtime) to check or uncheck it.
- Checked: When you check this box, the check mark will appear on the menu item, so it is checked.

You can see how those properties are used with the "Word Wrap", "Right to left" and "Multiline" menu items under the "View" menu in the "FrmMain" form in the "sVB Notepad 2" application in the "sVB samples" folder.

Now click the OK button to close the menu designer and add the main menu to the form.

You can now see the menu bar above the form surface, showing the menu items that we have just designed:



You can click any child menu item (like New) to add its [OnClick](#) handler to the code editor, and double-click any parent menu item (like edit) to add its [OnOpen](#) handler to the code editor. So, let's try the OnClick event of New menu item by writing this code:

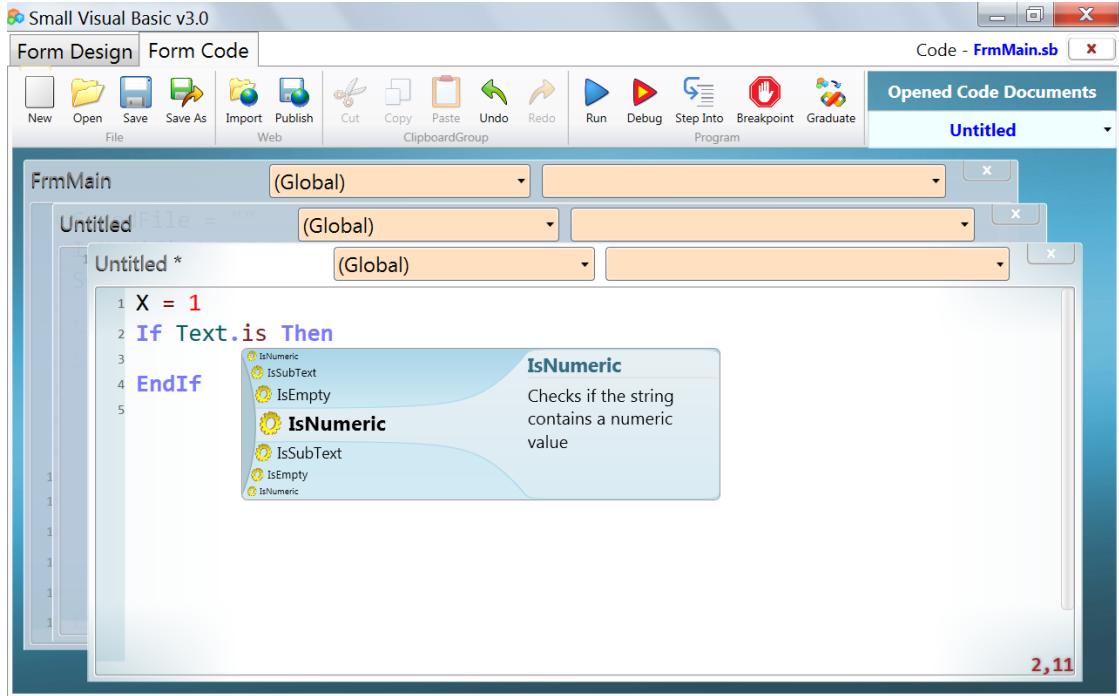
```
Sub MnuNew_OnClick()
    Me.ShowMessage("New is clicked", "Test")
EndSub
```

Press F5 to run the project, expand the Open menu then click the New command. This will show the message box.

Close the message box, then press Ctrl+N. This will also show the message box, which means that you don't need to write any code to react to the shortcut keys of the menu items, because the menu designer has already done the work for you!

The sVB Code Editor

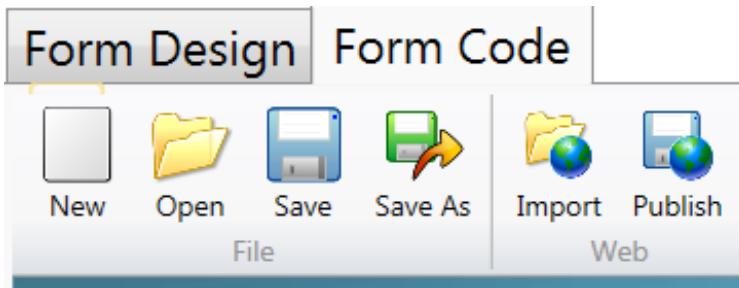
The sVB editor is built on the SB editor with many improvements and additions.



* Opening and saving documents:

The editor can open multiple documents at the same time, where you can resize them and drag them to change their locations inside the editor. You can activate any document by clicking it, or by choosing its name from the upper right dropdown list named that displays the opened documents.

You can close any document by clicking its x-button, and you can close the active document by clicking the general x button on the upper toolbar.



Use the "New" button from the toolbar (or press Ctrl+N) to open a new empty code document. This document is a single code file that is not related to any form, and you can run it alone by clicking the Run button on the toolbar, or pressing F5. This allows you to run Small Basic code files. But if you want to create a new form, you should switch to the Form Design tab and use the New button from there. When you switch back to the Form Code tab, the document file of the current selected form will be opened and activated.

Use the "Open" button from the toolbar to open an existing sb file. If the sb file is the code file of a form, its design will be opened when you switch to the Form Design tab.

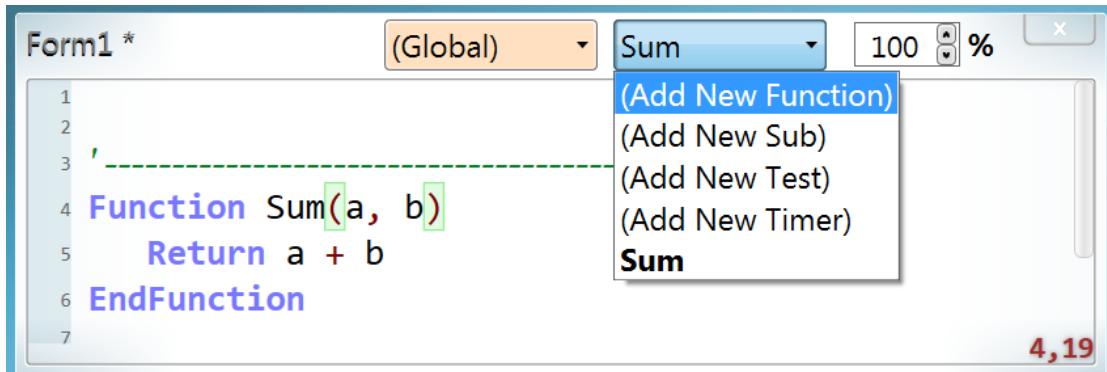
You can also use the Import button to open a code file saved on the Small Basic server, but you should have the ID of this file. You can also save any file to the server by clicking the Publish button, and you will get the ID of the saved file so, you can share it with your friends.

Note that saving an sVB file that is related to a form will not save the form design nor other forms in the project, so you should publish stand alone files only, and save your project files to your file system using the Save and Save As buttons.

There is another button that is suitable for pure small basic files only, which is the Graduate button that converts the Small Basic code to VB.NET project. This button will not work correctly if the code file contains any new sVB syntax, and I didn't even try to

upgrade it, because the sVB project needs to be converted to a WPF project, which will not be an easy transition for kids and beginners, and it will need a lot of work to try to translate it to a windows forms project!

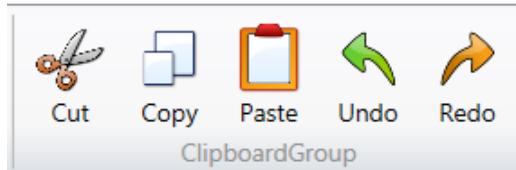
* The code document:



You can write your code in the document window, which provides you with the standard editing operations with the same famous keyboard shortcuts like:

- Copy (Ctrl+C)
- Cut (Ctrl+X)
- Paste (Ctrl+V)
- Select All (Ctrl+A)
- Undo (Ctrl+Z)
- Redo (Ctrl+Y)... etc.

You can also do some of these jobs using the buttons of the toolbar.



The code document shows a left margin, which offers you some useful options:

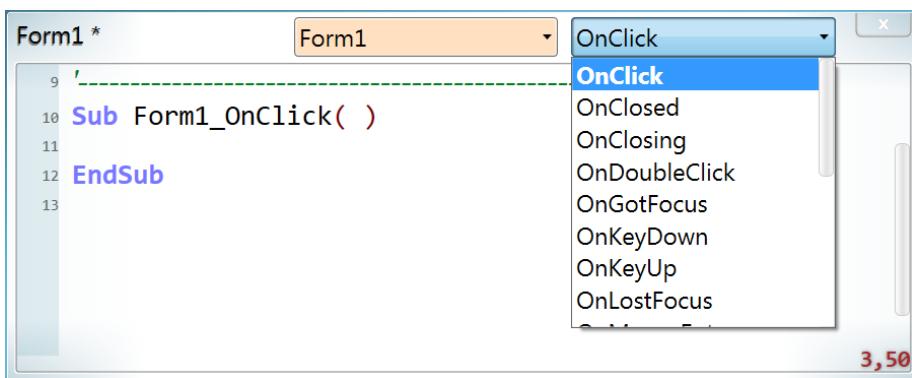
1. It shows each line number.
2. You can click this margin to select the entire line, excluding the new-line characters at the end of the line, so if you press Delete, this will remove the line completely.
3. You can select multiple lines by clicking the margin then dragging the mouse up or down.
4. You can extend the selection by pressing the Shift key and clicking the margin at another line, to select all lines in between.
5. You can double-click the margin to add a breakpoint or remove an existing one. For more information see [Debugging sVB projects](#).

The document also shows a bottom-right label that displays the line and column numbers of the line that currently contains the caret.

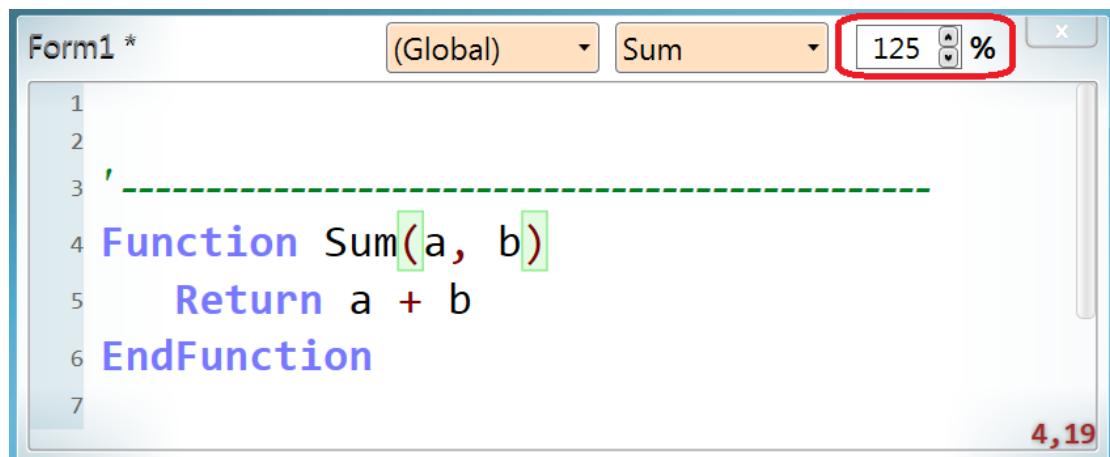
The document title bar shows its name, and two dropdown lists:

- The first list shows the (Global) item, and the names of the form and controls if the document is related to a form.
- The items of the second list depends on the item selected from the first list:
 - When you select the (Global) item, the second list will show the (Add new sub) and (Add new function) items and when you click them a new empty sub or function will be added to the file so that you can change its name and start writing its code. The list will also show the names of all subroutines and functions written in the document, where you can select any of them to select it in the document, which makes easy for you to navigate between the document subroutines and functions.
 - When you select the name of the form or any control, the second list will show its event names, where you can

select any of them to add its handler subroutine to the document.



The document title bar also shows a zoom box, where you can write a number between 10 and 1000 to represent the percentage of the zooming you want to apply. If you delete the number written in the box, this will reset the zooming to 100%.



You can also use the spin up and down buttons of the box to increase or decrease the zoom ration by 5%, or you can simply click the box to make sure it is focused, then move the mouse wheel up and down to change the zoom ratio.

Note that you can change the zooming also when the code document is focused by clicking the Ctrl key from the keyboard and moving the mouse wheel up and down.

The last zoom ration you set, will be used as the default zooming for all the documents you open after that, but this will not affect the already append document until they are closed then reopened.

* **Code formatting and pretty listing:**

The code editor makes it easy and fun to write sVB code, as it auto formats it for you while you are typing, so you can just focus on the job of your program:

- It colors the sVB keywords, literals, types and methods, which makes the code clear and readable.
- It adjusts line indentation for code blocks and fragmented lines, so you can have an organized code structure.
- It adjust spaces between tokens, to add missing spaces, and erase extra ones.
- It fixes the keywords and identifiers casing.
- It enforces using lower-case initial letters for local variables, and upper-case initial letters for global variables, labels, subroutines and functions.
- It reformats the current sub after leaving any line that has changes.

* **Intellisense:**

sVB got rid of the SB side help panel to save space. Instead, it shows the help info in a tip window that pops up after 2 seconds from moving the caret to any token (word) in the code editor, and stays open for 10 seconds, unless you move to another position, scroll through the document, or press the Esc key.

TextBox1.AppendLines(

TextBox.AppendLines(lines)

Appends the items of the given array as new lines in the textbox, just after the last character of the textbox

Parameter Info:

lines: An array containing the lines to add to the textbox.

To be less annoying, sVB prevents showing the help for the same token until you move to another one then come back, but you can force sVB to show help by pressing F1 at any time.

You can select any part of the text displayed in the popup window. You can also magnify the font by pressing Ctrl while moving the mouse wheel. This is one of many functionalities built-in the WPF FlowDocument control that is used to show the help info.

The pop up help offers valuable info about the current code token, including:

- The variable scope (local or global).
- The definition signature of subroutines, functions, types, properties, dynamic properties events and Method including their parameters and return values.
- If the token is a variable, a subroutine or a function, it will be shown as a hyper link, so, you can click it to go to its definition line. If the token is the name of a form or a control that are created by the form designer, clicking the link will switch you to the form designer and select the form or the control.
- The popup info includes a summary about the selected token, and if it is a function or a method, it will show info about its parameters and return value.
- You can add a summary for variables, subroutines, functions, and dynamic properties by adding one or more comment

lines directly above their definition line, and/or add a comment at the end of the definition line.

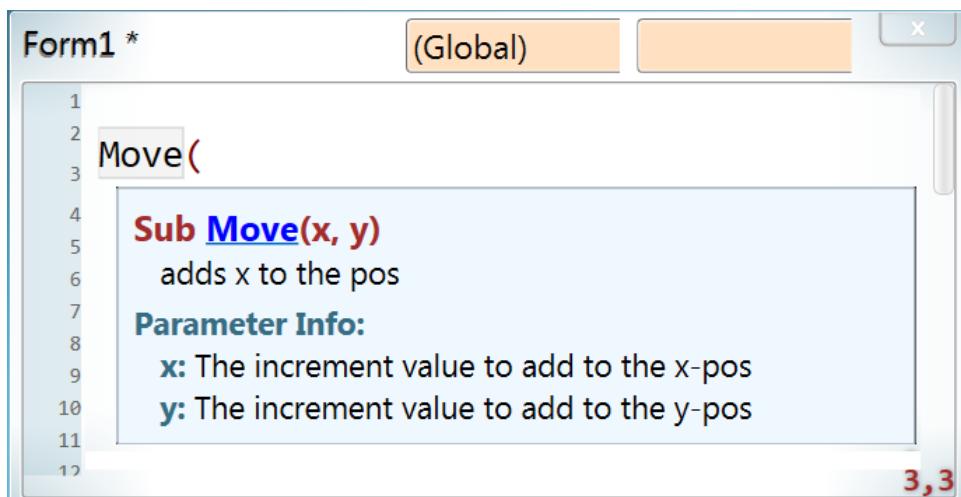
For subroutines and functions, you can add the additional summary comment after the opening parenthesis if you split them over multi lines, which also allows you to add a comment for each parameter to be used as its documentation.

For Functions, the comment placed after the closing parenthesis will be used as the documentation info for the return value, but for subroutines, it will be considered an additional line of the summary.

For example:

```
XPos = 1      ' the horizontal position
YPos = 1      ' the vertical position

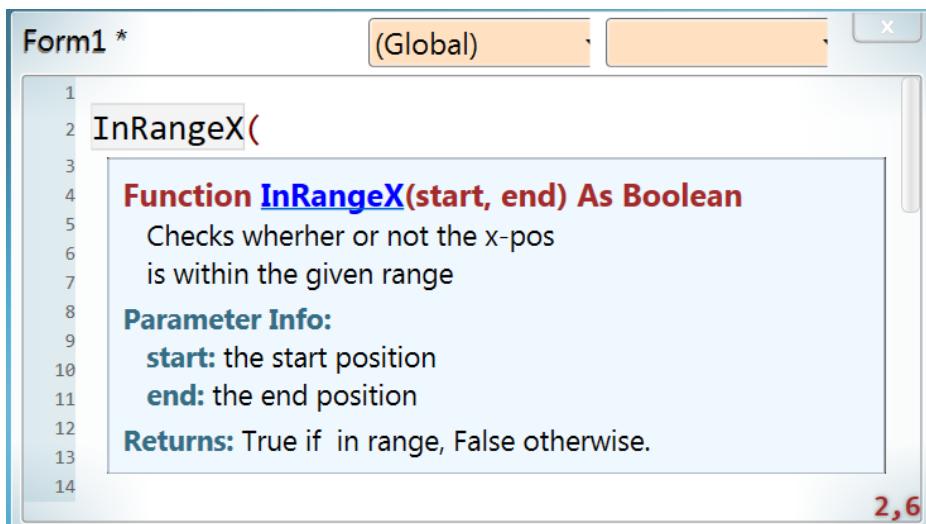
' adds x to the pos
Sub Move(
    x, ' The value to add to the x-pos
    y ' The value to add to the y-pos
)
XPos = XPos + x
YPos = YPos + y
EndSub
```



```

' Checks wherher or not the x-pos
Function InRangeX(' is within the given range
    start, ' the start position
    end   ' the end position
) ' True if in range, False otherwise.
Return XPos >= start And XPos <= end
EndFunction

```



While typing a method call argument, the popup help will show the target parameter with in red, and display only this parameter info, so you can focus only on the task in hand.



- When you move the caret to an opening token like ({ [, the editor highlights its closing token like) }] , and vise versa.

Form1 *

(Global)

```

1
2 A = {1, 2, 3, 4}

```

2,17

- When you move the caret to an identifier like variable, subroutine, function, label, type, method and property names, the editor highlights every occurrence of the current identifier in the code file.

Form1 *

(Global) Move

```

1
2 XPos = 1
3 YPos = 1
4
5 'adds x to the pos
6 Sub Move(x, y)
7     XPos = XPos + x
8     YPos = YPos + y
9 EndSub

```

7,14

- When move the caret to a keyword that belongs to a code block like the If, For, While, Sub and Function statements, the editor highlights all the basic block keywords, including the Return, ExitLoop and continue loop statements.

Form1 *

(Global) InRangeX

```

14
15 Function InRangeX(start, end)
16     result = XPos >= start And XPos <= end
17     Return result
18 EndFunction

```

17,6

- You can navigate between highlighted tokens by pressing F4 or Ctrl+Shift+down to go to the next highlighted token, and

press Shift+F4 or Ctrl+Shift+Up to go to the previous highlighted token. You can also press such shortcuts keys even there are no highlighted keywords, to highlight the nearest block that contains the current line, and move to the next or previous highlighted token.

* Auto completion:

The intellisense also offers you an auto-completion list that appears while you are typing keywords and identifiers.

The completion list filters identifier names by partial words. for example: Typing "name" can select Name1 and MyName.

The list tries to remember the last selected object for each first letter, and the last selected used method for each object.



You can use the keyboard Up and Down arrows and the mouse wheel to scroll through the completion list, and you can write the selected item in the code editor by double-clicking it or pressing Enter, space, or any of the following symbols: . . ([{ + - * / = > < . You can also press Ctrl+Enter to commit the suggested name to a new line. This is helpful when you are splitting the arguments over multi-lines, so, when you write one argument then "," and the auto completion appears to offer something, just choose the name and press Ctrl+Enter to add it to the next line.

Note that the auto completion list offers the color names when you deal with a color variable or sending a color argument to a

method. The same will happen when you deal with other enum types like Keys, DialogResults, font names, and form names.



In all those cases, the list will show only the members names without the enum name, but the full qualified name will be written in the code when you choose an item, like writing Colors.AliceBlue when you choose AliceBlue from the colors list. You can dismiss the completion list by pressing the escape key, and you can press Ctrl+Space to show it again if it contains more than one item, otherwise the only suggested name will be written directly to the code!

The editor also completes block statements. For example, after you type a space after If, the editor will add Then and EndIf. The editor also adds the closing char like ",),], } after you type the opening one like ", (, [, {. The editor is smart enough not to add the closing char when you immediately write it after the opening char, and in some other situations like typing " inside a string literal.

Furthermore, the editor allows you to use some shortcuts for some especial methods. These shortcuts do not change sVB syntax, because the editor replaces them with the formal code and correct syntax once you leave the line, save the document, or run the project. Let's look at these shortcuts:

- You can use MsgBox as a shorcut name to show the message box with the default title "Message". Ex:
MsgBox "Hello!"

or:

MsgBox("Hello!")

When you leave the line or save the file, the above code will be converted to:

Forms.ShowMessage("Hello!", "Message")

- You can also provide the title for the MsgBox shortcut syntax like this:

MsgBox "How are you?", "Hello"

or:

MsgBox("How are you?", "Hello")

Which will be converted to:

Forms.ShowMessage("How are you?", "Hello")

- You can use these shortcuts for the TW methods:

Shortcut	method	example
?	TW.WriteLine	? "Hello"
	TW.WriteLine	? x, y
	TW.Read	x = ?
:?	TW.Write	? : "Enter x: "
#?	TW.ReadNumber	x = #?

For more details, see the samples provided for these methods in the [TextWindow](#).

- The sVB code editor makes it easy to compose a multi-line string literal. All you need is to press Enter within the string literal, and the code editor will split it over two lines, and add the necessary code to embed the new line string. For example, type:

? "Line 1

then press Enter and type:

Line 2

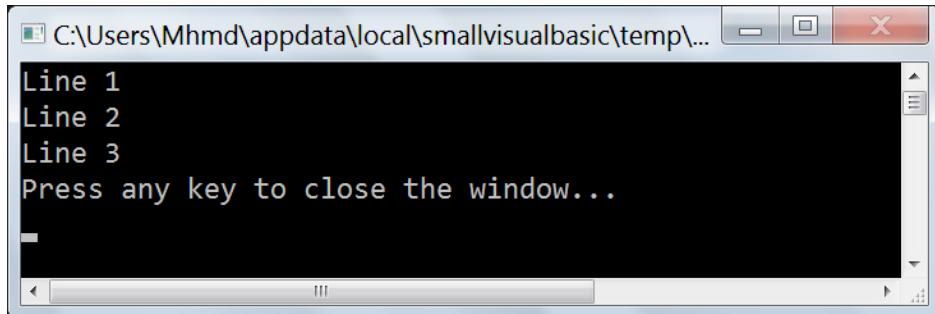
then press Enter and type:

Line 3

This is what you get on the code editor:

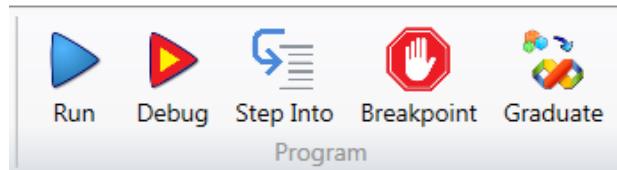
```
TW.WriteLine("Line 1" & Text.NewLine &
    "Line 2" & Text.NewLine &
    "Line 3")
```

And this is what you get when you run the code:



* The program buttons on the toolbar:

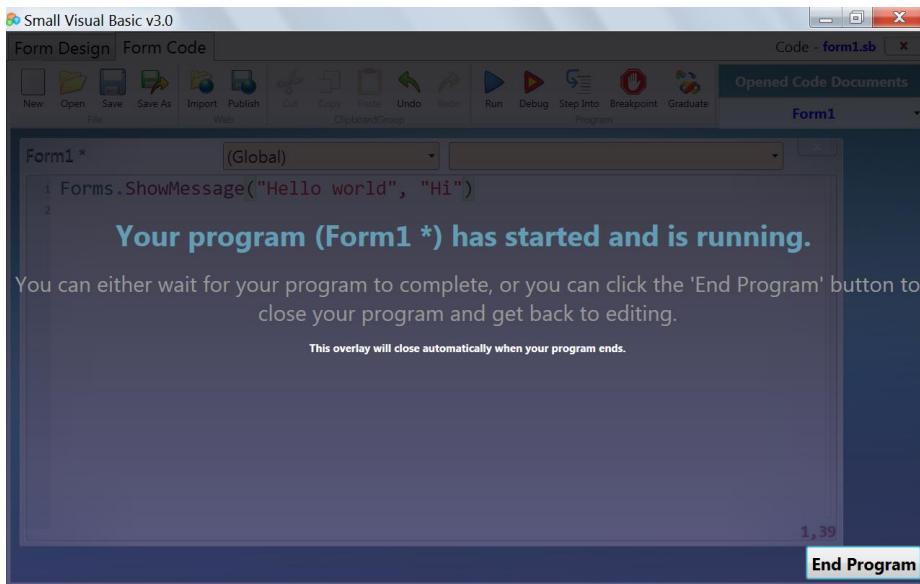
The editor toolbar has a program section that displays 5 buttons:



- **Run (F5):**

Runs the current project starting with the current form. If the current file doesn't belong to any form (single code file), it will be executed alone as old Small Basic Files. But if this code file is the global file, it will be considered a part of the project, and the first form in the folder will be the startup form of the project.

When the project runs, the IDE is blocked behind a semi-transparent screen to prevent you from making any change until the program ends. This screen displays an "End Program" button for that purpose, or you can also close the program by closing all the displayed forms.



- **Debug (Ctrl+F5):**

This button is similar to the Run button, but it runs the project in debug mode. For more info see [Debugging sVB projects](#).

- **Step Into (F11):**

Runs the project in debug mode and pauses at the first line, so we say that the program is in break mode, and you can click step into again to execute the current line code and pause at the next one.

For more info see [Debugging sVB projects](#).

- **Breakpoint (F9):**

Toggles the breakpoint at the current line in the code document, so if you click it once, the line background turns red, and a red circle will be added to the line left margin, and if you click it again, the line will return to its normal background and the red circle will be removed from its margin.. and so on.

Note that you can also toggle the breakpoint of any line by double-clicking its margin.

```
5 RacketDelta = 15
6 Timer.Interval = 30
● 7 Timer.Tick = MoveBall
8 RacketTimer = Me.AddTimer("RacketTimer", 20)
9 RacketTimer.Pause()
10 RacketTimer.OnTick = MoveRacket
```

Note also that if you make any changes to the code, all breakpoints in the document will be removed.

When you run the program in debug mode, the execution will pause at any breakpoint it encounters, but breakpoints have no effect when you run the program without debugging. For more info see [Debugging sVB projects](#).

- **Graduate:**

As mentioned before, the Graduate button translates Small Basic code to VB .NET code, but it has many issues.

Note that the code editor toolbar will change when you run in debug mode, where some buttons will be hidden, while others will be shown, as you can see in the following picture:



For more info about these new buttons, see [Debugging sVB projects](#).

Small Visual Basic Syntax

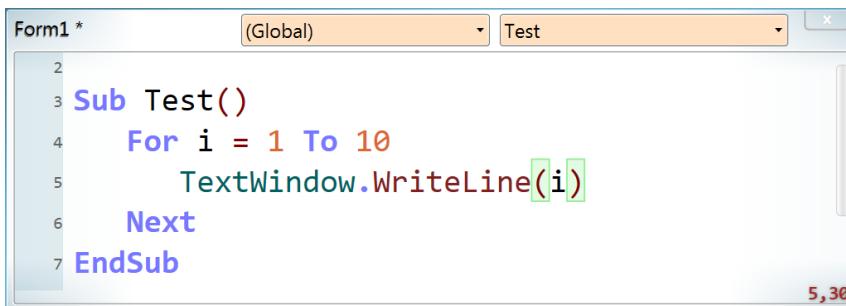
Each program consists of a set of instructions, each of them orders the computer to do a task like saving a value in RAM, reading data from a file, or drawing something on the screen.

Some of these tasks are executed by the language runtime like dealing with the RAM, and others are executed by the code libraries that contain method to deal with files and graphics, etc.. Either way, we need to write code to invoke those instructions, so we need to learn the syntax of sVB, to write its instructions in the correct way.

* Tokens:

The programming code consist basically of tokens. A token is a single unit of characters, which includes:

1. sVB keywords, which are preserved blue words that you can't use as names for variables nor functions.



```
Form1 *
(Global) Test
2
3 Sub Test()
4     For i = 1 To 10
5         TextWindow.WriteLine(i)
6     Next
7 EndSub
```

sVB has only 30 keywords (14 SB keywords + 16 new):

If, Then, Elself, Else, EndIf,
True, False, Nothing,
For, To, Step, EndFor,
ForEach, In, Next,

While, EndWhile, Wend,
ExitLoop, ContinueLoop,
Sub, EndSub, Function, EndFunction,
Goto, Return,
Global, Me, Global, Mod

2. identifiers, like variable, function and parameter names.
3. operators, like **+ - * / , . () [] {} ! And Or**
4. the line continuation character **_**, which is used to split the statement into two lines.
5. literals, like numbers, strings, and dates.
6. comments. A comment is a string token that starts with the **'** character.

* Expressions:

The expression is a part of an instruction. It can consist of one or more tokens. sVB has 9 types of expressions:

1. Identifier expression like **Name**.
2. Literal expressions like **1.5**, **"Hello sVB"** and **#1/1/2022#**.
3. Array expression like **x[1]**.
4. Array initializer expression like **{1, 2, 3}**.
5. Binary operator expression like **1 + 2** and **x > 3**.
6. Negate expression like **- n**.
7. Method call expression like **TextBox.Append("")**.
8. Property expression like **TextBox.Text**.
9. Dynamic property expression like **Std!Name**.

* **Statements:**

sVB instructions are called statements. There are two types of statements in sVB:

- **Single statements:**

Which consists of tokens and expressions, and written as a single line of code. The BASIC family languages don't need any symbol to indicate the end of the single statement, as it ends normally at the end of the line. For example, the following line of code instructs sVB to show the "Hello world" text on the form title bar:

```
Form1.Text = "Hello world"
```

The above line is an assignment statement, which uses the = token to assign a string literal to a property expression.

sVB has 7 types of such single instructions:

1. [Assignment statement.](#)
2. [Goto statement.](#)
3. [Label statement.](#)
4. [ExitLoop statement.](#)
5. [ContinueLoop statement.](#)
6. [Return statement.](#)
7. [Subroutine call statement.](#)

- **Code blocks:**

A code block consists of many single statements, grouped together to perform a certain job. Each block in Basic languages ends with a certain token (the **If** statement block ends with **EndIf**, the **Sub** block ends with **EndSub** and the **For** block ends with **Next**). For example, The next subroutine block will show the "Hello world" statement on the form title bar when the user clicks the button named Button1:

```
Sub Button1_OnClick()
    Me.Text = "Hello world"
EndSub
```

sVB has 6 types of blocks:

1. [If statement.](#)
2. [For statement.](#)
3. [ForEach statement.](#)
4. [While statement.](#)
5. [Sub statement.](#)
6. [Function statement.](#)

In the following sections, we will learn about the syntax of all sVB expressions and statements.

Identifier Expressions

The identifier is a single word that is used as a name for a variable, a Goto label, a subroutine, a function, a parameter, a type, a method, or a property.

A valid identifier must comply to these rules:

1. The identifier can only contain alphabetic letters, numbers and the hyphen (under score) _ , like **student**, **student2**, and **_student**.
2. The identifier can't start with a number, but numbers can appear anywhere else. For example the name **1Year** is not valid as an identifier, but then name **Year1** is valid.
3. The identifier can't contain any symbols, like quotes, parentheses and spaces. For example, **ab,c** , **"hello"** and **my name** are not valid identifiers.
4. If you want to create an identifier name from more than one word, you can use a hyphen between them (like **my_name**), or combine them directly using Pascal casing (like **MyName**, which is used in sVB to name global variables, subroutines and functions) or camel casing (like **myName**, which is used in sVB to name local variables and parameters).
5. sVB keywords are not valid as identifier names, like Me, If, Then, For, GoTo.... etc.
6. sVB type names are not valid as identifier names, like Text, Math, Form, TextBox, etc.. If you want to use such names, add a hyphen prefix (like **_Form**) or a number suffix (like **TextBox1**).
7. You can't use two identifiers with the same name in the same scope. For example, you can't declare a global variable and a function both with the name Test, because this will confuse

the sVB compiler when trying to understand what you mean with the name Test, so it will give you error messages.

The screenshot shows the sVB IDE interface with a code editor window titled "Form1 *". The code is as follows:

```
1 Test = 1
2
3 Function Test()
4     Return Test
5 EndFunction
6
```

Below the code editor, a message box displays three errors:

- 1,1: 'Test' is a Subroutine, but is being used as a variable.
- 1,1: The variable 'Test' is used before being initialized.
- 4,11: 'Test' is a Subroutine, but is being used as a variable.

A red "Close" button is visible in the top right corner of the message box, and a red "1,5" is in the bottom right corner.

8. sVB, like all other BASIC languages, is case-insensitive. This means that identifier names can't differ from each other just by character casing, hence the names student, Student and STUDENT are actually the same and refer to a single identifier! So, don't worry about the character casing of the sVB keywords and identifiers. In fact the editor will format the code for you and restore the casing.

* Naming Conventions:

sVB tries to infer variable types from its initial values, but this may be not possible in some cases, such as when the initial value is a return value from a function that can return any value type. Besides, it is not possible to infer the parameter type. This is why sVB allows you to name the variable and parameter identifiers in a way that tells it about their data types. The naming convention is to add a certain prefix or suffix to the identifier to indicate the data type. The following table summarizes the valid prefixes and suffixes:

Prefix/suffix	Type	Examples
Str	String	Str1, str2, strName
String	String	_string, string_name
Arr	Array	Arr_Names, names_arr
Array	Array	_Array, myArray
Dbl	Double	DblN, dblN, nDbl
Double	Double	double_Age, doubleAge
Color	Color	Color1, color2
Key	Key	_Key, pressedKey
Dlg	DialogResult	dlgResult
Dialog	DialogResult	dialog1, dialogResult
Type	ControlType	type, typeName, cntrlType
Control	Control	control1, myControl
Frm	Form	FrmInput, frm_names
Form	Form	Form1, TestForm
Txt	TextBox	txtID, txt_result
TextBox	TextBox	IDTextBox
Lbl	Label	lblName
Label	Label	sumLabel
Lst	ListBox	IstNames
ListBox	ListBox	IDsListBox
Cmb	ComboBox	CmbCities
ComboBox	ComboBox	CountriesComboBox
Chk	CheckBox	chkAllow
CheckBox	CheckBox	VisibleCheckBox
Rdo	RadioButton	RdoGreen
RadioButton	RadioButton	RedRadioButton
Tgl	ToggleButton	tglItalic
ToggleButton	ToggleButton	BoldToggleButton
Btn	Button	btnApply

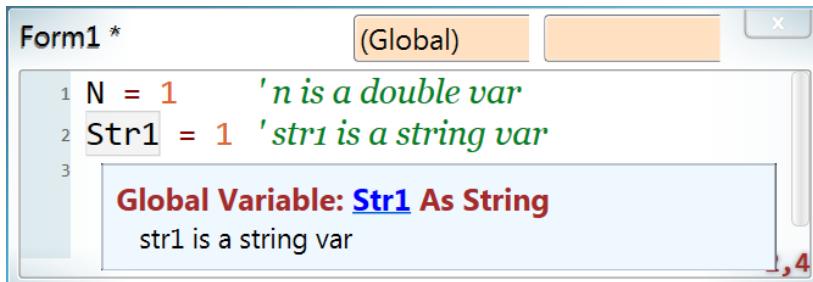
Button	Button	OkButton
MenuItem	MenuItem	MenuItem1
MainMenu	MainMenu	_MainMenu
Dtp	DatePicker	DtpBirthDay
DatePicker	DatePicker	DatePicker1
Date	Date	startDate
Duration	Date	meetingDuration
ProgressBar	ProgressBar	ProgressBar1
Slider	Slider	Slider1
ScrollBar	ScrollBar	ScrollBar1
ImageBox	ImageBox	ImageBox1
Timer	WinTimer	Timer1

Note that:

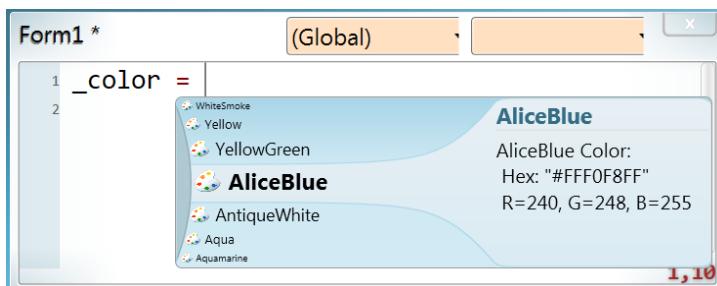
1. Those abbreviations can be use as prefixes or suffixes.
2. Those abbreviations should be distinguished from the rest of the word, by:
 - a. using `_`, like `str_name`.
 - b. using Pascal casing, like `StrName` or `MyStr`.
 - c. using camel casing, like `strName` or `myStr`.
 - d. appending a number, like `str1`.

These rules will prevent confusing cases such as a variable named "strange" that starts with "str" but it is just a part of the word not a prefix, so, it will not be considered of type "String", unless you named it `strStrange`, or `strAnge`.

3. This feature makes it easy to work with complex expressions, function parameters, and `ForEach` iteration variables, as sVB can't infer their types directly.
4. Naming convention overrides the type inference. For example:



5. As you see in the above picture, when you move the caret to the identifier, the intellisense shows you info about the identifier including its inferred type.
6. Intellisense also can offer auto completion for some data types like color, key and dialog results. For example, when you assign a value for a color variable, the auto completion list suggests the "Colors" class to choose a color from it's members.



Literal Expressions

Literal expressions are constant values that you can assign to variables, send as arguments to subroutines, or use as operands in arithmetic or logical operations.

For example:

```
Name = "Adam"           ' A string literal
Age = 13                ' A numeric literal
BirthDate = #1/1/2010#   ' A date literal
```

sVB has 5 types of literal expressions:

1. Numeric literals.
2. String literals.
3. Date and time literals.
4. Time span literals.
5. The **True** and **False** keywords that are equivalent to the "True" and "False" string literals respectively.

Let's take a deeper look at these expressions:

1. Numeric literals:

These are the familiar numbers you use daily. They can be positive or negative, and can be integers or decimals. For example:

```
X = 1
Y = -2
Z = 3.5
W = -0.06
```

Note that sVB will infer the X, Y, Z and W variables in the above code to be of the type Double, because they are initialized with numeric values. A double variable can access the methods of the [Math type](#) directly like this:

```
TextWindow.WriteLine(Y.Abs) ' 2
```

The above code is a short form to this equivalent code:

```
TextWindow.WriteLine(Math.Abs(Y))
```

2. String literals:

A string literal is any text of any length enclosed in double quotes like "Hello sVB", which represents the string value **Hello sVB** without the quotes.

You can even use just the double quotes without anything in between ("") to represent an empty string (or nothing).

Any sVB keyword or symbol written inside the double quotes will be treated as a normal text, like:

```
x = "If you say 'Hello', I'll say 'Hello'."
```

In the above string literal, the comment character ' used 5 times, and it doesn't treated as a comment. A dot is also used as a normal character, and the word If is not considered to be the sVB **If** keyword.

The same goes for the line continuity character _, which means you can't use it to split a single string literal over 2 lines, instead you can write two separate string literals and use the + operator to join them, which allows you to use the _ symbol in between:

```
x = "If you say 'Hello', " + _  
     "I'll say 'Hello'."
```

In fact, you don't need the _ in the above sample, because there is an implicit line continuity around the + operator:

```
x = "If you say 'Hello', " +  
     "I'll say 'Hello'."
```

But, what if you want to use a quote character inside the string? If you did so, the string literal will end at the first quote found in the text, and the rest of the string will seem to sVB as a normal syntax (which will be a wrong syntax of course).

To avoid this issue, you must escape it by using an extra quote like "'''", which represents a single quote, as the first and last quotes are a part of the syntax not a part of the string value, and the second quote are an escaping character to tell the sVB

compiler that the next quote is a part of the string value not an enclosing quote! For example:

```
Msg = "Let's say ""Hello"" to sVB."  
TextWindow.WriteLine(Msg)
```

```
Let's say "Hello" to sVB.
```

Note that sVB will infer the Msg variable in the above code to be of the type String, because it is initialized with a string value. A string variable can access the methods of the [Text type](#) directly like this:

```
TextWindow.WriteLine(Msg.UpperCase)
```

The above code is a short form to this equivalent code:

```
TextWindow.WriteLine(Text.ToUpper(Msg))
```

Take Care:

Although sVB is case-insensitive, the string literals are case sensitive, which means that "test" is not equal to "Test".

So if you want to do a case-insensitive comparison between two strings, you should compare a normalized versions of the two strings (by converting both of them to the lower case). For example:

```
X = "test"  
Y = "Test"  
TextWindow.WriteLine({  
    X = Y,                      ' False  
    X.LowerCase = Y.LowerCase,   ' True  
    X.UpperCase = Y.UpperCase   ' True  
})
```

3. Date and time literals:

You can enclose a string that represents a date or a time or both in a pair of hashes like #1/31/2023#. The date must be formatted with the English culture not your local culture, so your code can run exactly the same on any PC in the planet.

This means that when you use the numeric date format, you must start with the month then the day then the year, like in the above example.

Note:

When you move the caret to the date literal, the code editor will show a tip window that contains the date representation in both the English culture and your PC's local culture. The following picture shows how this looks on my PC with the Arabic-Egypt culture:

D = #1/31/2024#

Date Literal: #1/31/2024#

Value = "31/01/2024 12:00:00 ظ"

And if the date is invalid, the tip window will tell you that:

D = #31/1/2024#

Date Literal: #31/1/2024#

Invalid date format!

sVB date literals are flexible, so you can use full or abbreviated month names and different part separators like /, \, - and spaces. In this case, the position of the month doesn't matter and you can start with the day like #31 Jan 2023#, because the month name resolves any ambiguity.

You can represent the time part with the 24 hours system or use the 12 hours system with the day time part (AM/PM). The following examples show you some of the valid date literal formats:

```
TextWindow.WriteLine({
    #1/31/23|,
    #15:00:01|,
    #03:00:01 PM|,
    #06-02-2022 1:0:0|,
    #11 April 2023|,
    #10 oct 2020|
})
```

And this is a picture of the results:

```
01/31/2023 12:00:00 AM
04/16/2023 03:00:01 PM
04/16/2023 03:00:01 PM
06/02/2022 01:00:00 AM
04/11/2023 12:00:00 AM
10/10/2020 12:00:00 AM
```

Note that when you don't provide the time part, 12:00:00 AM will be used by default, and when you don't provide the date part, the current date of your system will be used, and you should be careful in this case, because such a date will be changed every day, and you should use the [Date type](#) methods get rid of the date part before using or displaying it. But there is another solution, which is using the 1/1/0001 date with such times, to make sVB ignore the date part:

```
D = #1/1/0001 15:00:01#
TextWindow.WriteLine(D) ' 3:00:01 PM
```

Note that the date 1/1/1 means 1/1/2001 not 1/1/0001.

Note that sVB will infer the D variable in the above code to be of the type Date, because it is initialized with a date value. A date variable can access the methods of the [Date type](#) directly like this:

```
TextWindow.WriteLine(D.EnglishDayName) ' Monday
```

The above code is a short form to this equivalent code:

```
TextWindow.WriteLine(Date.GetEnglishDayName(D))
```

4. Time span literals:

A duration (or time span) is a difference between two dates, so it can be positive or negative.

A duration can contain only days, hours, minutes, seconds, and milliseconds. So, if the duration contains months or years, they must be expressed in total days. This is because the number of days of the month depend on what is the month and what calendar you are using. Also the number of days of the year depends on the calendar, and even in the Gregorian calendar, the year can have 265 or 366 years because there is a leap year every 4 years! So, days are the only reliable part to be used in the time span.

sVB supports time span literals, and they also use the double hashes, similar to the date and time literal, but the time span literal must start with a + or - sign to differentiate it from the time literal, like #+10:12:00#. Also, the days part are separated from the hours part by a dot, like #-1.0:0:1.500#. For example:

```
T1 = #+2.3:0#
T2 = #-1.1:0:30#
D = #1/1/2023 23:00:30#
TextWindow.WriteLine({
    T1,
    T2,
    T1 + T2,
    T1 - T2,
    T2 - T1,
    D + T1,
    D - T1,
    D + T2,
    D - T2,
    Date.Now - D
})
```

```
2.03:00:00
-1.01:00:30
1.01:59:30
3.04:00:30
-3.04:00:30
01/04/2023 02:00:30 AM
12/30/2022 08:00:30 PM
12/31/2022 10:00:00 PM
01/03/2023 12:01:00 AM
104.16:07:52.9783765
```

Note that sVB will infer the T1 and T2 variables in the above code to be of the type Date, because it is initialized with a time span value. A date variable can access the methods of the [Date type](#) directly like this:

```
TextWindow.WriteLine(T1.TotalHours) ' 51
```

The above code is a short form to this equivalent code:

```
TextWindow.WriteLine(Date.GetTotalHours(T1))
```

5. The True and False keywords:

True is equivalent to the string literal "True", and **False** is equivalent to the string literal "False" string literals respectively. They can be used in logical comparisons and to set the value of Boolean variables. For example:

```
CanFly = False
CanSwim = True
If CanSwim = False Then
    TextWindow.WriteLine("Can't swim")
EndIf
```

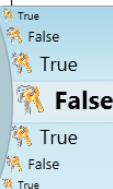
Note that sVB will infer the CanFly and CanSwim variables in the above code to be of the type Boolean, because they are initialized with numeric values. This will allow the code editor to suggest True and False in the auto-completion list when you write = after the name of the variable.

Form1 *

(Global)

```
1 CanFly = False
2 CanSwim = True
3 If CanSwim = Then
4   TextWind
5 EndIf
6
7
```

False



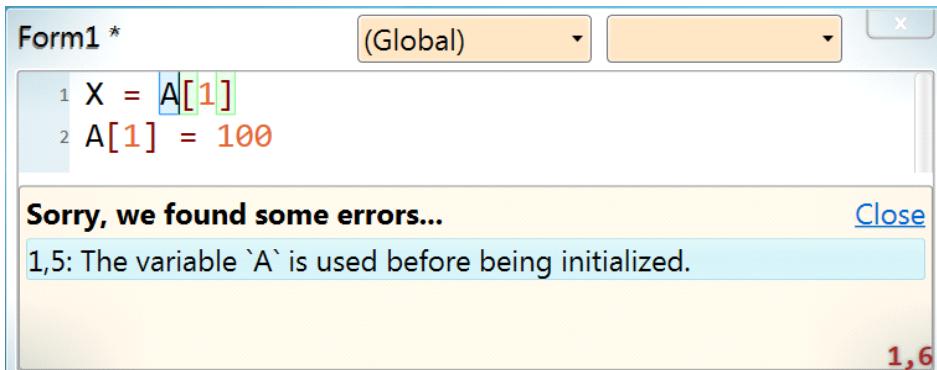
Array Expressions

You can use the array expression to get or set a value of an item of the array.

The array expression consists of an identifier (the array name) followed by the indexer, which is a pair of brackets [] containing the index of the array, like names[1].

Note that:

- When you use the indexer to set a value at a new index, a new item will be added to the array at this index.
- If the array doesn't exist, it will be created the first time you set an item in it. But if you try to read an item from the array before creating it, you will get an error.



- If you use the indexer to set a value at an already existing index, this will change the value of the item existing at this index.
- The index of the array can be a numeric literal like 1, 2, .. etc.

For example:

```
A[1] = 10 ' Create the array and add the first Item
A[2] = 15 ' Add the second item
A[2] = 20 ' Modify the second item
A[3] = 30 ' Add the thirs item
A[4] = 40 ' Add the forth item
A[5] = 50 ' Add the fifth item
```

- The index can also be a string literal key like "id". The array key is case-insensitive, so "ID", "Id" and "id" are actually the same key. For example:

```
Student["name"] = "Adam"
Student["age"] = 18
Student["sex"] = "male"
TextWindow.WriteLine({
    Student["Name"],           ' Adam
    Student["AGE"],           ' 18
    Student["sex"]            ' male
})
```

- If the string key is a valid [identifier](#) (like in the above example), sVB allows you to use the dictionary lookup operator **!** to access it directly:

```
TextWindow.WriteLine({
    Student!Name,
    Student!Age,
    Student!sex
})
```

The key syntax is verbose but more flexible, as you can use any string (like "my name" or ";") as a key.

- You can also use the dictionary lookup operator to add the items:

```
Student!Name = "Adam"
Student!Age = 18
Student!Sex = "male"
```

In this case, the Student array looks like a dynamic object (Exando Object in VB.NET), with three [dynamic properties](#) Name, Age and Sex.

- It is possible to use 0 or even a negative number as an index! By default, sVB array index starts at 1 and this is what you should expect when you get an array from any method of the sVB library types, but since you can use string keys as indexers, it is valid to use "0" and "-1" as keys, and since sVB

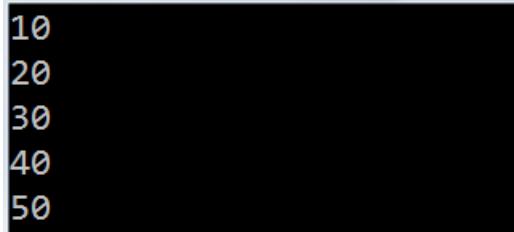
does implicit conversion between numbers and strings, you can directly use 0 and -1!

for example:

```
X["-1"] = "Test"  
TextWindow.WriteLine(X[-1]) ' Test
```

- You can use a variable or any expression to get the index or the key of the item. This is what we usually do in for loops:

```
For I = 1 To 5  
    TextWindow.WriteLine(A[I])  
Next
```



```
10  
20  
30  
40  
50
```

- You can use the = operator to check the equality of two arrays, where they are considered equal if they have the same items count, with each key has the same value in each of them. For example:

```
X["one"] = 1  
X["two"] = 2  
Y["Two"] = 2  
Y["One"] = 1  
A = {1, 2}  
B = {2, 1}  
C = {1, 2}  
  
TextWindow.WriteLine({  
    X = Y,      ' True  
    X = A,      ' False  
    Y = A,      ' False  
    A = B,      ' False  
    A = C      ' True  
})
```

- sVB infers the array expression to be of the Array type. This allows you to use the a variable as an array object, and access the [Array type](#) methods directly from it, like:

```
TextWindow.WriteLine(A.Count) ' 0
```

Which is a short form of this equivalent code:

```
TextWindow.WriteLine(Array.GetItemCount(A))
```

- You can implicitly split the line after the left bracket [and before the right bracket]. For example:

```
X[  
    Listbox1.SelectedIndex  
] = Listbox1.SelectedItem
```

* An array or a Dictionary?

The fact that the array is accessed by keys, means it is actually a dictionary, and it can contain gaps, so if you try to read a value of an item that doesn't exist, the array expression will return an empty string. But some existing items may actually contain an empty string, so how to distinguish between the two cases? The solution is to use the [Array.ContainsIndex](#) method to know whether the item exists or not before trying to read it.

The array also can have unordered indexes, like:

```
A[5] = 1  
A[1] = 2  
A[2] = 3
```

So, when you call [Array.GetAllIndices](#) method or the [Array.Indices](#) property to get an array containing all the keys of the array, you will get the indexes by the order you defined them. The following code will print {5, 1, 2}:

```
TextWindow.WriteLine(Array.ToString(A.Indices))
```

So, you can use that index array to write a for loop to read all the array items!

That behavior is inherited from Small Basic, but sVB makes it a little easier on you, by adding the [Array.GetKeyAt](#) method, but the easiest way is to use the [ForEach loop](#) to walk through all the array items regardless of their keys. For example:

```
ForEach Item In A
    TextWindow.WriteLine(Item)
Next
```

* Multi-dimensional array:

You can use more than one indexer after the array names, each of them deal with the corresponding sub array. For example, the following array contains the data of 3 students, where each student data is a sub array that contains 3 items (his name, age and sex):

```
Student[1]["name"] = "Adam"
Student[1]["age"] = 18
Student[1]["sex"] = "male"
Student[2]["name"] = "John"
Student[2]["age"] = 17
Student[2]["sex"] = "male"
Student[3]["name"] = "Nora"
Student[3]["age"] = 17
Student[3]["sex"] = "female"
```

You can use two nested loops to show all the students data like this:

```
For I = 1 To 3
    ForEach Item In Student[I]
        TextWindow.WriteLine(Item)
    Next
    TextWindow.WriteLine("")
Next
```

```
Adam  
18  
male  
  
John  
17  
male  
  
Nora  
17  
female
```

* Array Performance:

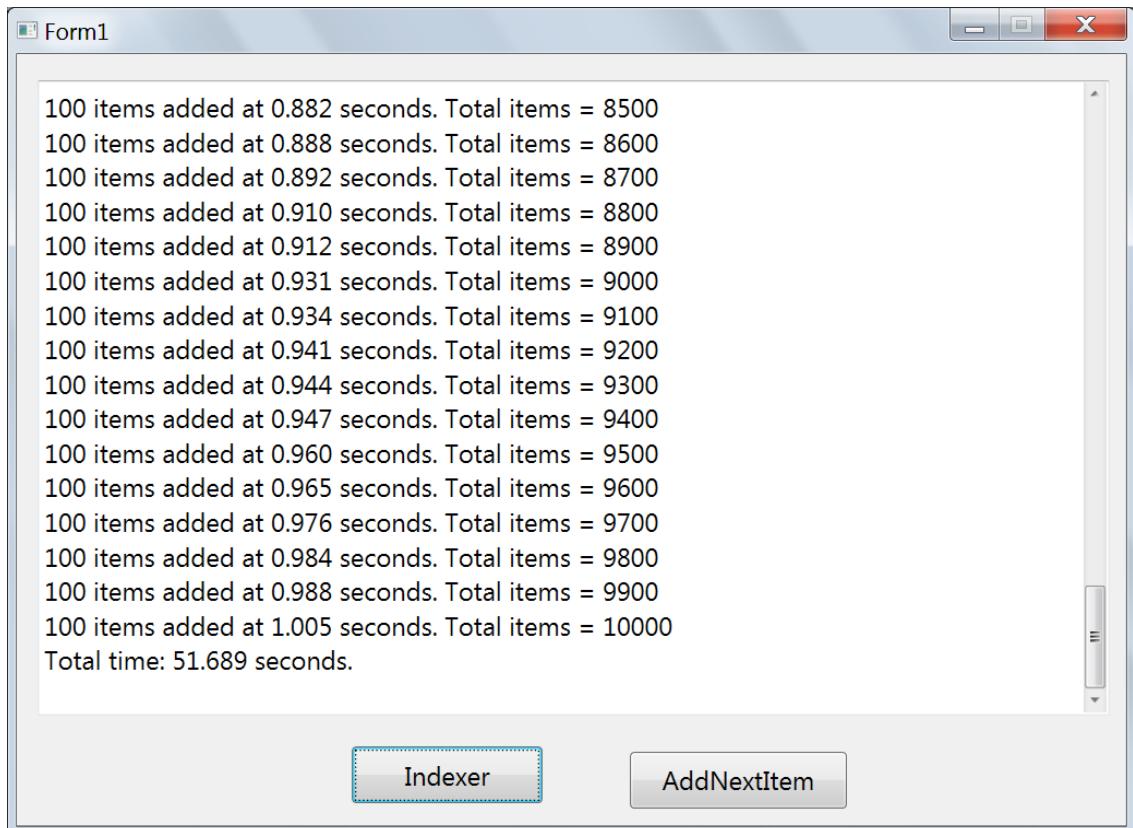
To be easy for kids and beginners as an educational programming language, the Small Basic compiler and its type system were designed for simplicity not for best performance. The array type is an obvious example on that. Each time you use the array indexer to add an item to the array or change the value of an existing item, a new array is created, and the old array values are copied to it then the change is applied on the new array!

This is of course the worst performance ever regarding the memory consumption and the application speed, but it shouldn't be that bad when you deal with a small array, where in most cases you don't expect it to exceed 20 items.

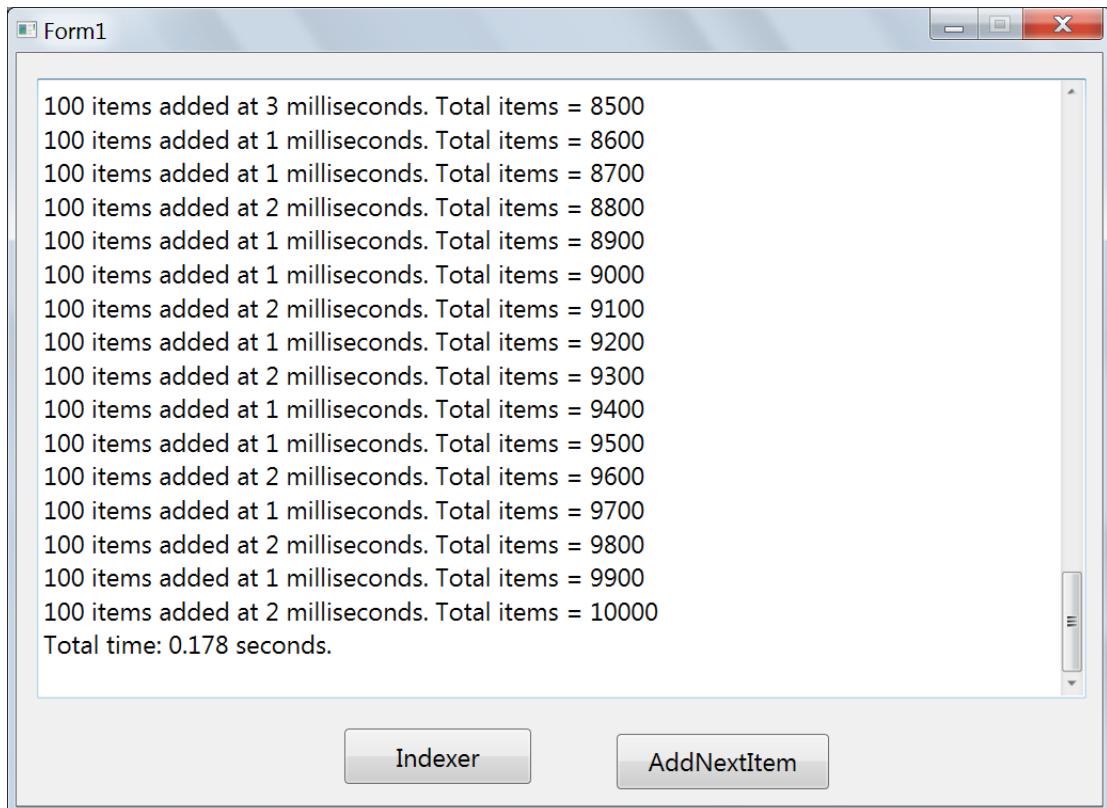
But the performance impact will increase exponentially with every new element you add, because of copying all the previously added items first!

To see this by your own eyes, Open the "Array performance" project in the samples folder, press F5 to run the project, and click the Indexer button. This button adds 10,000 items to the array, which will take a while. The textbox will be updated every

100 items, to show you the time used to add them, where you will notice that every 100 items takes a longer time, where the last 100 items will be about 54 times longer than the first 100 items! The total time will be show after the process completes. In my old PC, this will take about 52 seconds!



The good new is that sVB provided a solution for this, by using the [Array.Append](#) and [Array.SetItemAt](#) methods. These two methods add and modifies the array items by reference, without having to create a new array and copy the old one to it. This is super fast, and you can try that by clicking the SetItemAt button in the "Array performance", which adds the 10,000 items to the array in less that 0.2 of a second on my old PC (compare that to the 52 seconds to add them by indexer!).



You may ask me here: Why don't sVB use the [Array.Append](#) and [Array.SetItemAt](#) methods to execute the indexer directly?

Unfortunately, it is not that easy! By reference operations can be dangerous, because they can modify another array while you are not paying attention!

This can happen if you got array by one of these ways:

1. Copying an array using the = operator.
2. Sending an array to a function parameter
3. Receiving an array from a function return value.

The array you got by theses ways is just a reference to the original array, and this is why SB creates a new array before adding or modifying any item. But when you use the [Array.Append](#) and [Array.SetItemAt](#) methods to change the array, the change will happen on your array and the original array

because they have the same reference (same storage space in memory)! For example:

```
X = {1, 2, 3}  
Y = X  
X.Append(4)  
Y[5] = 5  
Y.Append(6)  
TextWindow.WriteLine({X.ToString(), Y.ToString()})
```

```
{1, 2, 3, 4}  
{1, 2, 3, 4, 5, 6}
```

It is obvious that adding the 4th item using the Y.Append method appeared in both arrays, because executing the Y = X statement made X and Y have the same reference (point to the same array in memory), while using the indexer to add the 5th item created a new array and added the new item to it. After that, using the Y.Append method will has no effect on X, because Y points to a different array.

I think you realize now how thins can be confusing!

My advice here is to use the Array.Append and Array.SetItemAt methods on fresh independent array you create yourself, and as a simple safe guard, always make sure you use this line:

```
a = {}
```

before you start using a.Append and a.SetItemAt.

As you can see, by reference operation are not something to introduce to kinds nor beginners, and this is why sVB kept the array indexer behavior inherited from SB, as it is suitable for small samples and lightweight applications where performance is not an issue.

Array Initializer Expressions

You can use a pair of braces {} to set multiple elements to the array at once:

```
x = {1, 2, 3}
```

Nested initializers are also supported when you deal with multi-dimensional arrays (jagged arrays):

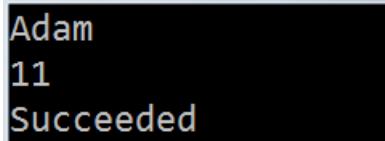
```
y = {"a", "b", {1, 2, 3}}
```

You can also use variables or function return values inside the initializer. For example, the above code can be rewritten as:

```
Y = {"a", "b", X}
```

And you can send an initializer as an argument to a function:

```
TextWindow.WriteLine({"Adam", 11, "Succeeded"})
```



```
Adam  
11  
Succeeded
```

Note that assigning an array initializer to a variable makes sVB infer it to be of the Array type. This is why you may sometimes need to use an empty initializer to create an empty array:

```
a = {}
```

This allows you to use the a variable as an array object, and access the [Array type](#) methods directly from it like:

```
TextWindow.WriteLine(A.Count) ' 0
```

Which is a short form of this equivalent code:

```
TextWindow.WriteLine(Array.GetItemCount(A))
```

Note also that you can make use of the implicit line continuity to format the array initializer in a pretty readable way:

```
Y = {  
    "a",  
    "b",  
    {1, 2, 3}  
}
```

or:

```
Y = {  
    "a", "b",  
    {1, 2, 3}  
}
```

or:

```
Y = {  
    "a",  
    "b", {  
        1,  
        2,  
        3  
    }  
}
```

or:

```
Y = {  
    "a",  
    "b",  
    {  
        1,  
        2,  
        3  
    }  
}
```

Where you have the freedom whether or not to split the line at these places:

- after any opening brace {,
- after any comma,
- and before any closing brace },

Just split the line at a valid place, and the sVB code editor will take care of line indentations for you.

Binary Operator Expressions

A binary operator operates on two (left and right) operands. The following table summarizes the sVB binary operators, where we will use x and y as two operands, given that $x = 0$ and $y = 1$:

Operator	Operation	Example	Result
<code>+</code>	Addition	$x + y$	1
<code>&</code>	Concatination	$x & y$	01
<code>-</code>	Subtraction	$x - y$	-1
<code>*</code>	Multiplication	$x * y$	0
<code>/</code>	Division	x / y	0
<code>Mod</code>	Remainder	$10 \text{ Mod } 3$	1
<code>=</code>	Equality	$x = y$	False
<code><></code>	Inequality	$x <> y$	True
<code>></code>	Greater than comparison	$x > y$	False
<code>>=</code>	Greater than or equals to comparison	$x >= y$	False
<code><</code>	Less than comparison	$x < y$	True
<code><=</code>	Less than or equals to comparison	$x <= y$	True
<code>And</code>	Logical And	$x \text{ And } y$	False
<code>Or</code>	Logical Or	$x \text{ Or } y$	True

Notes:

1. You can cascade binary operators like:

$x + y + (z + 1) * 2$

The compiler will perform these operations in this order:

- a. Operations between quotes.
- b. `*` and `/`.
- c. `+` and `-`.

2. The + operator can be also used to join two string together. The following example makes the label display the word "Hello " followed by the name the user enters in the textbox, followed by a full stop:

```
Label1.Text = "Hello " + TextBox1.Text + "."
```

So, what result do you expect from this code:

```
TextWindow.WriteLine("1" + "2")
```

Will it be "12" or 3?

If you tried it, you will get 3, because the + operator tries to convert each of the operands to a number to be able to add them arithmetically, but if the conversion of any of them failed, it will convert each of them to a string and join them into one string. If this is not the result you expect, you can use the Text.Append method to join two strings regardless of their content. The following code will print 12:

```
TextWindow.WriteLine(Text.Append("1", "2"))
```

In fact, the string concatenation operator & can give you the same result of the above code. It always concatenate its operands, even if they are two number. The following code will print 12:

```
TextWindow.WriteLine(1 & 2)
```

3. The * operator can be also used to duplicate a string. To do that, one of its operands must be a string that can't be converted to a number, while the other operand is a number or a string that can be converted to a number. For example:

```
TextWindow.WriteLine("{*} " * 10)
```

The above code will ill print:

```
{*} {*} {*} {*} {*} {*} {*} {*} {*} {*} {*}
```

4. The Mod operator returnms the same result as the [Math.Remainder](#) method. See the notes and examples provided there.

5. You can also use the **%** operator to get the remainder, but it will be converted by the code editor to Mod as soon as you leave the line. So, it is actually just a shortcut.
6. When you use **And** and **Or** together, you should enclose some operations with parentheses to tell sVB in what order it should perform these operations. For example, the expression:

(X > 0 Or Y > 3) And Z < 7

is different than this one:

X > 0 Or (Y > 3 And Z < 7)

7. The positions before and after a binary operator is a valid implicit line continuity position. In other words, you can split the line before or after any operator. You can also add comments at the end of any line segment. For example:

```
X = 0
Y = 1
If X = Y Or           ' split after Or
    Text.GetText("some text", 6, 3)
        <> "abc" Then      ' split before <>
    TextWindow.WriteLine("OK")
EndIf
```

Negate Expressions

A negation expression applies on a single operand, to negate its value, like `-x`, which is the short form of the binary operation `0 -x`. The negated operand can be any valid expression, like a function or a method call, or any quoted operation. For example:

```
X = -Math.Decimal("A")          ' - 10
Y = -X                          ' 10
TextWindow.WriteLine(-(X - Y))   ' 20
```

Function and Method Call Expressions

A function call expression consists of the function name followed by its argument list, in the form:

Func(p1, p2, p3, ..., pn)

You must send the exact number of arguments that the function expect, no more, no less, and if the function has no parameters, the argument list is empty so you must use an empty parentheses after the function name on the form:

Func()

The function expression returns the value of the function, so it can be assigned to a variable, used as an operand in binary expressions, or passed as a parameter. The following example shows you two ways to use a call expression to a function named Sum:

```
X = Sum(4, 2)
TextWindow.WriteLine({
    X,
    Sum(1, 2)
})
Function Sum(a, b)
    Return a + b
EndFunction
```

A methods is also a function, but it belongs to a type, so you must mention its type before its name on the form:

Type.Method(p1, p2, p3, ..., pn)

And the same rules of function call expressions are applied on method call expressions.

The following example shows you two ways to use a call expression to the Text.GetLength method:

```
X = Text.GetLength("abc") + 1
TextWindow.WriteLine({
    X,
    Text.GetLength("abcd")
})
```

Note:

You can use the line continuity character _ to split a code line at any position except before or after the dot "." used in method call expressions.

Property Expressions

You can think of a property as a variable that belongs to a type, so to access such a variable (property) to set or read its value, you must mention its type before its name on the form:

Type.Property

The following example shows you different ways to use the TextBox.Text property:

```
TextBox1.Text = 100  
If Text.IsNumeric(TextBox1.Text) Then  
    N = 1 + TextBox1.Text  
EndIf  
TextWindow.WriteLine({N, TextBox1.Text})
```

Note:

You can use the line continuity character _ to split a code line at any position except before or after the dot "." used in property expressions.

*** Handling Events:**

The property expression can be used to set the handler of an event, in the Form:

Control.Event = Handler

You can also remove the event handler, by setting it to Nothing:

Control.Event = Nothing

Note that all event names of sVB controls starts with the On prefix like Button.OnClick, except the Timer.Tick because it is inherited from Small Basic.

The handler is a name of a subroutine (without parentheses). Functions can't be used as event handlers.

For example, this code sets the Click subroutine as the handler for the From.OnClick event:

```
Me.OnClick = Click
Sub Click()
    Me.Text = Text.Format(
        "([1],[2])", {Me.MouseX, Me.MouseY}
    )
EndSub
```

For more information about available events and how to use them, read the [handling control events](#) section.

Dynamic Property Expressions

sVB supports using the array as a dynamic object, that you can add dynamic properties to.

There are two different ways to do that:

1. Using the Data naming convention:

You can add the word "Data" as a prefix or a suffix to any identifier to treat it as a dynamic object, so you can add properties to it using the dot notation as if it is a normal property expression. For example, you can create a car object that has Color and Speed properties like this:

```
CarData.Color = Colors.Red  
CarData.Speed = 100  
TextWindow.WriteLine({  
    CarData.Speed,  
    CarData.Color  
})
```

In fact, sVB converts the above syntax to:

```
CarData["Color"] = Colors.Red  
CarData["Speed"] = 100  
TextWindow.WriteLine({  
    CarData["Speed"],  
    CarData["Color"]  
})
```

You can use both, but the using dynamic object has the following advantages:

- a. It is shorter and more readable.
- b. The intellisense offers to auto-complete the dynamic properties names, which prevents you from mistyping them, as could happen while writing string array keys without any editor support.

```

Form1 *
(Global) ▾
(5,15)

1 CarData.Color = Colors.Red
2 CarData.Speed = 100
3 TextWindow.WriteLine({
4     CarData.Speed,
5     CarData.
6 })
7

```

The tooltip displays the properties of the `CarData` object:

- Speed
- Color
- Speed
- Color**
- Speed
- Color
- Speed

- c. A dynamic object can inherit property names from other dynamic object if it contains its name (after trimming the "Data" part from both). For example, the `Car2Data` and `myCarData` objects will show the `Color` and `Speed` properties (inherited from `CarData` object) in their auto-completion list":

```

Form1 *
(Global) ▾
(9,10)

6 }
7
8 Car2Data.Acceleration = 10
9 Car2Data.
10
11
12

```

The tooltip displays the properties of the `Car2Data` object:

- Speed
- Acceleration
- Color
- Speed**
- Acceleration
- Color
- Speed

Also, the `MyCar2Data` object, will inherit all properties from `MyCarData`, `Car2Data` and `CarData`!

2. Using the dictionary lookup operator:

Adding the `Data` word to variable names makes them longer, and you may dislike using them, so, sVB offers you an alternative way to define dynamic properties, using the `!` operator (which is called the dictionary lookup operator in VB.NET). So, the `CarData` example can just be rewritten as:

```

Car!Color = Colors.Red
Car!Speed = 100
TextWindow.WriteLine({
    Car!Speed,
    Car!Color
})
Car2!Acceleration = 10
Car2!Speed = 200

```

This is a nicely shorter code, but you may take some time to be used to writing ! instead of the dot.

* Dynamic Properties Domains:

When inheriting dynamic properties names, sVB ignores variable domain rules, to allow you to reuse the properties across subroutines and functions.

This is totally safe as it only affects the property names that will appear in the auto completion list, but it has no effect on the values of the variables in runtime.

This means that the auto-completion list may offer you property names from a dynamic object from another function, but if you read these properties values in code they will return empty strings regardless of their values in the other function, and they will not have any values unless you set them in the current function. So, the two dynamic objects are still two different local variables, despite they use similar key names!

For example, when you run this code, the Car!Color property in the Car1 subroutine will show a value, while the same property in the Car2 subroutine will show an empty sting:

```
Car1()
Car2()

Sub Car1()
    car!Color = Colors.Red
    car!Speed = 100
    TextWindow.WriteLine({
        "car1 speed = " + car!Speed,
        "car1 color = " + car!Color
    })
EndSub

Sub Car2()
    car!Speed = 200
    TextWindow.WriteLine({
        "car2 speed = " + car!Speed,
        "car2 color = " + car!Color
    })
EndSub
```

```
car1 speed = 100
car1 color = #FFFF0000
car2 speed = 200
car2 color =
```

Assignment statements

The assignment statements uses the = operator to set the left hand expression to the value of the right hand expression, in the form:

Left hand expression = right hand expression

Where the right hand expression can be any valid expression, but the left hand expression can be one of these expressions only:

1. Identifier expression:

The left hand of the assignment statement can be an identifier, which allows you to define new variables, or change the value of existing ones. For example:

X = 1	' Declare a new variable X
Y = X + 2	' Declare a new variable Y
X = 0	' Change the value of X

sVB can infer the variable type from its initial value, so X and Y are inferred as Double in the above example.

Note that the variable name can't appear on both sides in its declaration line. This code will give you a compilation error:

Z = Z + 1

But it is ok to use the variable name on both sides after it is initialized. This code will compile correctly:

Z = 2
Z = Z + 1

The last line may be confusing, because it looks like an impossible mathematic equation, as Z can never equal z+1!

The confusion is cause because the = symbol is used in BASIC languages to do two functions: the assignment and logical comparison, and here we are using it for the first.

To understand how it works, you should know that the right hand side will be evaluated first, then the value will be

assigned to the left hand side. So, you can read the above code as:

The new value of z will equal the old value of z plus 1

And at runtime, the code will seem like this after substituting z by 2 :

`z = 2 + 1`

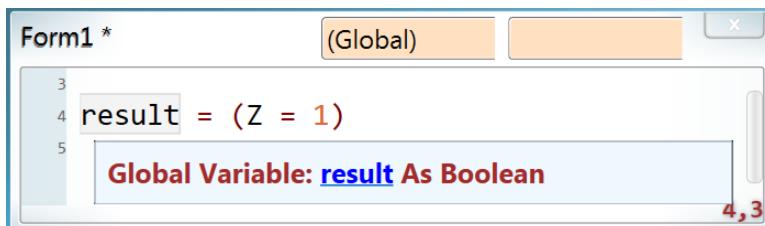
Look also at this statement:

`result = z = 1`

Here, we have two = symbols in the same statement. As a rule, the first one only can be the assignment operator, so, the next one must be the equality operator. So, the above statement can rewritten as:

`result = (z = 1)`

Where the right hand expression is a logical expression that will return True or False, and this is why sVB infers the result variable to be of the Boolean type:



You can also create an array, by assigning an array initializer to its identifier:

`a = {1, 2, 3}`

And you can copy the above array to another one:

`b = a`

A Small Basic Breaking change:

this funny code compiles in SB:

`x = y`

`y = 1`

Where y is considered declared because it is assigned in the second line, and its value will be "" in the first line! This will not compile anymore in sVB, because it doesn't allow you to read a variable before you initialized it.

2. Array expression:

The left hand of the assignment statement can be an array expression, which allows you to define new arrays, or change the value of existing ones. For example:

```
' Define the array and set the first item
A[1] = 1
A[2] = 2 ' Set the second item
A[1] = 1 ' Change the first item
A[3] = A[1] + A[2]
B[1][1] = 5
```

3. Property expression:

You can use the assignment statement to set the value of a property like this:

```
TextBox1.Text = "Hello sVB"
```

4. Dynamic property expression:

The left hand of the assignment statement can be a dynamic property expression, which allows you to define new dynamic properties, or change the value of existing ones. For example:

```
P!X = 0
P!Y = P!X + 1
P!X = 100
```

* Inline Documentation comments:

The sVB intellisense will use any comment you provide at the end of a variable line as its documentation info, that will be shown in the auto-completion list and in the popup help window:

Form1 *

(Global)

```

3
4 X = 0 'The horizontal position
5
6 ' Global Variable: X As Double
7 Y The horizontal position

```

4,1

And the same works with the dynamic property declaration::

Form1 *

(Global)

```

1 Student!ID = 1 'The student number
2
3 TextWindow.WriteLine(Student!ID)

```

Dynamic Property: Student!ID As Double

The student number

3,31

In the above picture, you see that the popup help window shows the comment provided for the ID dynamic property, and its name appears in as a hyper link, so you can click it to go to the its declaration line.

You can also provide more documentation comment lines above the variable or the dynamic property declaration line, and if you add a comment at the end of the declaration line, it will be appended to them:

Form1 *

(Global)

```

1 ' The student name.
2 ' It can include first and last names.
3 Student!Name = "Ali" 'An empty name is not allowed.
4
5 TextWindow.WriteLine(Student!Name)

```

Dynamic Property: Student!Name As String

The student name.
It can include first and last names.
An empty name is not allowed.

31

* Global and Local Variables:

Small Basic doesn't have variable scopes, as all variables are considered global, and you can define them in any place in the file (inside or outside subroutines) and use them from any other place in the file (up or down).

Small Visual Basic has cleaned this mess, which is a break change that can prevent some SB code from running probably in sVB, but it is a necessary step to make kids and beginners organize their code and write clean codes.

This is also necessary to make sub and function parameters work correctly, and allow you to use recursive subs and functions.

These are the new scope rules applied by sVB:

- Variables defined outside subroutines and functions are global variables.
- Subroutine and function parameters are local, and hide any global variables with the same names.
- The For loop counter (iterator) in any subroutine or function is local and hides any global variable with the same name.
- A variables defined inside any subroutine or function is local, unless there is a global variable with the same name defined above of this subroutine or function. But if the global variable is defined below, then the local variable will hide it.

So, as a good practice, follow these guidelines:

- Define all global variables at the very top of the file.
- Give global variables a prefix (such as "g_") to avoid any possible conflicts with local variables.
- Don't use global variables unless there is no other solution. Instead, pass values to subroutines and functions through parameters, and receive values from functions through their return values.

Label and Goto Statements

The label statement is very simple. It is just an identifier followed by a colon:

Label1:

and ifs function is simpler, as it just marks a line to allow you to jump to using the GoTo statement:

Goto Label1

If fact, the Goto statement is less important in sVB, than SB, because sVB has more safe jump commands like Return, ExitLoop, and ContinueLoop, while the GoTo command is the only way to do such jumps in SB! So, you may never use GoTo in sVB!

* Notes:

1. The label is local to the subroutine or function it is defined in. This means you can't jump out of any subroutine or function.
2. You can define a label in the global area of the file (outside subroutines and function), but you still can't jump to them from a subroutine nor a function.
3. You can't define a local label with the same name of a global label!
4. Don't jump into the middle of a for loop, otherwise you will get unexpected results.
5. You can jump up or down, but be careful, because jumping up can cause the program to enter into an infinite loop, unless you provide an exit condition. For example, this code uses GoTo to execute a for loop:

```
Start = 1
_To = 10
_Step = 2
I = Start

_Next:
If I >= _To Then
    Goto _ExitLoop
EndIf

TextWindow.WriteLine(I)

I = I + _Step
Goto _Next
```

_ExitLoop:

In fact sVB compiler translates the For loop to a code similar to the above (with some extra code to handle the negative step loops).

So, Goto is a low level instruction, and it is better to leave it for the compiler, and try to avoid using it yourself unless you have to.

ExitLoop Statements

Exits the For and While loops. It has 3 forms:

1. ExitLoop

Exits the inner loop that contains it.

2. ExitLoop -

Exits the inner loop that contains it and exits its parent loop. The minus sign means go back (out) one more level. You can use -- to exit two parent loops, and you can add more minus signs as you need, but practically, you will not have more than three nested loops, and if you wrote four and want to exit them all, the next form will do it.

3. ExitLoop *

Exits all nested loops whatever their count.

For example:

```
For I = 1 To 3
    For J = 1 To 3
        K = 1
        While K <= 4
            For N = 1 To 5
                TextWindow.WriteLine(Text.Format(
                    "([1],[2],[3],[4])",
                    {I, J, K, N}
                ))
                ExitLoop
            Next
            K = K + 1
        Wend
    Next
Next
```

Run this code and see the results. Try it 3 more times after changing the ExitLoop command to ExitLoop-, ExitLoop-- and ExitLoop* respectively, and compare the results.

ContinueLoop Statements

Exits the current iteration and continues the next one of the For and While loops. It has 3 forms:

1. ContinueLoop

Continues the next iteration of the inner loop that contains it.

2. ContinueLoop -

Exits the inner loop that contains it and Continues the next iteration of its parent loop. The minus sign means go back (out) one more level. You can use -- to continue two parent loops, and you can add more minus signs as you need, but practically, you will not have more than three nested loops, and if you wrote four and want to continue them all, the next form will do it.

3. ContinueLoop *

Continues the next iteration of most outer loop whatever the count of the nested loops.

For example:

```
For I = 1 To 3
    For J = 1 To 3
        K = 1
        While K <= 4
            For N = 1 To 5
                TextWindow.WriteLine(Text.Format(
                    "([1],[2],[3],[4])",
                    {I, J, K, N}
                ))
                ContinueLoop
            Next
            K = K + 1
        Wend
    Next
Next
```

Run the above code and see the results. Try it 3 more times after changing the ContinueLoop command to ContinueLoop-, ContinueLoop-- and ContinueLoop* respectively, and compare the results.

Note that using ContinueLoop- will make it an infinite loop, because it skips the inner for loop and jumps to the start of the while loop, without executing the $K = K + 1$ statement at its end, which will make the while condition ($K \leq 4$) true forever!

So, be careful and remember to update the while loop condition before continuing it. For example, you can avoid the infinite loop by using this:

K = K + 1

ContinueLoop-

Return Statements

You can optionally use the **Return** command in a subroutine to exit it at once. It actually jumps to the **EndSub** line, so any code line that follows the **Return** line will never be executed. Practically, this command is used inside an **If** statement to control when to exit the subroutine. For example:

```
Sub Print(msg)
    If msg = "" Then
        Return
    EndIf
    TextWindow.WriteLine(msg)
EndSub
```

The **Return** command is used also in functions, but it is not optimal, as it is the only way to return the function value. It also exits the function at once, but here it pushes its return value at the stack, so that the caller can read (or ignore) it.

For example:

```
TextWindow.WriteLine({
    Divide(4, 2),
    Divide(1, 0)
})

Function Divide(a, b)
    If b = 0 Then
        Return "Can't devide by 0"
    EndIf

    Return a / b
EndFunction
```

When you run the above code, it will print:

2

Can't divide by 0

Subroutine Call Statements

To call a subroutine, you just write its name followed by the argument list. If the subroutine doesn't have parameters, you must use an empty parentheses () after the name. For example:

```
SayHello()  
Print("Hello sVB!")  
  
Sub SayHello()  
    TextWindow.WriteLine("Hello")  
EndSub  
  
Sub Print(msg)  
    TextWindow.WriteLine(msg)  
EndSub
```

Note that you can also call a function as if it is a subroutine, which means ignoring its return value. For example, this code calls the ShowMsg function twice: Once as method call to check its return value, and the other as a subroutine call ignoring its return value:

```
If ShowMsg("Are you happy with sVB?")  
    = DialogResults.Yes Then  
    ShowMsg("Thanks")  
EndIf  
  
Function ShowMsg(msg)  
    Return Dialogs.MessageBox("Attention!", msg)  
EndFunction
```

You can also do the same with any sVB method, to ignore its return value. For example:

```
Dialogs.MessageBox("Test", "Hello sVB!")
```



If Statements

The If statement allows you to check condition and take a decision based on whether it is true or false.

The If statement has many forms:

* **The If .. EndIf statement:**

In its simplest form, the If statement has only one code block (branch):

```
If condition Then  
    ' The code you want to execute  
    ' when the condition is true  
EndIf
```

Where the condition can be:

1. any Boolean variable. For example:

```
A = True  
If A Then  
    TextWindow.WriteLine("A is True")  
EndIf
```

2. Any function or method call or any property expression that returns a Boolean value:

```
X = "12"  
If Text.IsNumeric(X) Then  
    TextWindow.WriteLine("x contains a number")  
EndIf
```

3. any logical comparison that includes the =, <>, >, >=, < or <= binary operators. For example:

```
X = 7  
Y = 2  
If X = Y Then  
    TextWindow.WriteLine("x = y")  
EndIf
```

```
If X < 10 Then
    TextWindow.WriteLine("x < 10")
EndIf
```

Note that using a Boolean variable or a method return value as a condition is in fact a short cut for comparing them to

True:

```
A = True
If A = True Then
    TextWindow.WriteLine("A is True")
EndIf
```

This means you can check if a string is not numeric like this:

```
X = "12$"
If X.IsNumeric = False Then
    TextWindow.WriteLine("x is not a number")
EndIf
```

4. A composite condition that contains multiple conditions combined together with one or more **And** or **Or** binary operators, like:

```
X = 7
Y = 2
If X = Y Or X > 5 Then
    TextWindow.WriteLine("OK")
EndIf

If X > 0 And Y > 0 And X + Y < 10 Then
    TextWindow.WriteLine("Also OK")
EndIf
```

Note that:

1. When the left hand condition of an **And** operator is false, sVB realizes that the operation result is false, so it doesn't evaluate the right hand condition because its value will not matter.
2. When the left hand condition of an **Or** operator is true, sVB realizes that the operation result is true, so it doesn't evaluate the right hand condition because its value will not matter.
3. When you use **And** and **Or** together, you should enclose some operations with parentheses to tell sVB in what order it should perform these operations. For example, the following expression:

$(X > 0 \text{ Or } Y > 3) \text{ And } Z < 7$

is different than this one:

$X > 0 \text{ Or } (Y > 3 \text{ And } Z < 7)$.

* The If ..Else statement:

The If statement can optionally contain an **Else** block, where you provide the code you want to execute when the condition is false. For example:

```
X = -1
If X > 0 Then
    TextWindow.WriteLine("x is a positive number")
Else
    TextWindow.WriteLine("x is a negative number")
EndIf
```

* The If ..ElseIf statement:

The If statement can optionally contain one or more **ElseIf** blocks, where you can check two or more related conditions. The first one of those conditions found to be true, sVB executes its code block and jumps to the EndIf line without checking the rest of conditions. For example, this code checks the student marks to state its grade:

```
Marks = 55
If Marks < 30 Then
    Grade = "Too week"
ElseIf Marks < 50 Then
    Grade = "Week"
ElseIf Marks < 65 Then
    Grade = "Accepted"
ElseIf Marks < 75 Then
    Grade = "Good"
ElseIf Marks < 85 Then
    Grade = "Very good"
ElseIf Marks >= 85 Then
    Grade = "Excellent"
EndIf
TextWindow.WriteLine(Grade)
```

* The If ..ElseIf ..Else statement:

The If ElseIf statement can optionally end with an **Else** block, where you provide the code you want to execute when all the above conditions are false. For example, there is no need in the last example to check the condition:

```
ElseIf Marks >= 85 Then
```

because this when all the above condition fails, this implies that the student marks are not < 85, which means they are of course ≥ 85 , so we can just use Else to execute this last default case:

```
Marks = 90
```

```
If Marks < 30 Then
```

```
    Grade = "Too weak"
```

```
ElseIf Marks < 50 Then
```

```
    Grade = "Weak"
```

```
ElseIf Marks < 65 Then
```

```
    Grade = "Accepted"
```

```
ElseIf Marks < 75 Then
```

```
    Grade = "Good"
```

```
ElseIf Marks < 85 Then
```

```
    Grade = "Very good"
```

```
Else
```

```
    Grade = "Excellent"
```

```
EndIf
```

```
TextWindow.WriteLine(Grade)
```

* Nested If statements:

The code blocks for the If, ElseIf, and Else can contain one or more statements of any type, including If statements and other code blocks like for loops. and this can go deeper to any number of levels as you need. To see this in action, open the "Simple Calculator 3" project from the samples folder, and look at the validation subroutine:

```
Sub Validate(numTextBox)
    c = Event.LastTextInput
    If Text.IsNumeric(c) = False Then
        If c = "-" Then
            If Text.StartsWith(
numTextBox.Text, "-") Then
                Sound.PlayBellRing()
                Event.Handled = True
            Else
                numTextBox.Select(1, 0)
            EndIf
        ElseIf c = "." Then
            If Text.Contains(
numTextBox.Text, ".") Then
                Sound.PlayBellRing()
                Event.Handled = True
            EndIf
        Else
            Sound.PlayBellRing()
            Event.Handled = True
        EndIf
    EndIf
EndSub
```

For Statements

The for loop is used to repeat a task for a certain number of times. It uses a counter (also called an iterator) that begins with a given start value, and increases by a given step value at every iteration of the loop, and keeps doing that until it reaches a final value, which will be the last iteration of the loop. This is the general form of the For loop:

```
For Counter = start To Final Step increment  
    ' a task to repeat  
Next
```

* **Next and EndFor:**

For backward compatibility with SB, the **EndFor** keyword can still be used instead of **Next**, but I encourage you to use **Next**, as it gives the meaning of repeating and circulating over the loop, while **EndFor** can be confusing because it gives the meaning of finishing and exiting which may imply that "the loop finishes here", not just "the end of the for block is here"!

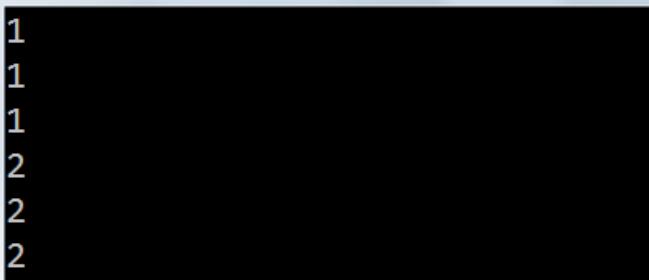
* **The loop counter (iterator):**

The counter can be any variable with any name you choose. After the loops ends, the counter value will be grater than the final value. This code for example will print 11:

```
For n = 1 To 10 Step 2  
Next  
TextWindow.WriteLine(n) ' 11
```

The counter variable is inferred as a Double type, which allows you to use it as an object and access the Math type methods directly from it:

```
For I = 1 To 2 Step 0.2
    TextWindow.WriteLine(I.Round)
Next
```



```
1
1
1
2
2
2
```

If the for loop is written at the file level (global the domain), the counter will be a global variable, and it can be changed from any subroutine you call from the loop body, which will alter how the loop and may cause it to loop forever! For example, the following loop will step by 2 despite we define it to step by 1, because we tampered with the counter in the Test subroutine:

```
For Counter = 1 To 10 Step 1
    Test()
Next

Sub Test()
    TextWindow.WriteLine(Counter)
    Counter = Counter + 1
EndSub
```



```
1
3
5
7
9
```

If the loop is defined in a subroutine or a function, its counter will always be a local variable, and it will hide any global variable with the same name. For example, the following code (which you can try in the "loop iterator domain.sb" file in the samples folder, contains two counters with the name i, but one of them is global while the other is local, and each of them is isolated form the

other:

```
X = {  
    "1. First item",  
    "2. second item",  
    "3. Third item"  
}  
' I is a global variable  
For I = 1 To 3  
    TextWindow.WriteLine(X[I])  
    WriteSubItems()  
Next  
  
Sub WriteSubItems()  
    y = {  
        "a) First subItem",  
        "b) second subItem",  
        "c) third subItem"  
    }  
' "i" is a local variable and it hides  
' the global variable with the same name "I"  
    For i = 1 To 3  
        TextWindow.WriteLine("    " + y[i])  
    Next  
EndSub
```

* The loop step:

The step clause of the for loop is optional, which means you can omit it, and in this case the step will be 1 by default:

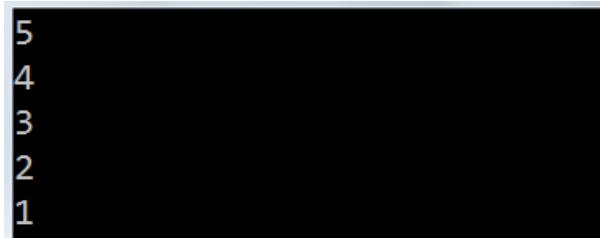
```
For i = 1 To 5  
    TextWindow.WriteLine(i)  
Next
```



1
2
3
4
5

The step can be a negative number, to allow you to loop backward from a large number to a smaller one:

```
For i = 5 To 1 Step -1  
    TextWindow.WriteLine(i)  
Next
```



5
4
3
2
1

Don't use a 0 step, otherwise the loop will continue infinitely and will never end!

If the final value is less than the start value while the step is positive, the loop will be skipped and the task will never be executed. The same will happen if the final value is greater than the start value while the step is negative.

If the final value and the start value are equal, the loop will iterate for only once.

* Nested For loops:

You can write two or more nested loops (loops inside loops), where each of them must have a unique counter name, because all of them belong to the same variable scope (global or local):

```
For x = 1 To 2  
    For y = 1 To 2  
        For z = 1 To 2  
            TextWindow.WriteLine(Text.Format(  
                "([1],[2],[3])",  
                {x, y, z}  
            ))  
        Next  
    Next  
Next
```

```
(1,1,1)
(1,1,2)
(1,2,1)
(1,2,2)
(2,1,1)
(2,1,2)
(2,2,1)
(2,2,2)
```

* Exit Loop:

You can use the **ExitLoop** statement inside the For loop body to immediately exit the loop and jump to the code line that follows the **Next** line. This code for example will print 6 not 11, because the loop exits when n equals 6 which makes the if condition true:

```
For n = 1 To 10
  If n > 5 Then
    ExitLoop
  EndIf
Next
TextWindow.WriteLine(n)
```

If you want to exit the parent loop, use **ExitLoop-** , and if you want to exit the parent of the parent loop, use **ExitLoop--** , and so on. For an example, see the "ExitLoop.sb" file in the samples folder.

If you want to exit the most outer loop to exit all nested loops at once use **ExitLoop*** . For example, this code will print nothing on the screen, because the **ExitLoop*** command exits all loops before executing the WriteLine method, but if you removed (or commented out) the **ExitLoop*** line, you will see 1 and 2 printed on two lines :

```

For x = 1 To 2
    For y = 1 To 2
        For z = 1 To 2
            ExitLoop *
        Next
    Next
    TextWindow.WriteLine(x)
Next

```

Continue Loop:

You can use the **ContinueLoop** statement inside the For loop body to immediately jump to the **Next** line to increase the counter and continue the next iteration. For example, the following code will print all numbers from 1 to 10 except 7 because it fulfills the condition that executes the ContinueLoop command:

```

For n = 1 To 10
    If n.Remainder(2) = 1 And
        n Mod 5 = 2 Then
            ContinueLoop
    EndIf
    TextWindow.WriteLine(n)
Next

```

If you want to continue the parent loop, use **ContinueLoop-**, and if you want to exit the parent of the parent loop, use **ContinueLoop--**, and so on. For an example, see the "ExitLoop.sb" file in the samples folder.

If you want to exit the most outer loop to exit all nested loops at once use **ContinueLoop***.

The Loop Stability:

In SB, the for loop final and step values can be changed in loop body. For example, this is an infinite loop in SB:

```
n = 5
For i = 1 To n
    TextWindow.WriteLine(i)
    n = n + 1
EndFor
```

But in sVB, final and step values are set once and changing them in the loop body will have no effect on the loop. This means that the above for loop will print the numbers from 1 to 5 and end normally, and changing "n" in loop body is irrelevant. This makes sVB consistent with VB6 and VB.NET, and it is also a good optimization, to avoid recalculating final and step expressions in every loop iteration.

You can though change the counter in the loop body, which is very dangerous because it can corrupt the loop or make it an infinite loop, but sometimes it can be useful, like when you need to process the next item of the array with the current item, then jump over the next item by manually increasing the counter by 1. So, be careful when you manipulate the counter in the loop body.

sVB compilers applies Text.ToNumber method on the start, final and increment expression of the for loops to ensure that they are valid numbers. This method will return 0 if any of these expression is a string, but if you use a single character, this method will return its Unicode value. For example, the following loop will show the numbers from 97 to 122, which are the Unicode values of the letters from a to z:

```
For i = "a" To "z"
    TextWindow.WriteLine(i)
Next
```

This is useful when you want to loop in a range of letters. For example, this loop will show the letters from F to M:

```
For i = "F" To "M"  
    TextWindow.WriteLine(i.KeyChar())  
Next
```

ForEach Statements

The ForEach statement is a special form of the For statement. It is used to iterate through items of arrays in the form:

ForEach Item In Arr

Next

The Item identifier can have any name you want, and it will contain the first item of the array at the beginning, then will contain the next item at the next iteration, and so on. for example:

```
numbers = {10, 20, 30, 40}  
ForEach number In numbers  
    TextWindow.WriteLine(number)
```

Next

The ForEach statement is easier to deal with arrays with string keys, when you just want to get the items values regardless of their keys. For example:

```
arr[1] = "One"  
arr["msg"] = "Hello"  
arr!Name = "Ahmad"  
ForEach item In arr  
    TextWindow.WriteLine(item)
```

Next

The above code will print:

One

Hello

Ahmad

Note that sVB can't infer the iterator type (like number and item in the above samples), because the array can contain items from different types. But you can tell sVB what to expect by applying the [naming conventions rules](#) on the iterator identifier. For example, using the strName as an iterator tells sVB it is a string, so we can use the Text methods via it:

```
names = {"Adam", "John", "Yasser"}  
ForEach strName In names  
    TextWindow.WriteLine(strName.UpperCase)  
Next
```

All other rules of the For statement applies on the ForEach statement, such as using ExitLoop and ContinueLoop statements in the ForEach body.

While Statements

The while loop is used to repeat a task as long as a certain condition is available (true). This is the general form of the while loop:

```
While Condition  
    ' a task to repeat  
Wend
```

* **Wend and EndWhile:**

For backward compatibility, the **EndWhile** keyword can still be used instead of **Wend**, but I encourage you to use **Wend** to be consistent with Visual Basic. It is true that Visual Basic .NET uses **EndWhile**, but I don't like it because it seems confusing as it gives the meaning of finishing and exiting, which may imply that "the loop finishes here", not just "the end of the while block is here"!

Wend is the best choice here, as it gives the meaning of repeating and circulating over the loop.

* **Infinite While Loop:**

It is easy to get an infinite using **While**, as all you need is to provide a condition that is always true like $1 = 1$, or you can just use **True**:

```
While True  
    TextWindow.WriteLine("Infinite loop!")  
Wend
```

The challenge here is not to have an infinite while loop, since you have to make sure that the while condition can get false at some point. For example, this is an infinite loop, because it contains no instructions to update the value of n, hence the condition will

remain true forever:

```
n = 1
While n <= 5
    TextWindow.WriteLine(n)
Wend
```

It seems that the above code was meant to print numbers from 1 to 5 on the screen, but it will keep printing 1 endlessly!

So, to fix that, we must increment n in the loop body, so that the condition becomes false when n reaches 6 and we got 5 different numbers on the screen:

```
n = 1
While n <= 5
    TextWindow.WriteLine(n)
    n = n + 1
Wend
```

* Nested Loops:

You can have two or three or more nested while loops (loops inside loops). For example:

```
x = 1
While x < 3
    y = 1
    While y < 3
        z = 1
        While z < 3
            TextWindow.WriteLine(Text.Format(
                "([1],[2],[3])",
                {x, y, z}
            ))
            z = z + 1
        Wend
        y = y + 1
    Wend
    x = x + 1
Wend
```

```
(1,1,1)  
(1,1,2)  
(1,2,1)  
(1,2,2)  
(2,1,1)  
(2,1,2)  
(2,2,1)  
(2,2,2)
```

It is also OK to write nested for and while loops as you need.

* Exit Loop:

You can use the **ExitLoop** statement inside the while loop body to immediately exit the loop and jump to the code line that follows the **Wend** line.

The following program will keep asking the user to enter a number to print the numbers from 1 to it, and the loop will exit only when the users enters 0 or an empty line:

```
While True  
    TextWindow.WriteLine("Enter a number")  
    TextWindow.Write(" (or 0 exit):")  
    N = TextWindow.ReadNumber()  
    If N = 0 Then  
        ExitLoop  
    EndIf  
  
    S = 0  
    For I = 1 To N  
        S = S + 1  
        TextWindow.WriteLine(S + " ")  
    Next  
    TextWindow.WriteLine("")  
Wend
```

If you want to exit the parent loop, use **ExitLoop-** , and if you want to exit the parent of the parent loop, use **ExitLoop--** , and so on.

If you want to exit the most outer loop (to exit all nested loops at once) use **ExitLoop*** .

* Continue Loop:

You can use the **ContinueLoop** statement inside the while loop body to immediately jump to the **Wend** line and continue the next iteration. But that can be dangerous if it leads to jump over the line that updates the while loop condition (like increasing a variable value or so). so make sure to update the condition if necessary before continuing the loop.

For example, the following code will print all numbers from 1 to 10 except 7 because it fulfills the condition that executes the ContinueLoop command:

```
N = 1
While N < 11
    If N Mod 2 = 1 And
        N.Remainder(5) = 2 Then
            N = N + 1
            ContinueLoop
        EndIf
    TextWindow.WriteLine(N)
    N = N + 1
Wend
TextWindow.WriteLine("The end.")
```

Note that if you removed the `N = N + 1` line from the if statement body, the loop will never end, and you will see the numbers from 1 to 6 on the screen but the "The end." statement will not appear because the loop still working forever!

If you want to continue the parent loop, use **ContinueLoop-**, and if you want to exit the parent of the parent loop, use **ContinueLoop--**, and so on.

If you want to exit the most outer loop to exit all nested loops at once use **ContinueLoop***.

Sub Statements

A Subroutine is a named block of code, so that you can execute it as many times as you need by just calling its name.

For example:

```
PrintTime()  
Program.Delay(1000)  
PrintTime()  
  
Sub PrintTime  
    TextWindow.WriteLine(Date.Now)  
EndSub
```

Editor support:

The upper right dropdown list in the code editor shows the names of all subroutines and function in the current file, which makes it easy for you to navigate to any of them by just clicking its name.

The list also contains the "(Add new Sub)" command, which you can click to get a new empty subroutine added to the file for you, which looks like this:

```
'-----  
Sub NewSub_1()  
EndSub
```

The NewSub_1 will be selected and you can type the subroutine name directly.

You can define a parameters list for the sub, like this:

```
Print("Distance", 120)  
  
Sub Print(name, value)  
    TextWindow.WriteLine(  
        "Name=" + name + ", Value=" + value)  
EndSub
```

and you can use the [naming conventions rules](#) to allow sVB to infer the parameters types. For example:

```
Sub Print(strName, dblValue)
    TextWindow.WriteLine(
        "Name=" + strName.LowerCase +
        ", Value=" + dblValue.Abs)
EndSub
```

Note that you can use the [Return statement](#) inside the sub body to exit the sub immediately.

* Event Handlers:

The subroutine can work as an event handler, which means that this subroutine will be called every time that event is fired.

You can add an event handler to the code editor by any of these ways:

1. Double-clicking the form or any control in the form designer:
This will switch to the code editor and add the handler for default event of the control, like Button1_OnClick.
2. Selecting the control name from the upper left dropdown list:
This will show the control events in the right dropdown list, and you can click anyone of them to add its handler to the file.
3. Using the naming convention to manually writing the subroutine:
sVB uses the same VB6 naming convention for event handlers, which is: ControlName_EventName.
4. Setting the event handler by code:
For example, this code will set the Click subroutine to be the handler of the form.OnClick event:

```

Me.OnClick = Click
Sub Click()
    Me.Text = Text.Format(
        "([1],[2])", {Me.MouseX, Me.MouseY})
EndSub

```

* Why do we need subroutines?

Using subroutines has these benefits:

1. Readability:

Using subroutines allows you to divide the program into named units, which makes it easier to understand.

2. Brevity:

Calling the subroutine name many times is less verbose than repeating the same code for many times in different places.

3. Controllability:

You can control how the subroutine work via its input parameters, so, the same block of code can do a slightly different job according to the arguments you sent to.

4. Maintainability:

You can easily fix and modify the code in only one place (the subroutine) rather than chasing your tail all over your code.

* Subroutine documentation:

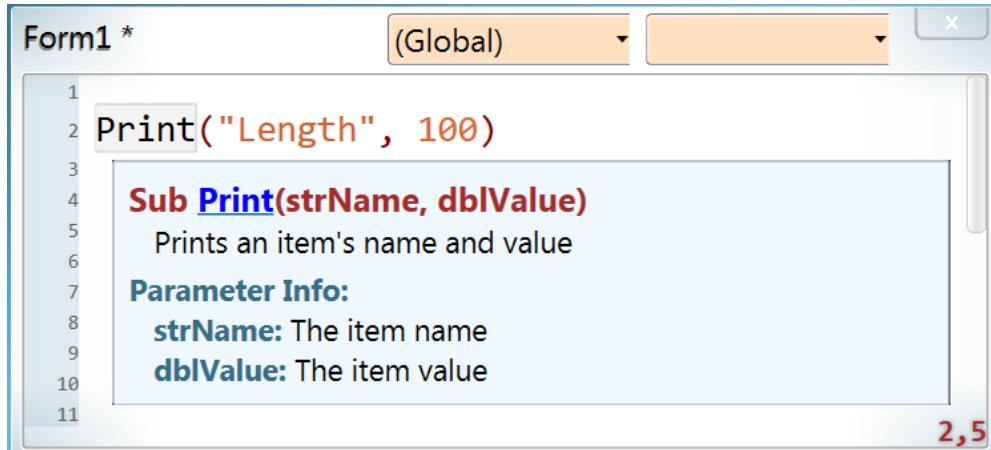
You can use inline comment documentation to provide information about the subroutine and its parameters.

You can provide the subroutine definition in the lines above it and at the end of its header. You can also wrap parameters over multi-lines and provide definitions for some or all of them. For example:

```

' Prints an item's name and value
Sub Print(
    strName, ' The item name
    dblValue ' The item value
)
    TextWindow.WriteLine(
        "Name=" + strName.LowerCase +
        ", Value=" + dblValue.Abs)
EndSub

```



Function Statements

A functions is a [subroutine](#) that can return a result. It can has zero or more parameters, and uses the [Return statement](#) to return a value. For example:

```
X = Sum(1, 2)  
Function Sum(x, y)  
    Return x + y  
EndFunction
```

Editor support:

The upper right dropdown list in the code editor shows the names of all subroutines and function in the current file, which makes it easy for you to navigate to any of them by just clicking its name.

The list also contains the "(Add new Function)" command, which you can click to get a new empty function added to the file for you, which looks like this:

```
'-----  
Function NewFunc_1()  
    Return 0  
EndFunction
```

The NewFunc_1 willl be selected and you can type the function name directly.

* **Test Functions:**

The test function is a normal function but its name starts with the Test_ prefix, and it returns a string that tells you about the result of testing a part of your form.

For more info on how to write and run test functions, see the explanation and samples provided with the [Form.RunTests method](#).

* Function documentation:

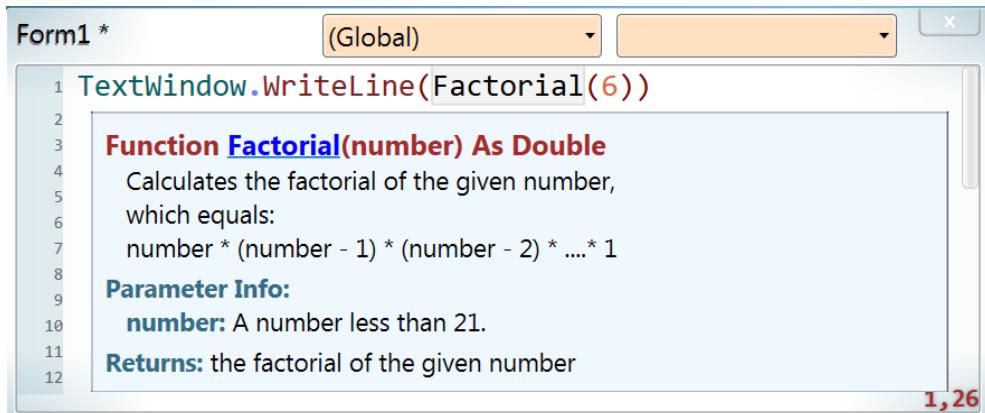
You can use inline comment documentation to provide definitions for the function and its parameters, exactly like you do with subroutines.

You can also provide a definition for the function return value after its right parenthesis. For example:

```
' Calculates the factorial of the given number,
' which equals:
' number * (number - 1) * (number - 2) * .... * 1
Function Factorial(
    number ' A number less than 21.
) ' the factorial of the given number

If number < 2 Then
    Return 1
EndIf

Return number * Factorial(number - 1)
EndFunction
```



Note that the Factorial function is a recursive function, because it call itself recursively until a certain condition stops that. Be careful when you crate a recursive function not to call itself indefinitely because this will crash your application with the "Out of stack memory" error.

* Inferring the function return type:

The good thing you can notice from the above picture is that sVB succeeded to infer the type of the function return value as a Double. sVB looks for all the return statements used in the function, and tries to infer their types, and if they all belong to the same type, it uses it as the function return type, otherwise it uses the wider type that can include all other types (like the Control type that includes TextBox, Button and all other controls). If there is no such type, sVB fails to infer the function return type, which by the way is expected in a recursive function because its return type depends on its own return type, but luckily, the multiplication operator is valid only for members, and this is why sVB succeeded to infer the type.

So, generally, if sVB failed to infer the return type, you should give it a hand, by using the [naming conventions rules](#) to define one or more variables so that sVB know their types, then use them in the return statements of the function!

sVB Projects

The sVB IDE starts with a default unsaved form named "Form1", which doesn't belong to any project. The files of this form is temporarily saved in a folder with a name based on the current date and time. This folder will be saved in:

C:\Users\[UserName]\AppData\Local\SmallVisualBasic

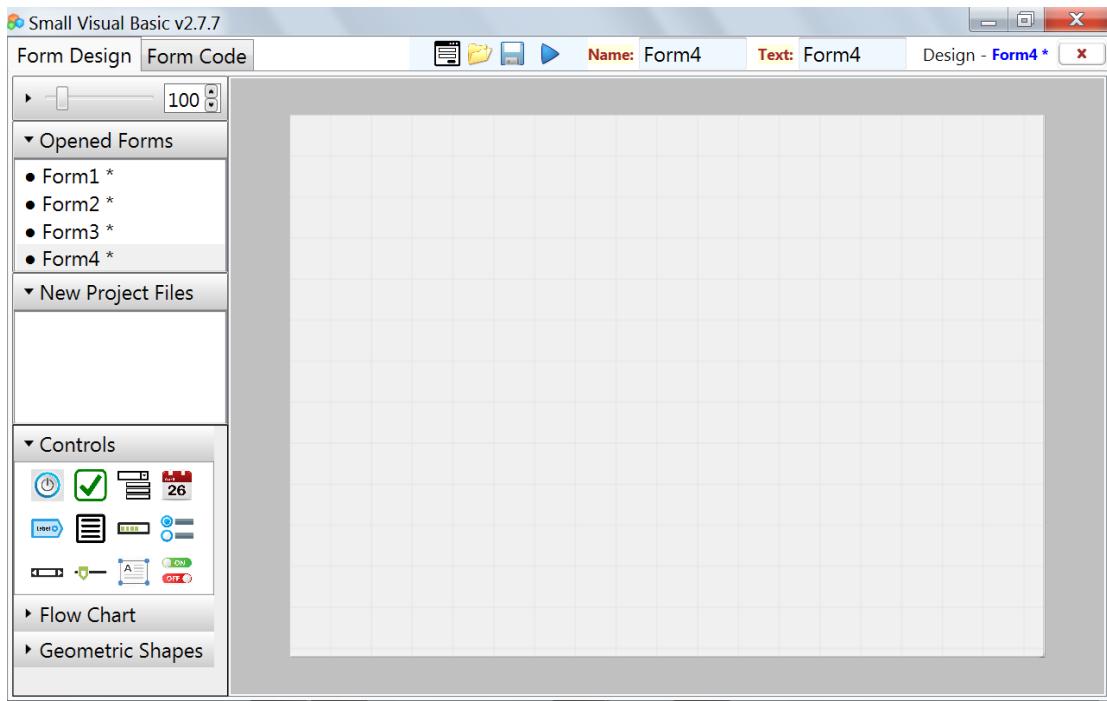
where you should replace the [UserName] in the above path with the name of your windows account. Note that sVB will remove this temporary folder after 10 days.

This allows you to quickly design the form, program its code and press F5 to run it, without having to save it yourself, which makes easy for you to try any small samples you want.

You can click the Add new form button from the toolbar, or press Ctrl+N to create as many new forms as you need, but they will not belong to any project. They will just appear in the list of the "opened Forms" side tab, which allow you to switch between open forms, and if any of them belong to a project, this project files will appear in the list of the "Project Files" tap. This means that you can open many projects in the same time in the sVB IDE, and use the opened Forms list to switch between them, and the Project Files list to switch between the current active project files.

But why don't you see any files in the Project Files list?

By definition, the sVB project is a folder that contains one or more forms, except the unsaved temporary folders.



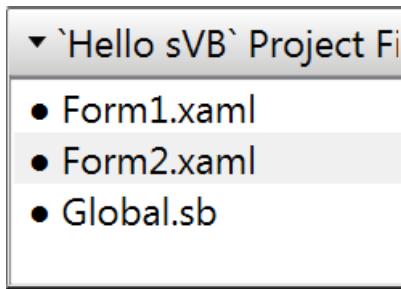
* Create a project:

The first step to create a project is to save one form to your file system, and I advice you not to use the desktop or "my documents" to save your forms, not to make them all belong to the same project.

In fact, you should avoid your windows drive (which is mostly the C:\), and create a folder names "sVB projects" on another drive, and use it to save you work so you don't lose it when you had to reformat your windows drive and reinstall it.

Now, when you want to create a project, add a new form and click the save button on the toolbar or press Ctrl+S to save it. sVB will show you the save file dialog, so you can navigate to your "sVB projects" folder, and create a new folder in it with the name of your project (say "Hello sVB") and save the form in it. Now you will see the name of the project on the project files tab, and the list will show the Form1.xaml and global.sb files. You can add a

second form and its name will appear in the list just after you save it to the project folder.



* The Form Files:

Each form you create with the designer is saved as three files:

1. **Form.xaml:** which contains the form and its controls design. This file is distributed with your application exe file, which allows you to make changes to the form design by external tools like VS.NET to use a more advanced design features not available in sVB, but you shouldn't do this unless you have a good knowledge about XAML language, and you must not change the form nor the controls names, because they are used in the code.
2. **Form.sb.gen:** Which contains a generated sVB code that loads the form from the form.xaml file, define the event handlers for the form and controls, and shows the form. This code is executed before the code you write in the form global area.
3. **Form.sb:** which contains the code you write to do the form job. You can switch to this file by clicking the Form Code tab on top of the form designer.

* Open an existing project:

To open a saved project, just open any form from its folder.

There are many ways to do this:

- In the form designer, click the "Open existing form" button from the toolbar or press Ctrl+O, and choose a xaml file from the project folder.
- In the code editor click the "Open" button from the toolbar or press Ctrl+O, and choose the global.sb file or an sb file of any form from the project folder.
- Open the project folder in the file explorer, and double-click any sb file. If sVB is not the default program to open sb files, Windows will ask you to choose a program, so choose it. You can also choose to open xaml files by sVB, but note that xaml files are used with VS.NET and not all xaml files contain form designs, so they can cause errors when you open with sVB, so you should be careful if you went with this choice.

* Generating the executable file:

You can run the active project by clicking the run button (from the toolbar of the designer or the code editor) or by pressing F5. This will make sVB compile the project to be able to run it.

Compilation means to convert the code you wrote to the **Microsoft Intermediate Language** (or in short MSIL or IL), which is saved in a file that has the name of the project with the exe extension (for example: "Hello sVB.exe").

This file is saved in the bin folder inside the project folder. sVB will also copy to the bin folder all files necessary to make the exe file work correctly, like the xaml files of all the project forms, any txt, image, sound or style files existing in the project folder, the SmallVisualBasicLibrary.dll file and any other external library

file that your project use.

You can select the exe file and press Enter or double-click it, to make the .NET Framework compile the IL language to the machine language suitable to your operating system, and run the application. So, this exe file is the executable file of your application, and the whole bin folder is considered your application, and you can copy it to any place on your file system, or distribute it to other PC's that have the .NET Framework 4.5, and it will work correctly.

* **The startup form:**

The main form of the project, is simply the active form that you run the project from.

This means that when you open any form of the project and run it, sVB will compile all the forms that exist in the same folder (project) into one exe, which will have the name of the folder (project), and that form will be shown in the runtime, and when you close it the program will be closed.

So if you want to start up with another form, you just need to switch to it in the designer and run the project!

This makes you easily test your project forms while you creating them, but remember after finishing the project to run it from the main form, so that you get your the final exe.

* **Show another form:**

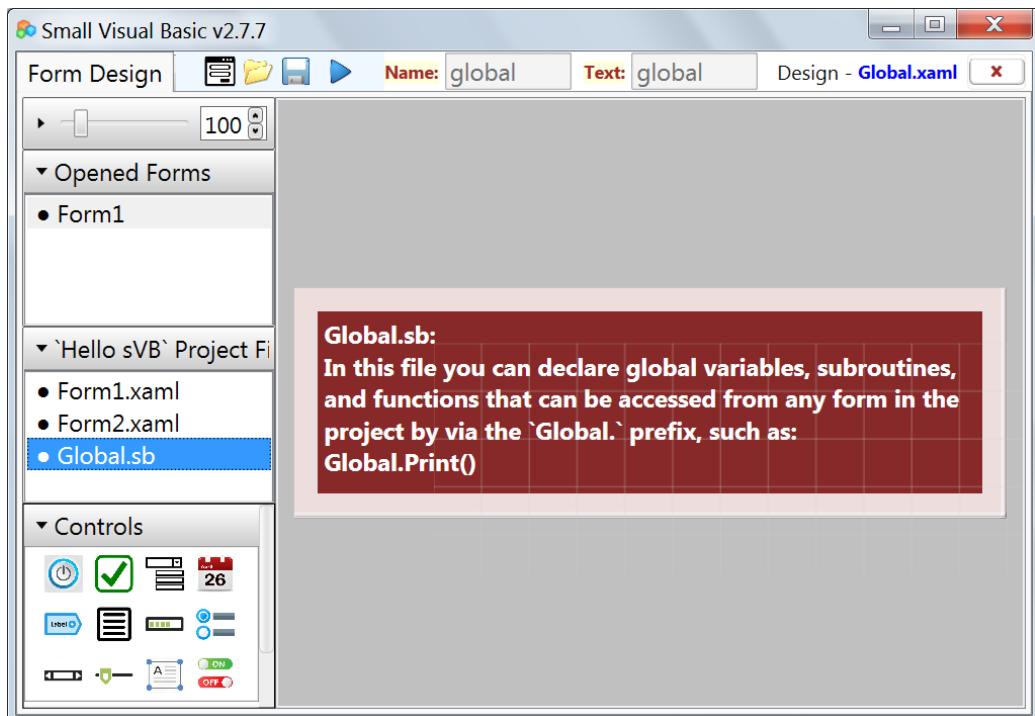
When you have a multi form project, it will start with the main form, and you will need to show the others from it. You can do that by using on of these methods:

1. [Forms.ShowForm](#) to show a normal form.
2. [Forms.ShowDialog](#) to show a modal dialog form.
3. [Form.ShowChildForm](#) to show an owned child form.

* The Global File:

Each project can contain a Global.sb file to work as a global module for the project. This is the only allowed sb file that doesn't belong to a form, and if you add any sb files to the project folder, the will be ignored.

You can create the global file as you create any sb file from the code editor and name it global.sb when you save it. But for simplicity, sVB provides you with another way too create the global file, just by double-clicking the Global.sb item in the project explorer.



Use the global file to declare variables, subroutines and functions that you want to use from any form in the project via the "Global" type, such as "Global.Print()".

For example, go to the "sVB Notepad" project in the samples folder, and open the global.sb file. This is the code of this file:

```

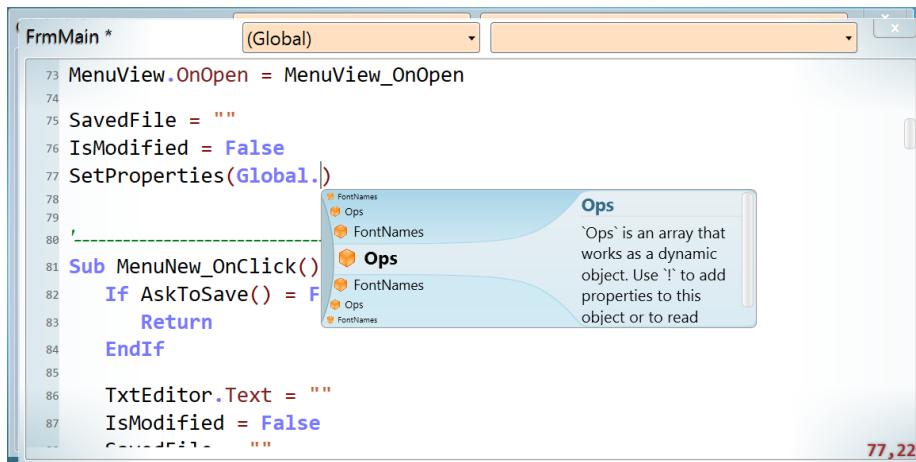
FontNames = Desktop.FontNames
_File = File.GetSettingsFilePath()
_Settings = File.ReadLines(_File)
If _Settings = "" Then
    Ops!FontName = "Segoe UI"
    Ops!FontSize = 20
    Ops!Bold = False
    Ops!Italic = False
    Ops!Underlined = False
    Ops!ForeColor = Colors.Black
    Ops!BackColor = Colors.White
    Ops!FrameColor = Colors.SystemControl
    Ops!RightToLeft = False
    Ops!MultiLine = True
    Ops!WordWrap = True
Else
    Ops!FontName = _Settings[1]
    Ops!FontSize = _Settings[2]
    Ops!Bold = _Settings[3]
    Ops!Italic = _Settings[4]
    Ops!Underlined = _Settings[5]
    Ops!ForeColor = _Settings[6]
    Ops!BackColor = _Settings[7]
    Ops!FrameColor = _Settings[8]
    Ops!RightToLeft = _Settings[9]
    Ops!MultiLine = _Settings[10]
    Ops!WordWrap = _Settings[11]
EndIf

```

In the above code, we defined two global variables:

1. FontNames: which is an array used to store the system font names, to be shown in the options forms.
2. Ops: which is a dynamic object that contains the default options, which are saved in the project settings file. we read this file to populate the Ops object, but if file isn't created yet or is empty, we manually set the values of the ops properties.

You can use these two global variables in any other form, via the Global keyword, with the editor auto-completion support. For example, we use the Global.Ops in the frmMain.sb file to set the textbox properties.



The screenshot shows the sVB IDE interface with a code editor window titled 'FrmMain *'. The code is written in sBasic. A tooltip is displayed over the word 'Ops' in the line 'SetProperties(Global.)'. The tooltip contains the following text: "'Ops' is an array that works as a dynamic object. Use `!` to add properties to this object or to read". The code editor also shows a status bar with the numbers '77, 22'.

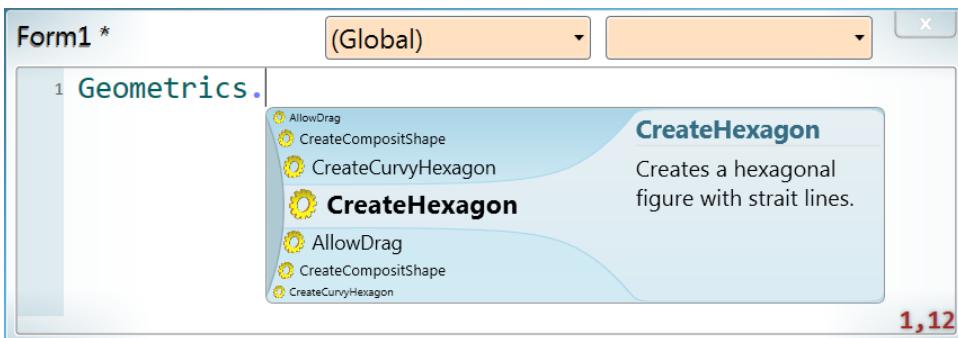
```
73 MenuView.OnOpen = MenuView_OnOpen
74
75 SavedFile = ""
76 IsModified = False
77 SetProperties(Global.)
78 '
79
80 '
81 Sub MenuNew_OnClick()
82   If AskToSave() = F
83     Return
84   EndIf
85
86   TxtEditor.Text = ""
87   IsModified = False
88 EndSub
```

Now, you may ask me: Why don't _File and _Settings appear in the auto-completion list?

The answer is that sVB considers any variable that starts with _ as a local variable, so it can't be used outside the global.sb file. This applies also on subroutines and functions that start with _. So, using this convention, you can write local variables and functions that are privately used in the global file only, so only a few global items can be used in other forms.

For example, the global.sb file of the Graphics project (found in the samples folder) contains the two private _OnMouseDown and _OnMouseMove subroutines that are used as event handlers and you don't need to manually call them from any other form. And there are 4 other global subroutines (AllowDrag, CreateCompositeShape, CreateCurvyHexagon, and CreateHexagon) that you can call from any form in the project. In fact, this project is used to create the Geometrics library, so you

can call these subroutines from any sVB project via the [Geometrics type](#):



* Using sVB to create a code library!

The [global file](#) makes it possible to use an sVB application as a code library! To do that, follow these instructions:

1. Create a sVB project with a global file. Choose a suitable name for the folder that you save the project files to (such as MyLib), because this name will be the name of your library. Don't use spaces nor symbols.
2. Add variables, subroutines, and functions to the global file. These are the members that you can access from the library. But if you want to declare some private members to use inside the global module only, you can start their names with "_" such as "_count" and "_GetNames()".
3. Add comments for each variable, subroutine, parameter, and return value. These comments will be saved as the documentation of your library, and will be shown in popup help in sVB when you use this library.
4. You can also add a comment at the beginning of the global file, to be used as the documentation of the Lib Type.
5. The project can contain as many forms as you need, but you must choose unique names when saving them to the project

folder. The names Form1.xaml, Form2.xaml and Form3.xaml can cause troubles later, so, if you have to, name them MyLib_Form1.xaml, MyLib_Form2.xaml and MyLib_Form3.xaml. Don't rename the files manually from windows explorer, and use the sVB project explorer to rename them, to do necessary changes to their .sb.gen files.

6. Run the project to create the exe file in the Bin folder in the project folder.
7. Now you can copy the Bin folder and paste in the "sVB\Bin\Lib" folder, then rename it to the name of your library such as "MyLib" in this example. The name of this folder is not important, but "Bin" is not a suitable name, and of course you can't add two libraries with the name "Bin" because their folders will be merged, which can cause troubles later!
8. Restart sVB, and in the code editor write the name of your library such as MyLib, then press dot. The auto completion list will show the members you declared in the global file of this library, and can normally use them. If you have added comments to those members, you will get help info about them while typing.

It is so simple, and you have two library projects ("Dialogs" and "Geometrics") in the samples folder and you can take a look at their global files. These two libraries are already added to the sVB/bin/Lib folder, so, you can use them in any sVB project via the [Dialogs](#) and [Geometrics](#) types.

It is amazing! You can now write a reusable code, and create your own libraries for sVB, using sVB itself.

Have fun.

Note:

When you write test functions in the global.sb file, they will be compiled as friend/internal methods, which means that you can't call them from any other project when you use their project as a code library. So, be careful when you write a code library not to start the name of any non-test function with the Test_ prefix.

* Using VB.NET and C# to Create a library for sVB:

sVB can use external libraries written with C# or VB.NET.

This is not necessary in most cases, as we can use sVB itself to create libraries as we saw in the previous topic, unless you need to add a new functionality to sVB that can't be programmed using sVB itself like creating Excel sheets or communicating with a data base. Such functionalities are provided by the .NET framework, so, it's easy to use VB.NET or C# to bring them to sVB via an external library, but of course this can only be done by a professional .NET programmer.

To create such library, follow these steps:

1. Create a .Net class library that targets the .NET Framework 4.5.
2. Add a reference to the SmallVisualBasicLibrary.dll file, located in the sVB\Bin folder.
3. Add some classes to the project, and mark any class you want sVB to use, with the SmallVisualBasicTypeAttribute.

For example:

```
<SmallVisualBasicType>
Public Class DemoLib
End Class
```

4. sVB detects only static (shared) properties and methods. For example:

```
Public Shared Property Value As Primitive
```

5. All fields, properties, parameters and return values should be of type Microsoft.SmallVisualBasic.Library.Primitive. For example:

```
Public Shared Function Increase(  
    delta As Primitive) As Primitive  
    Value += delta  
    Return Value  
End Function
```

6. All events should be of type Microsoft.SmallVisualBasic.Library.SmallVisualBasicCallback.
7. If you want sVB to infer the exact type of the primitive value, use the:

Microsoft.SmallVisualBasic.WinForms.ReturnValueTypeAttribute

to tell it what to expect. For example:

```
<ReturnValueType(VariableType.Boolean)>  
Public Shared ReadOnly Property IsPositive _  
    As Primitive  
    Get  
        Return Value >= 0  
    End Get  
End Property
```

8. You can use xml comments to provide sVB with info about each property and method. For example:

```
''' <summary>  
''' Decreases the value by the given delta  
''' </summary>  
''' <param name="delta">a number</param>  
''' <returns>the new value</returns>  
<ReturnValueType(VariableType.Double)>  
Public Shared Function Decrease(  
    delta As Primitive) As Primitive  
    Value -= delta  
    Return Value  
End Function
```

9. Compile your assembly and copy the dll file and the xml documentation file to the "sVB\Bin\lib" folder.

For an example, see the "DemoLib" project at the samples folder. It is a VB.NET project that creates a DemoLib.dll in its bin\release folder, which is already copied to the sVB\bin\lib folder, so you can use it in any sVB project. You can also find a sample that uses this DemoLib in the "DemoLibSample" project in the samples folder.

But in fact, all the sVB library itself is written with VB.NET and follows the rules mentioned above (except referencing itself of course). You can see a wealth of samples on how to create a sVB type in the [Small Visual Basic Library source code on GitHub](#).

The sVB Debugger

* Debugging sVB projects:

When you press F5 to run the project, sVB compiles the code and builds the Exe file of the project, which is saved to the bin folder in the project folder, with all necessary files copied to this folder like xaml, image and txt files plus the SmallVisualVasicLibrary.dll file that contains the sVB code library. After building the exe, sVB launches it, so your program runs and executes your code.

So actually, sVB IDE doesn't communicate with the running program, except that it can close it when you click the "End program" button that appears on the running screen.

So, what happens when an error occurs in your program?

Well, sVB can tell you about the errors that it can discover in your code, and the sVB library can alert you about the errors that happen when the program is running, but sVB IDE can't tell you where exactly the error happened in your code.

Finding and fixing program errors is called debugging.

Programming languages provide a tool to help you to debug your program, which is called: a debugger. Luckily, Small Visual Basic finally got a debugger starting from version 3.0, which is similar to VS .NET debugger that is used with VB .NET and C#. We will learn about this debugger and how to use it in the following sections, but first let's look a little deeper into errors and why they occur, so we can be able to fix them.

* **Types of errors:**

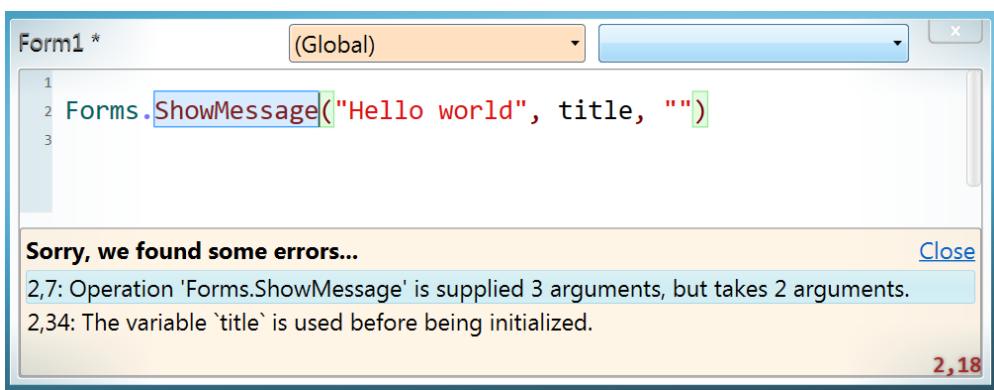
There are three types of errors:

1. Compiler errors:

These errors can happen while the sVB compiles your code, due to syntax errors, like missing an operator or mistyping a keyword. Programming languages expect you to write the code in the correct form, and if you don't, they give you syntax errors.

Semantic errors can also happen, like when you try to read a variable that you didn't set its value yet, or call a subroutine that doesn't exist, or pass the wrong number of arguments to a function... etc.

The sVB compiler detects syntax and semantic errors before building the exe, and if it finds any of them, the build process is cancelled, and a list of errors is shown at the bottom of the code editor.

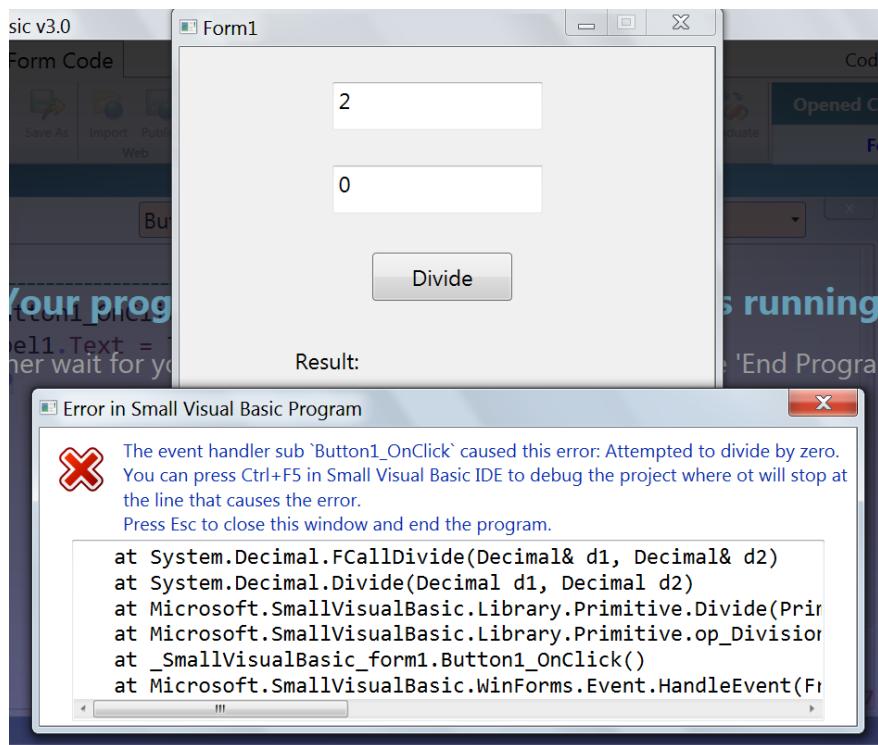


You can double-click any error in the list to highlight the part of code that caused it. You must fix all these errors then try to run the program again.

2. Runtime errors:

These errors can happen while your program is running, due to executing a wrong operation. The compiler can't detect such type of errors before running the program, because they depend on the values of variables and the user actions.

For example, if you asked the user to enter two numbers in two textboxes to divide them and show the result, the user can enter 0 in the second textbox, and if you don't write code to handle such case, it will cause a divide by zero error at run time:



When a runtime error happens, the program will display an error message that describes the error, and informs you that the program will be terminated after you close the message. It also advises you to run your program in debug mode, so you can discover the line of code that caused this

error, which is an important step that allows you to identify the source of the error to be able to fix it.

You can debug the program by clicking the Debug button or pressing Ctrl+F5.

3. Logical errors:

These errors can't be detected by the compiler nor the runtime, because they don't disrupt the execution of the program, but they can cause wrong results or make the program misbehave. These errors can be caused because you made a mistake in the program algorithm, so you and the users of the program are responsible for detecting such errors.

To fix such errors, you must review your code and find what's wrong in its logic, or add a missing code that handle some cases. Sometimes it is not easy to discover the cause of the logic error, so you may need to run the program in debug mode and trace the code and variable values to see what is going on.

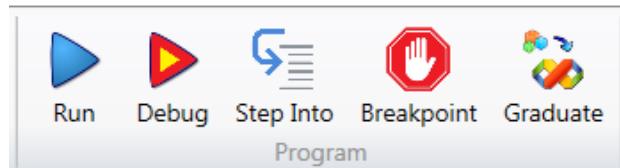
So, you can conclude that the debugger is important to help you fix runtime and logical errors.

*** Debugger Commands:**

You can run the program in debug mode by clicking the Debug button, which you can see on the form designer toolbar next to the Run button:



This button is also displayed on the code editor toolbar:



Or you can just press **Ctrl+F5** to start the debugger.

The code editor toolbar also displays a Step Into button, which runs the debugger and make it pause it at the first line of the program.

Note that the code editor toolbar will change when you run in debug mode, where some new buttons will be displayed to give you full control over the debugger. The following picture show you the program part of toolbar in the debug mode:



Let's see what those buttons can do:

- **Run (F5):**

Continues running the debugger if it is currently paused.

- **Pause(Ctrl+F11):**

Forces the program to pause at the code line that is about to be executed. This line will be marked with a golden background, as you can see in the following picture.

Note that when the program is in break mode, you can insert the caret at any variable or select any valid expression, to show its current value. In the picture, the popup tooltip tells you that the value of the SpeedY variable is 10 at this moment.

The screenshot shows the Microsoft Visual Studio debugger's code editor. The window title is "Form1". A tooltip is displayed over the variable "SpeedY", which is highlighted in red. The tooltip contains the following information:

- Global Variable: SpeedY As Double**
- S You can use this variable as an object of the form SpeedY.MemberName
- Value: 10**

The code in the editor is as follows:

```
1 SpeedX = 5
2 SpeedY = 10
3 G
4 S
5 R
6 T
7 Timer.Tick = MoveBall
8 RacketTimer = Me.AddTimer("RacketTimer", 20)
9 RacketTimer.Pause()
10 RacketTimer.OnTick = MoveRacket
11
12 Sub MoveBall()
13     Ball.Top = Ball.Top + SpeedY
14     If GameOver And Ball.Top > Me.Height Then
```

A status bar at the bottom right shows the number "2,5".

- **End program(Shift+F5):**

Ends the running program. The code editor toolbar will return to its normal state, showing the hidden buttons and hiding the extra debugging buttons.

- **Step into(F11):**

If the debugger is paused (in break mode), this command will execute the current line and pauses at the next line. If the current line calls any other part of the program like a subroutine or a function even in another file, the debugger will pause at the first line of that part.

In the following picture, the program is paused at the line that calls the Test subroutine. If you click Step Into, it will pause at the line pointed to by the red arrow, which is the first line of the Test subroutine.

You can click Step Into repeatedly to execute the code line by line, which allows you to see how the code is executed and check the values of the variables. Not only this can help you discover the source of the error that bugs your program, but also it can allow you to analyze and understand the code.

of some programs that are not written by you. For example, try to step into the Cars game in the sVB samples folder, and see how it works. This is the first step that may help you fix the issues I left in the game (like other cars running over your car, or your car getting out of the race track).

You also can do the same with other samples, like the Tetris game.

Note that executing the EndSub or EndFunction line will take you back to the line that called the current subroutine or function.

```
Form1 *
(Global)
1 Test()
2 X = 1
3 Sub Test()
4 y = 1
5 EndSub
```

2,1

- **Step Over(F10):**

If the debugger is paused (in break mode), this command will execute the current line and pauses at the next line. If the current line calls any subroutines or functions, the debugger will execute them but will not stop at any of them. even if they contain breakpoints.

In the following picture, the program is paused at the line that calls the Test subroutine. If you click Step Over, it will pause at the line pointed to by the red arrow, which is exactly the next line. The debugger will execute the Test subroutine, but will not pause inside it. This is why we say that the debugger stepped over the subroutine call.

```

1
2 Test()
3 X = 1
4
5 Sub Test()
6   y = 1
7 EndSub

```

- **Block Over(Ctrl+F10):**

If the debugger is paused (in break mode), this command will execute the current line, stepping over its inner blocks if any. For example: If the current line is the If statement header, clicking this button will execute the if statement block, but the debugger will not pause at any statement in the If statement body or any body of any related ElseIf and Else bodies, and will pause only at the line next to the EndIf line.

The same happens if the current line is ElseIf or Else.

The following picture shows you the Ball Game program in break mode. It is paused at an If statement, and if you click Block Over, it will pause at the line pointed to by the red arrow.

```

21 If Ball.Bottom >= Racket.Top Then
22   If Ball.Right >= Racket.Left
23     And Ball.Left <= Racket.Right Then
24       Ball.Bottom = Racket.Top
25       SpeedY = -SpeedY
26       Score = Score + 1
27       Me.Text = Score
28       Sound.Play("ball_hit.mp3")
29     Else
30       GameOver = True
31     EndIf
32
33   ElseIf Ball.Top < 0 Then
34     Ball.Top = 0
35     SpeedY = -SpeedY
36     Sound.Play("boing.wav")
37   EndIf
38
39 Ball.Left = Ball.Left + SpeedX

```

The debugger will do the same if the current line is the head of any other code block, like For, ForEach and While statements, where the debugger will pause at the line next to the block end, like Next and Wend.

If the current line calls a subroutine or a function, this command will step over it.

- **Step Out(F12):**

If the debugger is paused (in break mode), this command will execute the current subroutine, and pause at the line that called it.

In the following picture, the program is paused at a line inside the Test subroutine. If you click Step Out, the debugger will execute the for loop and exit the Test subroutine, then pause at the line pointed to by the red arrow, which is the line that called the Test subroutine.

```
Form1 *
(Global) Test
1 → Test()
2 X = 1
3
4 Sub Test()
5 y = 0
6 For i = 1 To 10
7     y = y + i
8 Next
9 EndSub
5,1
```

- **Block Out(Ctrl+F12):**

If the debugger is paused (in break mode), this command will execute the body of the code block that contains the current line, and pause at the line next to the end of the block.

In the following picture, the program is paused at a line inside the for loop. If you click Block Out, the debugger will continue executing the for loop, then pause at the line pointed to by the red arrow, which is first line following to the Next token.

```
Form1 *
(Global) Test
1 Test()
2 X = 1
3
4 Sub Test()
5 y = 0
6 For i = 1 To 10
7 y = y + i
8 Next
9 →Forms.ShowMessage(y, "Sum")
10 EndSub
7,1
```

Note that if the debugger is paused at the line of For or Next, clicking block out will step out the parent block that contains the For loop, which is the Test subroutine in our sample, so it will make you step out to the caller line.

- **Breakpoint(F9):**

You can use this command at any time (even while debugging) to add a breakpoint at the current line or remove an existing one.

Note that you can also toggle the breakpoint of any line by double-clicking its margin.

Breakpoints can save your time, because they allow you to pause the debugger at the lines you choose, without having to stepping into the code from the first line.

The screenshot shows a Windows application window titled "Form1 *". The status bar indicates "(Global)". The code editor displays the following VBScript code:

```
1 Test()
2 X = 1
3
4 Sub Test()
5     y = 0
6     For i = 1 To 10
7         y = y + i
8     Next
9     Forms.ShowMessage(y, "Sum")
10 EndSub
```

A red circle marks a breakpoint at line 6. The line "For i = 1 To 10" is highlighted in red. The status bar at the bottom right shows "1,1".

All you need is to add a breakpoint at a line, then run the program in debug mode. The debugger will execute the code of the program, and if it encounters the line with the breakpoint, it will pause to allow you to examine the variable values and step through the code.

The screenshot shows a Windows application window titled "Form1 *". The status bar indicates "(Global)" and "Test". The code editor displays the same VBScript code as the previous screenshot. A red circle marks a breakpoint at line 6. The line "For i = 1 To 10" is highlighted in yellow. A tooltip box appears over the variable "y", containing the following information:

Local Variable: y As Double
Note that you can use this variable as an object using the syntax:
y.MemberName
Value: 0

The status bar at the bottom right shows "7,8".

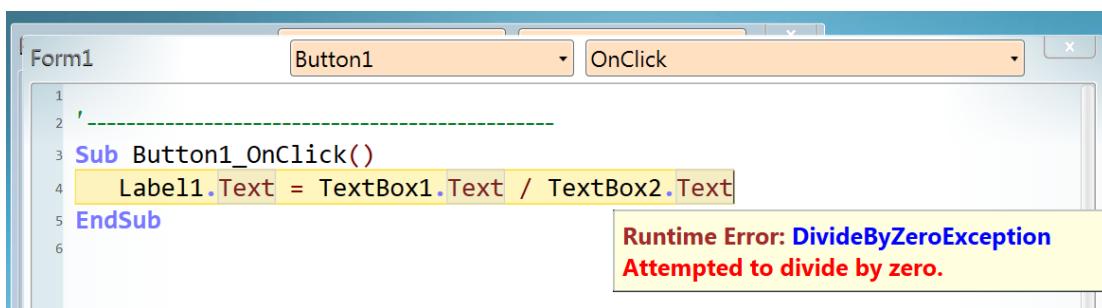
You can remove the breakpoint in the break mode or add new ones, and click Run to continue debugging and let the debugger stop at the next breakpoint.

* Break at errors:

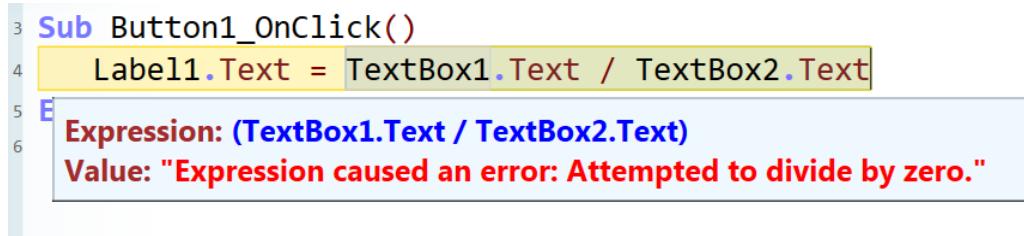
When you run your program in debug mode and encounter a runtime error, it will pause at the line that causes the error, which will allow you to fix it.

Note that you can't change the code while the program is running, because all the code files of the project will be read only. So, when the debugger pauses at error, you can only read the error message and check the values of the variables, but you can't change anything, nor continue executing the program, because it will be terminated immediately.

In the divide by zero example that we mentioned before, if we run the program in debug mode, the following picture shows you how the debugger behave when the error happens:



You can select the expression `TextBox1.Text / TextBox2.Text` to see its value, which will prove to you it is the source of the error:



You can also insert the caret in the Text property of TextBox1 or TextBox2 to show their values. But the only action you can take now is to click the End Program, so you can be able to fix this

error. One possible fix is to write an If statement to check the value if TextBox2.Text and show a message to the user to inform him that 0 is not an accepted value. You can see how this is done in the calculator samples in the sVB samples folder.

* Debugging Test Functions:

The debugger can debug test functions that you call using the [Form.RunTests](#), [Form.RunGlobalTests](#), [UnitTest.RunAllTests](#), [UnitTest.RunFormTests](#) or [UnitTest.RunGlobalTests](#) methods.

If you step into any of those methods, the debugger will step into the first test method, and after it ends it will step into the second one, the third one, and so on.

You can also add breakpoints in any test function, to force the debugger to stop at them when you run the tests.

* How does the debugger work?

It is important to understand how the debugger works, to avoid some issues that can happen when using it.

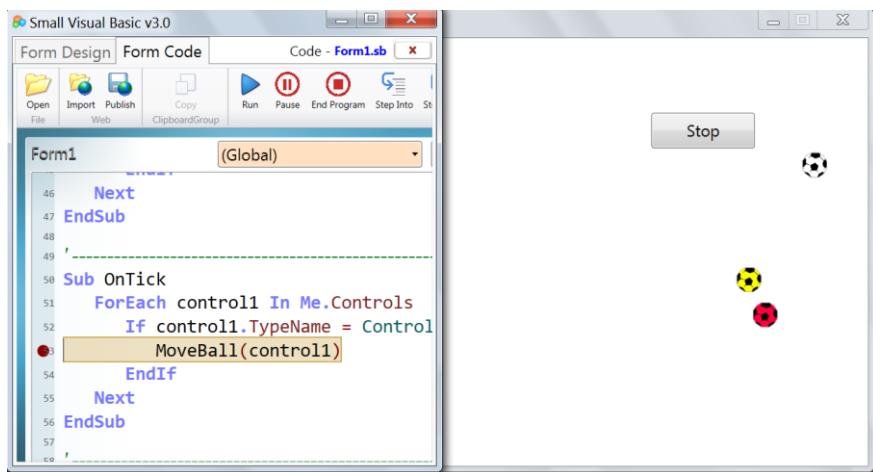
When you start the debugger, sVB compiles the code and builds the exe, but it doesn't launch it. The compile process is necessary to make sure that the code doesn't have any syntax or semantic errors, and to build the syntax tree that is used by the debugger to execute the program.

On the other hand, building the exe is not necessary for the debugger, but it ensures that the bin folder always contains the last updated version of your executable application, even after you debug the project. This exe is a clean build of the project that has no debugging symbols, and doesn't relate to the debugger by anyway, so you can use and distribute as you want. But you may ask: Why doesn't sVB run in debug mode by

default?

Actually, the debugger can run a bit slower than the exe and this can be noticeable when you debug an application that executes many turtle movements and turns even when you turn off turtle animation. This is because:

1. The debugger interprets each instruction at runtime, so it cannot execute subroutines, if statements and loops, and evaluates variables and method calls. If the same statement is called more than once (line in for loops or subroutine calls), it will be reinterpreted every time, which can be slower than executing compiled low-level code.
2. The debugger uses a high level dictionary to store the values of the variables and subroutine parameters which is slower than using the low level stack while executing the exe.
3. The debugger runs in the same process of the sVB IDE, which demands to call every event handler in a new thread to keep the IDE responsive in break mode, and allow you to view see the changes on the form while you are stepping through its code. Try to put a breakpoint in the MoveBall subroutine in one of the Ball projects in the sVB samples folder, show both of the form and sVB IDE on the screen side by side, and see how the balls move step by step on the form:



Creating a new thread for each event handler can be expensive, especially when you user timers, because every tick of each timer will create a new thread.

Note that the debugger prevents creating a new thread for the same event handler until the previous one is finished. It also suspends all threads in break mode, so that you can capture the program state at the breaking moment, and step through code lines without being disrupted by ticking events. This makes you able to debug games (where timers are often used), without getting a game over while you are in break mode!

But remember that the form is still responsive in break mode, so what if you can click any button while the debugger is paused?

In that case, the debugger will show you a message that reminds you that you are in the debug mode, and will switch you to the line that is about to be executed in the code editor.

Using threads for events can also cause some issues when debugging events that are fired by code, like when you change the CheckBox.Checked property from code, which fires the the OnCheck Event. So, be careful when you use the debugger in such situations.

You can conclude that you should not use the debugger unless you need to understand how the code works (or explain it to students), or discover the source of a bug, otherwise it is faster to run the compiled program.

* Debugging best practice:

1. You also should use the debugger wisely, because stepping through the code too fast and too much (like when you keep your finger down on the F11 key inside a long for loop) can cause some threads to enter a dead lock which stops some parts of the program from being executed, and if this happens, you should end the debugger and start over.
2. When you handle events that can be triggered frequently in a short time (like when the user presses the up key to speed up the car in a game), try to avoid unnecessary executions of these handlers if possible, especially if the handler code is long or contains loops or calls another subroutines, because this can block out the debugger thread. For example, the Form1_OnKeyDown handler in the Car game calls some subroutines to adjust some variables that are used in the OnTick handler of the timer to speed the car up or down, or move it left or right. The timer continues doing that until the user lifts his finger up, so that the Form1_OnKeyUp handler resets those variables to stop speeding up or down and moving left or right. So, the user will keep his finger down on one of the arrow keys (say the Up key to speed the car up), which means that the Form1_OnKeyDown will be called 50 or 100 times per second, which is OK in runtime, but it will overkill the debugger because each call will create a new thread, and each thread will try to block new ones until it finished, and this will hang up the game!
So, you should avoid such situation. Actually the solution is easy: Just exit the handler if a moving or braking command is being executed, because the OnKeyDown handler will actually do nothing while the timer is the one who doing the job. This

is the code that is used to fix the debugging issue, and also it makes the game performs better in runtime:

```
Sub Form1_OnKeyDown()
    If StartMove Or Braking Then
        Return
    EndIf
    ' The rest of the code
EndSub
```

3. Executing the Form.OnClosing event in a separate thread can cause an issue in debug mode when you set the Event.Cancel to True to cancel closing the form because the main thread continues executing its code and may not read the value you set, so the form may be closing anyway! To solve this, the debugger will force the main thread to wait for a 100 millisecond, which means that you make sure you write the:

```
Event.Handled = True
```

at the beginning of the handler code and avoid executing any loops or calling any long running functions before it to make sure it will be executed before the 100ms delay elapsed.

4. You can [create new threads](#) to run some tasks of your program in parallel. But in debug mode, threads will not be created, to allow you to trace the code. In this case your program may be slower and some of its logic may not function correctly.

The sVB Library

The rest of this book, will cover in full details the sVB class library and will provide samples on every method and property, and explain some of the projects that you can find in the samples folder inside the sVB folder.

Note that the LitDev external library is now installed with sVB. See [LitDev documentation](#) for more details. You can also download [litDev samples](#)

These are the types that we will learn:

Array	Dictionary	Program
Button	Event	ProgressBar
Chars	File	RadioButton
CheckBox	Form	ScrollBar
Clock	Forms	Shapes
Color	GeometricPath	Slider
Colors	Geometrics	Sound
ComboBox	GraphicsWindow	Stack
Control	ImageList	Text
Controls	Keyboard	TextBox
ControlTypes	Keys	TextWindow
Date	Label	Thread
DatePicker	ListBox	Timer
DemoLib	MainMenu	ToggleButton
Desktop	Math	Turtle
DialogResults	MenuItem	UnitTest
Dialogs	Mouse	WinTimer

The Array Type

Contains method to deal with arrays, which are listed below:

Array.AddItem(array, value) As Array

Adds an item to the array.

Parameters:

- **array:**

The input array. This argument can be omitted if you call this method as an extension method of an array variable.

- **value:**

The item you want to add after the last item in the array. If you send an array, it will be added as a single item as a whole, so, the new index of the array will refer to a sub array.
See examples below.

Returns:

A new array with the new item added. The input array will not be changed.

Examples:

```
a = {}  
a = Array.AddItem(a, "First item")  
a = Array.AddItem(a, "Second item")  
a = Array.AddItem(a, "Third item")
```

After executing the above code, `a` will contain the three added items at indexes 1, 2, and 3 respectively.

As you notice, we reassigned the array returned by the AddItem method again to the variable a, otherwise it will not

be changed.

Note that sVB can infer the type of a to be an array, because a is initialized with an empty array. This allows you to use the Array methods as extension methods of the variable a, hence the above code can be rewritten as:

```
a = {}  
a = a.AddItem("First item")  
a = a.AddItem("Second item")  
a = a.AddItem("Third item")
```

Which is a little shorter, but still not the best.

It is better to use the indexer syntax instead:

```
a[1] = "First item"  
a[2] = "Second item"  
a[3] = "Third item"
```

Yet we can do better using the array initializer:

```
a = {  
    "First item",  
    "Second item",  
    "Third item"  
}
```

But this doesn't mean you will not use AddItem sometimes. It is useful when you want to append an item to an array without knowing its index, so, you can avoid doing this:

```
a[a.Count + 1] = "New Item"
```

Where we added the new item at the array index next to the its last one. Wouldn't you prefer to use this instead?:

```
a = a.AddItem("New item")
```

Also, this method is desired when you want to clone an array to another, plus adding a new item to the new array, such as:

```
b = a.AddItem("New value")
```

After executing the above command, `a` remains as it was

without any change, while `b` will have the same items of `a` plus the "New value" item that is newly appended to the items.. That is the short alternative of this code:

```
b = a  ' Copy items form `a` to `b`
b[b.Count + 1] = "New value"
```

So, the AddItem method can be useful after all.

Note that you can add an array as an item:

```
a = {}
a = a.AddItem(
    {"First item", "Second item", "Third item"}
)
```

After executing the above code, `a` will contain only one item at index 1, which is the given array. In fact the above code is equivalent to:

```
a = {
    {"first item", "Second item", "Third item"}
}
```

💡 **Array.AddItem(array, items) As Array**

Adds many items to the array.

Parameters:

- **array:**

The input array. This argument can be omitted if you call this method as an extension method of an array variable.

- **items:**

An array containing the items to add each of them as a single item at the end of the array.

Returns:

A new array with the given items added. The input array will not be changed.

Examples:

```
a = {1, 2, 3}  
a = Array.AddItems(a, {4, 5})  
b = a.AddItems({100, 200, {300, 400}, 500})
```

After executing the above code, `a` will contain:

{1, 2, 3, 4, 5}

and `b` will contain:

{1, 2, 3, 4, 5, 100, 200, {300, 400}, 500}

⌚ **Array.AddKeyValue(array, key, value) As Array**

Adds an item to the array, with the given key and value

Parameters:

- **array:**

The input array. This argument can be omitted if you call this method as an extension method of an array variable.

- **key:**

The key of the item. If there is already an item with this key, its value will be modified.

- **value:**

The value of the item. This value can be an array, which will be added as a single item.

Returns:

A new array with the item added. the input array will not be changed.

Examples:

This code will add two items to the array, they first has the key "Name" with the value "Adam", and the second has the key "Grades" with an array values containing 4 grades:

```
a = {}  
a = Array.AddKeyValue(a, "Name", "Adam")  
a = a.AddKeyValue("Grades", {75, 60, 82, 99})
```

It is easier and more efficient to rewrite the above code as:

```
a["Name"] = "Adam"  
a["Grades"] = {75, 60, 82, 99}
```

You can also use the dictionary lookup operator `!` to express the keys as dynamic properties, with code editor auto-completion support:

```
a!Name = "Adam"  
a!Grades = {75, 60, 82, 99}
```

⌚Array.Append(array, value)

Adds an item to the array. It is similar to the AddItem method, but here the given array will be changed directly, so this method is faster when you want to build a large array, but be careful because it will affect the reference array that the current array is copied from! See the [array performance](#) topic for more info.

Note that sVB calls this method to populate the array when you use the array initializer {}, so the initializer is not just shorter but also faster than adding individual items using the array indexer[].

Also, if you want to modify an item in a large array, don't use the indexer because it will copy the whole array. There is another fast reference method that works like the AddNext item to do this, which is the [SetItemAt](#) method.

Parameters:

- **array:**

The input array. This argument can be omitted if you call this method as an extension method of an array variable.

- **value:**

The item you want to add after the last item in the array. If you send an array, it will be added as a single item as a whole, so, the new index of the array will refer to a sub array.

Examples:

You can see this method in action in the "Array performance" project in the samples folder. It is used in the Button2_OnClick event handler to add 10,000 items to the array in less than 20% of a second.

Array.ContainsIndex(array, index) As Boolean

Gets whether or not the array contains the specified index.

Note that sVB array is actually a dictionary where each item is a key-value pair. The key is a unique string that you can use to access an individual item in the collection, to set or get its value. So, for sVB, array indexes are actually keys. This means that can use numbers or strings as indexes, and numeric indexed don't have to be continuous. For example, this code is perfectly valid in sVB:

```
a[1] = 10  
a[7] = 9  
a["test"] = "OK"  
a[-1] = "It is possible!"
```

The above code creates an array that contains the indexes "1", "7", "test", and "-1". The fact that all indexes are actually string keys makes it possible to use negative indexes like `a[-1]` in the above code, which is not allowed in for arrays in other languages!

In the above code, there are no indexes between 1 and 7, and this is why you may need the `ContainsIndex` method to check if an index (like 4) exists before using it, because if you try to read a value of a non-existed index, the array will return an empty string "" and will not raise an error. You may see this as a good thing in most situations (and it actually is), but sometimes you may need to distinguish between an item that contains the value "" and a non-existed item! That is when you should use the `ContainsIndex` method.

Parameters:

- **array:**

The array to check. This argument can be omitted if you call this method as an extension method of an array variable.

- **index:**

A number or a string that represents the index to check. Note that string indexes are case insensitive, so if the array has an `test` index, checking for "test" and "Test" both will return True.

Returns:

True if the index exists in the array or false otherwise.

Example:

```
a[1] = 10
a[3] = 20

If a.ContainsIndex(2) Then
    Forms.ShowMessage(a[2], "Sample")
Else
    Forms.ShowMessage("Item doesn't exist", "Sample")
EndIf

If Array.ContainsIndex(a, 4) = False Then
    a[4] = 30
EndIf
```

Array.ContainsValue(array, value) As Boolean

Gets whether or not the array contains the specified value.

Parameters:

- **array:**

The array to check. This argument can be omitted if you call this method as an extension method of an array variable.

- **value:**

The value to check.

Returns:

True if the value was present in the specified array.

False otherwise.

Examples:

```
x = Array.ContainsValue({1, 2, 3}, 2)
a = {1, 2, 3}
y = a.ContainsValue(4)
```

After executing the above code, `x` will equal True, and `y` will equal False.

Note that string values are case sensitive, while string keys are case insensitive!

```
a["test"] = "Hi"  
z = a.ContainsValue("Hi")  
w = a.ContainsValue("hi")
```

After executing the above code, `z` will equal True, and `w` will equal False.

Array.Count As Double

This is an extension property of array variables. It gets the number of items stored in the array.

You can also use the [Array.GetItemCount](#) method to do the same job.

Returns:

The number of items of the specified array.

Example:

```
a = {10, 20, 30}  
n = a.Count ' n will contain 3
```

Array.EmptyArray As Array

Creates an empty array. This is helpful when you need to initialize a variable as an array with no items, which you can also do with an empty pair of braces {}.

Initializing the variable is important to tell sVB about its type, so

you can use the methods of this type as extension methods of this variable.

Returns:

An empty array.

Example:

```
x = Array.EmptyArray  
count = x.Count
```

You can also use empty braces {} to create an empty array:

```
x = {}  
count = x.Count
```

After executing the above code, `count` will be 0.

⌚ **Array.Find(array, value, start, ignoreCase) As String**

Searches the array for the given value.

Parameters:

- **array:**

The input array. This argument can be omitted if you call this method as an extension method of an array variable.

- **value:**

The item to search for.

- **start:**

An integer representing the array position to start searching at. This value must be > 0.

- **ignoreCase:**

When set to True, the search will be case-insensitive.

Returns:

The index (key) of the item if found, otherwise empty string.

Example:

Suppose we have this array:

```
a = {10, 20, 30, "Adam"}  
a["Name"] = "Adam"  
a!Age = 15
```

Let's do some find operation on it. I will write the find result as a comment beside each line. If the result is an empty string (item not found), I will keep the comment empty:

```
x = a.Find(30, 1, True)           ' 3  
x = a.Find(30, 4, True)           ''  
x = a.Find("adam", 2, True)        ' 4  
x = a.Find("adam", 3, False)      ''  
x = a.Find("adam", 5, True)        ' Name  
x = a.Find("10", 1, True)          ''  
x = a.Find(15, 1, True)           ' Age
```

The importance of the second parameter (start) appears when we search for "Adam", as the array contains two of them, one with the key "4" and another with the key "Name". We can't find the last one unless we start searching at the 5th position of the array.

The importance of the third parameter (ignoreCase) appears when we search for `adam`, while the array contains only Adam. We can do a case insensitive search by sending True to this parameter, otherwise the search will be case sensitive, and we will not find `adam`.

The return result is the key of the item. The item 30 exists in the third position of the array and it's key is also 3, likewise "Adam" at the 4th position with the key 4, but the second

"Adam" exists at position 5 with the key "Name". So, The key of the item is not always his position in the array.

Array.GetAllIndices(array) As Array

Gets an array containing all the indexes for the given array.
You can use this method as an extension property of array variables, with the name [Indices](#).

Parameter:

- **array:**

The array whose indexes are requested.

Returns:

An array filled with all the indexes (keys) of the specified array.
The returned array indexes start from 1.

Examples:

Suppose we have this array:

```
a = {10, 20, 30, "Adam"}  
a["Name"] = "Adam"  
a!Age = 15
```

We can get its indexes like this:

```
ids = a.Indices
```

The ids array will be {1, 2, 3, 4, "Name", "Age"}.

And we can get id indexes of the ids array just to be sure:

```
ids2 = Array.GetAllIndices(a.Indices)
```

The ids2 array will be {1, 2, 3, 4, 5, 6}.

Array.GetItemAt(array, position)

Gets the value of the item that exists at the given position in the array.

Parameters:

- **array:**

The input array. This argument can be omitted if you call this method as an extension method of an array variable.

- **position:**

A number that represents the position of the item in the array, which is not always the same as its index (key). Note that position must be > 0 .

Returns:

The item value (if found), otherwise empty string "".

Examples:

Suppose we have this array:

```
a = {10, 20, 30, "Adam"}  
a["Name"] = "Adam"  
a!Age = 15
```

We can write a for loop to iterate through the array like this:

```
For i = 1 To a.Count  
    TextWindow.WriteLine(a.GetItemAt(i) + " ")  
Next
```

The TextWindow will show the next result:

10 20 30 Adam Adam 15

Array.GetItemCount(array) As Double

Gets the number of items stored in the array.

You can use this method as an extension property of array variables, with the name [Count](#).

Parameter:

- **array:**

The array for which the count is requested.

Returns:

The number of items of the specified array.

Example:

```
a = {10, 20, 30}
```

```
n = Array.GetItemCount(a) ' n will contain 3
```

Array.GetKeyAt(array, position) As String

Gets the key of the item that exists at the given array position.

Parameters:

- **array:**

The input array. This argument can be omitted if you call this method as an extension method of an array variable.

- **position:**

A number that represents the position of the item in the array, which not always be the same as its index (key). Note that position must be > 0.

Returns:

The key of the item if found, otherwise empty string.

Example:

Suppose we have this array:

```
a = {10, 20, 30, "Adam"}  
a["Name"] = "Adam"  
a!Age = 15
```

We can write a for loop to iterate through the array like this:

```
For i = 1 To a.Count  
    TextWindow.WriteLine(a.GetKeyAt(i) + " ")  
Next
```

The TextWindow will show the next result:

1 2 3 4 Name Age

⌚Array.IndexOf(array, value, start, ignoreCase) As Double

Searches the given array for the given value, and returns the index of this value in the array.

Parameters:

- **array:**

The input array. This argument can be omitted if you call this method as an extension method of an array variable.

- **value:**

the item to search for.

- **start:**

an integer representing the array index to start searching at.

• **atignoreCase:**

set it to true if you want to do an case-insensitive search.

Returns:

the position (index) of the item in the array if found, otherwise 0.

Examples:

```
X = {1, "ABC", 2}
TextWindow.WriteLine({
    Array.IndexOf(X, 10, 1, True), '0 (Not found)
    X.IndexOf(2, 1, True),           '3
    X.IndexOf(1, 2, True),           '0 (Not found)
    X.IndexOf("abc", 1, True),       '2
    X.IndexOf("abc", 1, False),      '0 (Not found)
    X.IndexOf("ABC", 1, True)        '2
})
```

Array.Indices As Array

This is an extension property of array variables that gets an array containing all the indexes for the current array.

Note that you can do the same job by calling the [Array.GetAllIndices](#) method.

Returns:

An array filled with all the indexes (keys) of the specified array.
The returned array indexes start from 1.

Examples:

Suppose we have this array:

```
a = {10, 20, 30, "Adam"}
a["Name"] = "Adam"
a!Age = 15
```

We can get its indexes like this:

ids = a.Indices

The ids array will be {1, 2, 3, 4, "Name", "Age"}.

And we can get id indexes of the ids array just to be sure:

ids2 = Array.GetAllIndices(a.Indices)

The ids2 array will be {1, 2, 3, 4, 5, 6}.

Array.IsArray(array) As Boolean

Gets whether or not a given variable is an array.

Parameter:

- **array:**

The variable to check.

Returns:

"True" if the specified variable is an array. "False" otherwise.

Examples:

This function receives two parameters `x` and `y` and adds them. If x and y are numeric or strings, the + sign will add them correctly, but if one of them is an array, the + sign will add the string representation of the array, which is not desired, so, in such case, we will do the addition in a different way. So, we need the IsArray method to know what we are dealing with:

```

Function Add(x, y)
  If Array.isArray(x) Then
    ' Add y as an item to the x array,
    ' even if y is an array
    Return Array.AddItem(x, y)

  ElseIf Array.isArray(y) Then
    ' Create a new array and add x to it,
    ' then add the items of the y array
    z[1] = x
    Return z.AddItems(y)

  Else ' Add x and y as numbers or strings.
    Return x + y
  EndIf
EndFunction

```

You can try this function in the `AddArray.sb` app in the samples folder.

Array.Join(array, separator) As String

Joins the given array items into one text.

Parameters:

- **array:**

The input array. This argument can be omitted if you call this method as an extension method of an array variable.

- **separator:**

A string to use as a separator between array items. You can use a single character or a many characters as a separator.

You can even use an empty string `""` as a separator to join all items without any separator!

Returns:

A string containing the array items, separated by the given separator

Example:

```
a = {1, 2, 3}  
x = a.Join(",")  
y = Array.Join({10, 20, 30}, ", ")
```

After executing the above code:

- x will contain the string "1,2,3"
 - y will contain the string "10, 20, 30"
-

Array.RemoveItem(array, index) As Array

Removes the array item at the specified index (key).

Parameters:

- **array:**

The input array. This argument can be omitted if you call this method as an extension method of an array variable.

- **index:**

The key of the item to remove.

Returns:

A new array with the item removed. The original array will not change.

Example:

```
a = Array.RemoveItem({"a", "b", "c", "d"}, 2)  
x = a.Join("")
```

After executing the above code, x will contain the string "acd".

Array.SetItemAt(array, position, value) As String

Sets the value of the array item that exists at the given numeric position.

Note that the input array will be changed directly, so this method is faster than the array indexer [] when dealing with a large array, but be careful because it will affect the reference array that the current array is copied from! See the [array performance](#) topic for more info.

Parameters:

- **array:**

The input array. This argument can be omitted if you call this method as an extension method of an array variable.

- **position:**

A number that represents the position of the item in the array, which is not always the same as its index (key).

Note that position must be > 0.

- **value:**

The value to set to the item.

Returns:

The item key if found at the given position, otherwise an empty string "".

Example:

```
a = {1, 2, 3, "Adam"}  
a.SetItemAt(4, "Ali")
```

After executing the above code, `a` will contain {1, 2, 3, "Ali"}.

Array.ToString(array) As String

Converts the current array to a string.

Parameters:

- **array:**

The input array. This argument can be omitted if you call this method as an extension method of an array variable.

Returns:

The string representation of the array.

If the array is simple, its string can be of the form {1, 2, 3}.

If the array contains keys, its string can be of the form:

{[1]=1, [2]=2, [Name]=Adam}.

Example:

```
a = {1, 2, 3, "Adam"}  
TextWindow.WriteLine(a.ToString())  
a["Name"] = "Adam"  
a!Age = 15  
TextWindow.WriteLine(a.ToString())
```

After executing the above code, The TextWindow will show:

```
{1, 2, 3, Adam}  
{[1]=1, [2]=2, [3]=3, [4]=Adam, [Name]=Adam, [Age]=15}
```

The Button Type

Represents the Button control, that the user can click to perform the task that you provide in the [OnClick](#) event handler

You can use the form designer to add a button to the form by dragging it from the toolbox.

It is also possible to use the [Form.AddButton](#) method to create a new button and add it to the form at runtime.

Note that the Button control inherits all the properties, methods and events of the [Control](#) type.

Besides, the Button control has some new members that are listed below:

Button.IsFlat As Boolean

Gets or sets whether or not to show the button with a flat style.

The default value is False, which means the button will have a 3D style by default.

Example:

Add two buttons on the form, and add this code to the form global area:

Button2.IsFlat = True

This will give Button2 a flat style, while Button1 keeps its 3D style, as shown in the picture:



Button.Text As String

Gets or sets the test that is displayed on the button.

Example:

To display OK on Button1:

```
Button1.Text = "OK"
```

Button.Underlined As Boolean

Gets or sets whether or not to draw a line under the text.

Example:

To display OK on Button1:

```
Button1.Text = "OK"  
Button1.Underlined = True
```

Button.WordWrap As Boolean

Gets or sets whether or not to the text will be continue on the next line if it exceeds the width of the control.

The default value is True, so the button title will be wrapped over lines by default. You can use the [Button.FitContentHeight](#) method to ensure that the button height fits the wrapped lines.

Example:

Add two buttons on the form, and add this code to the form global area:

```
Button1.Text = "Test wordwrap property"  
Button1.FitContentHeight()  
Button2.Text = "Test wordwrap property"  
Button2.WordWrap = False
```

The result will be as shown in the picture:



The Chars Type

Contains some useful non-printed characters, such as the escape and new line characters.

Note that all properties of this type are read-only (you can't change their values), and they all can be used of the form:

`C = Chars.Quote`

The `c` variable will contain the quote string " after executing the above code. So, I will note give an example for each of the Chars properties, as they all alike. These are the properties:

Chars.Back

Represents a backspace character.

Chars.Cr

Represents a carriage return character, which is the end of line.

Chars.CrLf

Represents a carriage return and a line feed character. Together, they indicate the presence of a new line.

Chars.FormFeed

Represents a form feed character for print functions.

Chars.Lf

Represents a line feed character, which is the start of a new line. In some cases, this character alone can be used to represent the new line.

Chars.Null

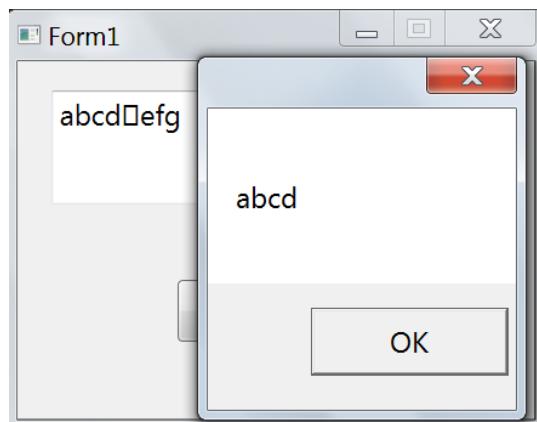
Represents a null character.

Note that null means nothing. It is different than empty string "" which is conceded `something` in programming. To see the difference, try this code in a form with a textbox:

```
x = "abcd" + Chars.Null + "efg"  
TextBox1.Text = x  
Me.ShowMessage(x, "")
```

After executing this code, the textbox will show the null character as a rectangle which is the way that textbox uses to display unrecognized characters. The letters efg will be displayed after that rectangle.

But in the windows message box, the null character will not be displayed, nor any character after it!. This is because windows deals with null-terminated strings, where the string continues until the first null character is found.



Now, try to change the Chars.Null in the above code with "", and run the program. Both the textbox and the message box will show the string "abcdefg".

Chars.Quote

Represents a double-quote character.

Chars.Tab

Represents a tab character.

Chars.VerticalTab

Represents a vertical tab character.

The Chars type has also these methods:

Chars.GetCharacter(characterCode) As String

Gets the corresponding character for the given Unicode character code.

Parameter:

- **characterCode:**

The character code in Unicode encoding system. This code can be 0 (the null character) but can't be a negative number. There are more than 1,1 million Unicode characters that covers all languages letters and all symbols of all kinds.

You can omit this argument when you call this method as an extension method (with the name ToChar) of a numeric variable.

Returns:

A Unicode character that corresponds to the code specified.

Note that the actual return value is a string, because a Unicode character in some cases is composed of two surrogates. This happens because a string character size is 2 bytes, but some Unicode characters size is 4 bytes, so the

string can represent each of them as a pair of surrogates. This means that the length of the string returned from this method will be 2, when you use a character code larger than 65535.

But the good news is that controls such as TextBox and Label controls can correctly show the Unicode character returned by this method, regardless its string representation.

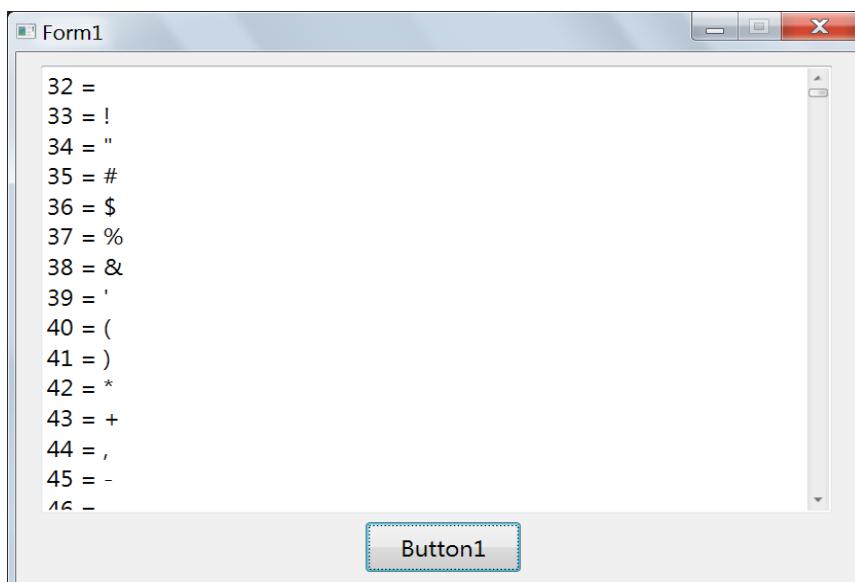
Examples:

Add a textbox and a button on a form, double-click the button and write this code in the Button1_OnClick event handler:

```
For code = 32 To 1000
    c = Chars.GetCharacter(code)
    TextBox1.Append(code)
    TextBox1.Append(" = ")
    TextBox1.AppendLine(c)
```

Next

Run the program and click the button. This will print 1000 Unicode character in the textbox, except the first 31 characters, because most of them are control characters that can't be printed. The code 32 represents the space character, and you can see what next codes represent!



You can print many more characters, but take care that appending thousands of lines to the textbox will be slow, and useless as you can't read thousands of lines. So, I recommend you to view ranges of characters, where each range contains 100 characters only. You can easily add a textbox to enter the range start in, and modify the above code to make the for loop start from the range start and end at range start + 99.

Chars.GetCharacterCode(character) As Double

Gets the character code for the given a Unicode character.

Parameter:

- **character:**

The character whose code is requested. This can be a 2 characters string that represents a pair of surrogates of a Unicode character. Such characters has a Unicode greater than 65535.

Returns:

A Unicode based code that corresponds to the character specified.

Example:

The Unicode 100000 is represented with a pair of surrogates. We don't need to be aware of that when we use the GetCharacter and GetCharacterCode methods to do the conversions for us. The following examples shows that converting the Unicode 100000 to a character will return a 2 length string, and you can send this string back to the GetCharacterCode method to get the Unicode 100000 again!

```
x = Chars.GetCharacter(100000)
TextWindow.WriteLine(Text.GetLength(x)) '2
c = Chars.GetCharacterCode(x)
TextWindow.WriteLine(c) ' 100000
```

⌚ Chars.IsDigit(character) As Boolean

Checks if the given character is a digit (0-9).

Parameter:

- **character:**

The character to check.

Returns:

True or False.

Examples:

```
TextWindow.WriteLine(Chars.IsDigit("12")) ' False
TextWindow.WriteLine(Chars.IsDigit(3)) ' True
TextWindow.WriteLine(Chars.IsDigit("3")) ' True
TextWindow.WriteLine(Chars.IsDigit(-3)) ' False
TextWindow.WriteLine(Chars.IsDigit("a")) ' False
```

⌚ Chars.IsLetter(character) As Boolean

Checks if the given character is a letter in any language.

Parameter:

- **character:**

The character to check.

Returns:

True or False.

Examples:

```
TextWindow.WriteLine(Chars.IsLetter("ab")) ' false
TextWindow.WriteLine(Chars.IsLetter(3))    ' false
TextWindow.WriteLine(Chars.IsLetter("3"))   ' false
TextWindow.WriteLine(Chars.IsLetter(-3))   ' false
TextWindow.WriteLine(Chars.IsLetter("a"))   ' True
TextWindow.WriteLine(Chars.IsLetter("A"))   ' True
TextWindow.WriteLine(Chars.IsLetter("("))  ' false
```

The CheckBox Type

Represents a CheckBox control, that the user can check or uncheck.

Use the Checked property and [OnCheck](#) event to respond to the user choices.

You can use the form designer to add a check box to the form by dragging it from the toolbox.

It is also possible to use the [Form.AddCheckBox](#) method to create a new check box and add it to the form at runtime.

Note that the CheckBox control inherits all the properties, methods and events of the [Control](#) type.

Besides, the CheckBox control has some new members that are listed below:

CheckBox.AllowThreeState As Boolean

Controls what happens when the user clicks the checkbox to change its state:

- When it is False (which is the default value):
The checkbox state goes from checked to unchecked then back to checked and so on.
- When it is True, the checkbox state goes from checked to dimmed (indeterminate) to unchecked then back to checked and so on.

Unchecke

Checked

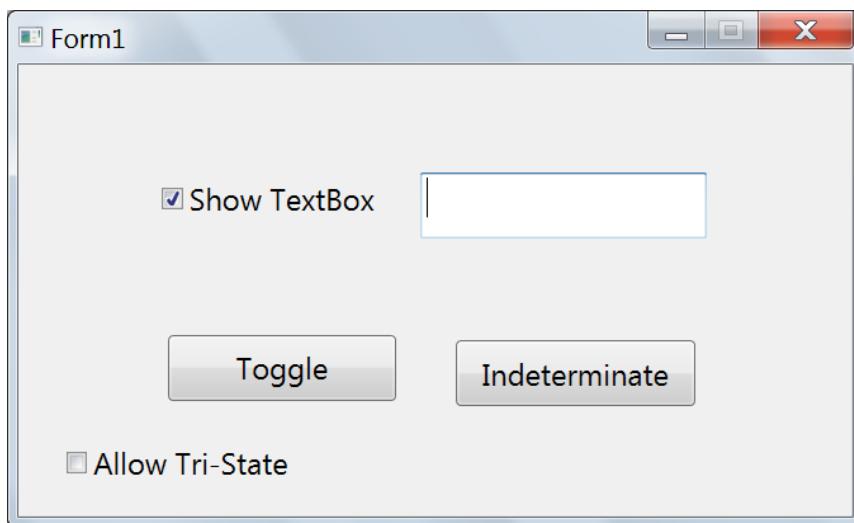
Indeterminate

Note that you can set the dimmed (indeterminate) state from code by setting the Checked property to an empty string "".

This will always work regardless the value of AllowTriState, which has effect on the user interaction only.

Example:

Try the "Allow Tri-State" checkbox in the "CheckBox Sample" project in the sVB samples folder. If this checked box is checked, the "Show TextBox" checkbox will be a tri-state, otherwise it will be a normal binary state.



CheckBox.Checked As Boolean

Gets or sets the checked state of the checkbox. It has three possible values:

- True: the check box is checked.
- False: the check box is unchecked.
- Empty string "": the check box is not checked nor unchecked (indeterminate state)

Examples:

Try the "CheckBox Sample" project in the sVB samples folder. The "Toggle" button changes the checked state of the "Show TextBox" checkbox from checked to unchecked and vice versa, by executing this code:

```
If ChkShowText.Checked Then  
    ChkShowText.Checked = False  
Else  
    ChkShowText.Checked = True  
EndIf
```

And the "Indeterminate" button changes the checked state of the "Show TextBox" checkbox to be indeterminate, by executing this code:

```
ChkShowText.Checked = ""
```

CheckBox.OnCheck

This event is fired when the checked state of the checkbox is changed, whether by user or from code.

Note that this is the default event for the CheckBox control, so, you can double-click any CheckBox on the form designer (say CheckBox1 for example), to get this event handler written for you in the code editor:

```
Sub CheckBox1_OnCheck()
```

```
EndSub
```

For more info, read about [handling control events](#).

Example:

Try the "CheckBox Sample" project in the sVB samples folder. The "Show TextBox" checkbox changes the textbox the Visible

and Enabled properties, by executing this code in the OnCheck event handler:

```
Sub ChkShowText_OnCheck()
    state = ChkShowText.Checked
    If state Then                                ' Checked
        TextBox1.Visible = True
        TextBox1.Enabled = True
    ElseIf state = "" Then                      ' Indeterminate
        TextBox1.Visible = True
        TextBox1.Enabled = False
    Else                                         ' Unchecked
        TextBox1.Visible = False
        TextBox1.Enabled = False
    EndIf
EndSub
```

CheckBox.Text As String

Gets or sets the text that is displayed by the CheckBox.

Example:

```
ChkShowText.Text = "Show TextBox"
```

CheckBox.Underlined As Boolean

Gets or sets whether or not to draw a line under the text.

Example:

```
ChkShowText.Text = "Show TextBox"
ChkShowText.Underlined = True
```

CheckBox.WordWrap As Boolean

Gets or sets whether or not to the text will be continue on the next line if it exceeds the width of the control.

The default value is True, so the check box title will be wrapped over lines by default. You can use the [CheckBox.FitContentHeight](#) method to ensure that the checkbox height fits the wrapped lines.

Example:

Add two checkboxes on the form, and add this code to the form global area:

```
CheckBox1.Text = "Test wordwrap property"  
CheckBox1.FitContentHeight()  
CheckBox2.Text = "Test wordwrap property"  
CheckBox2.WordWrap = False
```

The result will be as shown in the picture:



The Clock Type

This class provides access to the system clock. It provides properties to return the current system date and time, and to get the parts of this date and time.

The Clock type has these members:

Clock.Date As Date

Gets the current system date.

Clock.Day As Double

Gets the current day of the month.

Clock.ElapsedMilliseconds As Double

Gets the number of milliseconds that have elapsed since 1900.

You can use this property to calculate the duration of a certain task, by calling this method at the start of the task, and again at the end of the task, then calculate the deference between the two values, which is the time span of the task in milliseconds.

Example:

This code calculates the duration of executing a 1 million iteration loop:

```
t1 = Clock.ElapsedMilliseconds  
For i = 1 To 1000000  
    x = i  
Next  
t2 = Clock.ElapsedMilliseconds  
TextWindow.WriteLine((t2 - t1) + " ms.")
```

Clock.Hour As Double

Gets the current Hour.

Clock.Millisecond As Double

Gets the current Millisecond.

Clock.Minute As Double

Gets the current Minute.

Clock.Month As Double

Gets the current Month.

Clock.Second As Double

Gets the current Second.

Clock.Time As Date

Gets the current system time. It will use the 12 hours clock plus the AM and PM part in English. If you want to show this part with your local language, use the Following code:

now = Date.Now

time = Now.LongTime

Clock.WeekDay As String

Gets the name of the current day of the week, in English. If you want to get the name with your own culture, use this code:

now = Date.Now

day = Now.DayName

Clock.Year As Double

Gets the current year.

The Color Type

Contains methods to create and modify colors.

Note that the color in sVB is actually a string carrying the hexadecimal RGB representation of the color, on the form #AARRGGBB, where:

- AA: 2 hex digits representing the alpha component of the color, which is an indicator of the color opacity, where 0 means a fully transparent color, while 255 means a fully opaque solid color.
- RR: 2 hex digits representing the red component of the color.
- GG: 2 hex digits representing the green component of the color.
- BB: 2 hex digits representing the blue component of the color.

For example, a **solid green color** is represented with the string **#FF008000**, while a half transparent lime color (**light green**) is represented with the string **#80008000**, where FF and 80 are the hexadecimal representation of 255 and 128 respectively.

You don't have to worry about this string, because:

- the Color type has all necessary methods to handle it.
- sVB deals with this string as a color object, so you can use the color methods as extension methods of this object (such as using `green.Name` in a following example).
- you can also assign this string to color properties of controls such as the BackColor and ForeColor properties, as we will see in next examples.

But you may see this color hex string sometimes when you display the color name, because there are millions of color shades, but only about 140 of them have defined English names (and you can

access them via the [Colors type](#) properties like Colors.Green), hence, nameless colors will be displayed with their hexadecimal representations.

The color methods are listed below:

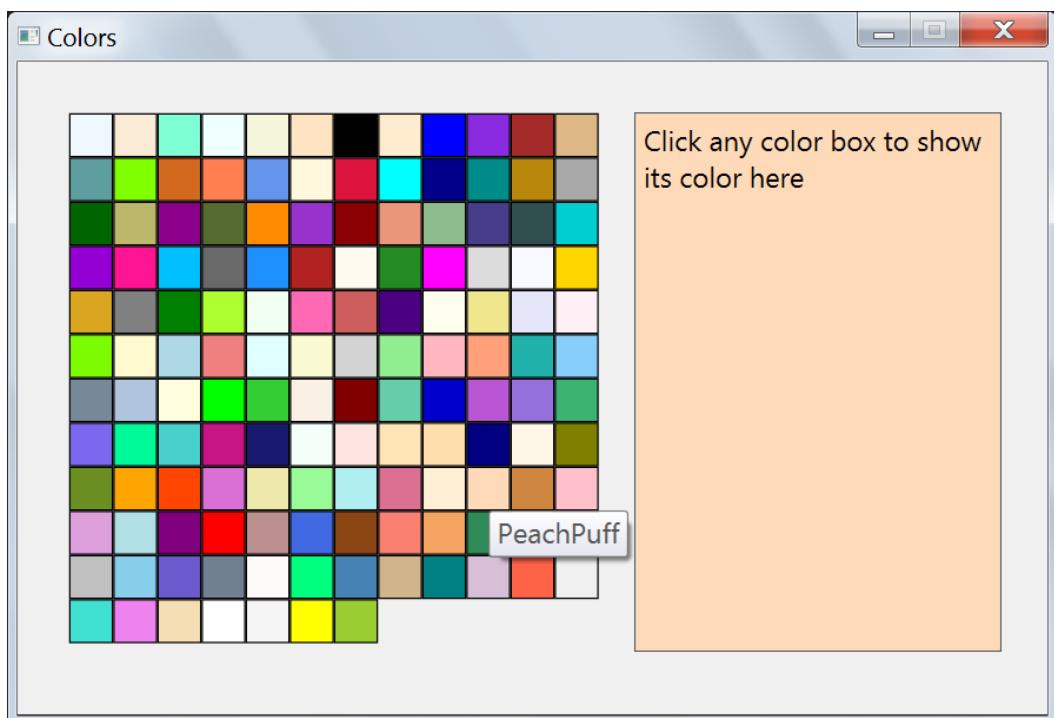
Color.AllColors As Array

Returns an array containing all pre-defined colors. These are the colors that have English names, and they are also defined as properties in the [Colors](#) type.

This property makes it easier for you to show a list with color names, or to display color boxes by a for loop.

Example:

This property is used in the "Color labels" project in the samples folder, to create a label for each color in runtime, and make it show the color name as a tool tip. When you click any label, its color will be displayed in the preview label.



Color.Alpha As Double

This is an extension property of color variables. It gets the alpha component of the color, which is an indicator of the color opacity.

This property is read only, and you must use [Color.ChangeAlpha](#) to change the color alpha component.

Returns:

A number between 0 and 255 that represents the alpha component of the color, where:

- 0 means a fully transparent color.
- 255 means a fully opaque solid color.

Examples:

```
C1 = Color.ChangeTransparency(Colors.Red, 10)
C2 = C1.ChangeTransparency(50)
C3 = C1.ChangeTransparency(100)
```

```
TextWindow.WriteLine(
    Color.GetAlpha(Colors.Red), '255
    C1.Alpha, ' 229
    C2.Alpha, ' 127
    C3.Alpha ' 0
)
```

Color.BlueRatio As Double

This is an extension property of color variables. It gets the blue component of the color.

This property is read only, and you must use [Color.ChangeBlueRatio](#) to change the color blue component.

Returns:

A number between 0 and 255 that represents the blue component of the color.

Examples:

```
x = Color.GetBlueRatio(Colors.Blue) ' 255  
y = Color.GetBlueRatio(Colors.Green) ' 0  
c = Color.FromRGB(120, 30, 100)  
z = c.BlueRatio ' 100
```

Color.ChangeAlpha(color, value) As Color

Creates a new color based on the given color, with the alpha component changed to the given value.

The alpha component is an indicator of the color opacity, where 0 means a fully transparent color, while 255 means a fully opaque solid color.

Parameters:

- **color:**

The input color. This argument can be omitted if you call this method as an extension method of a color variable.

- **value:**

A number between 0 and 255 that represents the new value of the alpha component.

Returns:

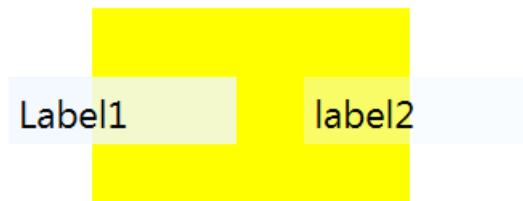
A new color with the alpha component changed to the given value.

Examples:

Put three labels on a form and try this:

```
C1 = Color.ChangeAlpha(Colors.AliceBlue, 204)
Label1.BackColor = Colors.Yellow
Label2.BackColor = C1
Label3.BackColor = C1.ChangeAlpha(102)
```

The picture shows you the result:



Color.ChangeBlueRatio(color, value) As Color

Creates a new color based on the given color, with the blue component changed to the given value.

Parameters:

- **color:**

The input color. This argument can be omitted if you call this method as an extension method of a color variable.

- **value:**

A number between 0 and 255 that represents the new value of the blue component.

Returns:

A new color with the blue component changed to the given value.

Examples:

Put two labels on a form and try this:

```
C1 = Color.ChangeBlueRatio(Colors.AliceBlue, 100)
Label1.BackColor = C1
Label2.BackColor = C1.ChangeBlueRatio(200)
```

The picture shows you the result:



⚙️ **Color.ChangeGreenRatio(color, value) As Color**

Creates a new color based on the given color, with the green component changed to the given value.

Parameters:

- **color:**

The input color. This argument can be omitted if you call this method as an extension method of a color variable.

- **value:**

A number between 0 and 255 that represents the new value of the green component.

Returns:

A new color with the green component changed to the given value.

Examples:

Put two labels on a form and try this:

```
C1 = Color.ChangeGreenRatio(Colors.AliceBlue, 100)
Label1.BackColor = C1
Label2.BackColor = C1.ChangeGreenRatio(200)
```

The picture shows you the result:



Color.ChangeRedRatio(color, value) As Color

Creates a new color based on the given color, with the red component changed to the given value.

Parameters:

- **color:**

The input color. This argument can be omitted if you call this method as an extension method of a color variable.

- **value:**

A number between 0 and 255 that represents the new value of the red component.

Returns:

A new color with the red component changed to the given value.

Examples:

Put two labels on a form and try this:

```
C1 = Color.ChangeRedRatio(Colors.AliceBlue, 100)  
Label1.BackColor = C1  
Label2.BackColor = C1.ChangeRedRatio(200)
```

The picture shows you the result:



Color.ChangeTransparency(color, percentage) As Color

Creates a new color based on the given color, with the transparency set to the given value.

Parameters:

- **color:**

The color you want to change its transparency.

- **percentage:**

A number between 0 and 100 that represents the new percentage of the color transparency.

Returns:

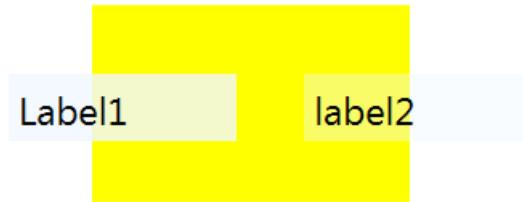
A new color with the given transparency.

Examples:

Put three labels on a form and try this:

```
C1 = Color.ChangeTransparency(Colors.AliceBlue, 20)
Label1.BackColor = Colors.Yellow
Label2.BackColor = C1
Label3.BackColor = C1.ChangeTransparency(60)
```

The picture shows you the result:



Color.FromARGB(alpha, red, green, blue) As Color

Creates a color from its alpha, red, green, and blue components.

Parameters:

- **alpha:**

A number between 0 and 255 that represents the color opacity:

- 0 means a fully transparent color.
- 127 or 128 means a half transparent color.
- 255 means a fully opaque solid color.

- **red:**

A number between 0 and 255 that represents the red component of the color.

- **green:**

A number between 0 and 255 that represents the green component of the color.

- **blue:**

A number between 0 and 255 that represents the blue component of the color.

Returns:

The new color.

Examples:

```
green = Color.FromARGB(255, 0, 128, 0)
tGreen = Color.FromARGB(128, 0, 128, 0)
lime = Color.FromARGB(255, 0, 255, 0)
tLime = Color.FromARGB(128, 0, 255, 0)
TextWindow.WriteLine(green.Name)
TextWindow.WriteLine(tGreen.NameAndTransparency)
TextWindow.WriteLine(lime.Name)
TextWindow.WriteLine(tLime.NameAndTransparency)
```

After executing the above code, the TextWindow will show:

Green

Green (50%)

Lime

Lime (50%)

Color.FromRGB(red, green, blue) As Color

Creates a color from its red, green, and blue components.

Parameters:

- **red:**

A number between 0 and 255 that represents the red component of the color.

- **green:**

A number between 0 and 255 that represents the green component of the color.

- **blue:**

A number between 0 and 255 that represents the blue component of the color.

Returns:

An opaque solid color with its alpha component set to 255.

Example:

```
green = Color.FromRGB(0, 128, 0)
lime = Color.FromRGB(0, 255, 0)
TextWindow.WriteLine(green)      ' #FF008000
TextWindow.WriteLine(green.Name) ' Green
TextWindow.WriteLine(lime)       ' #FF00FF00
TextWindow.WriteLine(lime.Name)  ' Lime
```

Color.GetAlpha(color) As Double

Gets the alpha component of the color, which is an indicator of the color opacity.

Parameters:

- **color:**

The input color. This argument can be omitted if you call this method as an extension property (with the name Alpha) of a color variable.

Returns:

A number between 0 and 255 that represents the alpha component of the color, where:

- 0 means a fully transparent color.
- 255 means a fully opaque solid color.

Examples:

```
C1 = Color.ChangeTransparency(Colors.Red, 10)  
C2 = C1.ChangeTransparency(50)  
C3 = C1.ChangeTransparency(100)
```

```
TextWindow.WriteLine({  
    Color.GetAlpha(Colors.Red), '255  
    C1.Alpha, ' 229  
    C2.Alpha, ' 127  
    C3.Alpha ' 0  
})
```

Color.GetBlueRatio(color) As Double

Gets the blue component of the color.

Parameters:

- **color:**

The input color. This argument can be omitted if you call this method as an extension property (with the name BlueRatio) of a color variable.

Returns:

A number between 0 and 255 that represents the blue component of the color.

Examples:

```
x = Color.GetBlueRatio(Colors.Blue) ' 255  
y = Color.GetBlueRatio(Colors.Green) ' 0  
c = Color.FromRGB(120, 30, 100)  
z = c.BlueRatio ' 100
```

Color.GetGreenRatio(color) As Double

Gets the green component of the color

Parameter:

- **color:**

The input color. This argument can be omitted if you call this method as an extension property (with the name GreenRatio) of a color variable.

Returns:

A number between 0 and 255 that represents the green component of the color.

Examples:

```
x = Color.GetGreenRatio(Colors.Blue) ' 0  
y = Color.GetGreenRatio(Colors.Green) ' 128  
c = Color.FromRGB(120, 30, 100)  
z = c.GreenRatio ' 30
```

Color.GetName(color) As String

Returns the English name of the color if its defined.

Parameter:

- **color:**

The color you want to get its name. This argument can be omitted if you call this method as an extension property (with the name "Name") of a color variable.

Returns:

The English name of the color if defined. If the known color has a transparency ratio it will be ignored, so, a blue color will return the name "Blue", and the a transparent blue color will also return the name "Blue".

If the color name is known, this method will return its hexadecimal string representation.

Examples:

```
tBlue = Color.FromARGB(127, 0, 0, 255)
c1 = Color.FromRGB(111, 222, 255)
c2 = c1.ChangeTransparency(50)

TextWindow.WriteLine({
    Color.GetName(Colors.Blue),      ' Blue
    tBlue.Name,                     ' Blue
    c1.Name,                        ' #FF6FDEFF
    c2.Name                         ' #7F6FDEFF
})
```

Color.GetNameAndTransparency(color) As String

Returns the English name of the color if its defined, followed by the transparency percentage of the color.

Parameter:

- **color:**

The color you want to get its name. This argument can be omitted if you call this method as an extension property (with the name NameAndTransparency) of a color variable.

Returns:

If the color has a known English name, it will be returned followed by the transparency percentage of the color like "Red (100%)". If the transparency is 0, it will not be displayed, and the return value will be the color name only.

If the color name is known, this method will return its hexadecimal string representation, followed by the transparency percentage (if not 0).

Examples:

```
tBlue = Color.FromARGB(127, 0, 0, 255)
c1 = Color.FromRGB(111, 222, 255)
c2 = c1.ChangeTransparency(50)

TextWindow.WriteLine({
    Color.GetNameAndTransparency(Colors.Blue), ' Blue
    tBlue.NameAndTransparency,           ' Blue (50%
    c1.NameAndTransparency,            '#FF6FDEFF
    c2.NameAndTransparency           '#7F6FDEFF (50%
})
```

Color.GetRandomColor() As Color

Selects a color randomly from the list of well-known colors, which contains 139 colors.

Alternatively, you can use the Colors.Random property, which calls this method, not only to shorten code, but also because the color members are displayed in the auto-completion list where a color is expected, so its easier and faster to type r and use Random from the list!

Returns:

A valid random color.

Example:

This method is used in the "Random Buttons" project in the samples folder, to give each new button a random fore and back colors. You can see these two lines in the CreateRndButton function in the Form1.sb file:

```
newButton.BackColor = Color.GetRandomColor()  
newButton.ForeColor = Colors.Random
```

Color.GetRedRatio(color) As Double

Gets the red component of the color

Parameter:

- **color:**

The input color. This argument can be omitted if you call this method as an extension property (with the name RedRatio) of a color variable.

Returns:

A number between 0 and 255 that represents the red component of the color,

Examples:

```
x = Color.GetRedRatio(Colors.Blue)      ' 0
y = Color.GetRedRatio(Colors.Red)        ' 255
c = Color.FromRGB(120, 30, 100)
z = c.RedRatio                          ' 120
```

Color.GetTransparency(color) As Double

Returns the transparency percentage of the color.

Parameter:

- **color:**

The color you want to get its transparency percentage. This argument can be omitted if you call this method as an extension property (with the name Transparency) of a color variable.

Returns:

A 0 to 100 value that represents the transparency percentage of the color.

Examples:

```
C1 = Color.FromARGB(127, 35, 135, 235)
C2 = Color.ChangeAlpha(C1, 128)
C3 = C1.ChangeAlpha(0)
TextWindow.WriteLine(
    Color.GetTransparency(Colors.AliceBlue), '0
    C1.Transparency,                      '50
```

C2. Transparency,
C3. Transparency
})

'50
'100

Color.GreenRatio As Double

This is an extension property of color variables. It gets the green component of the color.

This property is read only, and you must use [Color.ChangeGreenRatio](#) to change the color green component.

Returns:

A number between 0 and 255 that represents the green component of the color.

Examples:

```
x = Color.GetGreenRatio(Colors.Blue) ' 0
y = Color.GetGreenRatio(Colors.Green) ' 128
c = Color.FromRGB(120, 30, 100)
z = c.GreenRatio ' 30
```

Color.Name As String

This is an extension property of color variables. It gets the English name of the color if its defined.

Returns:

The English name of the color if defined. If the known color has a transparency ratio it will be ignored, so, a blue color will return the name "Blue", and the a transparent blue color will also return the name "Blue".

If the color name is known, this method will return its hexadecimal string representation.

Examples:

```
blue = Color.FromARGB(127, 0, 0, 255)
c1 = Color.FromRGB(111, 222, 255)
c2 = c1.ChangeTransparency(50)

TextWindow.WriteLine({
    Color.GetName(Colors.Blue),      ' Blue
    blue.Name,                      ' Blue
    c1.Name,                        '#FF6FDEFF
    c2.Name                         '#7F6FDEFF
})
```

Color.NameAndTransparency As String

This is an extension property of color variables. It gets the English name of the color if its defined, followed by the transparency percentage of the color.

Returns:

If the color has a known English name, it will be returned followed by the transparency percentage of the color like "Red (100%)". If the transparency is 0, it will not be displayed, and the return value will be the color name only.

If the color name is known, this method will return its hexadecimal string representation, followed by the transparency percentage (if not 0).

Examples:

```
tBlue = Color.FromARGB(127, 0, 0, 255)
c1 = Color.FromRGB(111, 222, 255)
c2 = c1.ChangeTransparency(50)

TextWindow.WriteLine({
    Color.GetNameAndTransparency(Colors.Blue), ' Blue
```

```
tBlue.NameAndTransparency,      ' Blue (50%)  
c1.NameAndTransparency,        ' #FF6FDEFF  
c2.NameAndTransparency         ' #7F6FDEFF (50%)  
})
```

Color.RedRatio As Double

This is an extension property of color variables. It gets the red component of the color.

This property is read only, and you must use [Color.ChangeRedRatio](#) to change the color red component.

Returns:

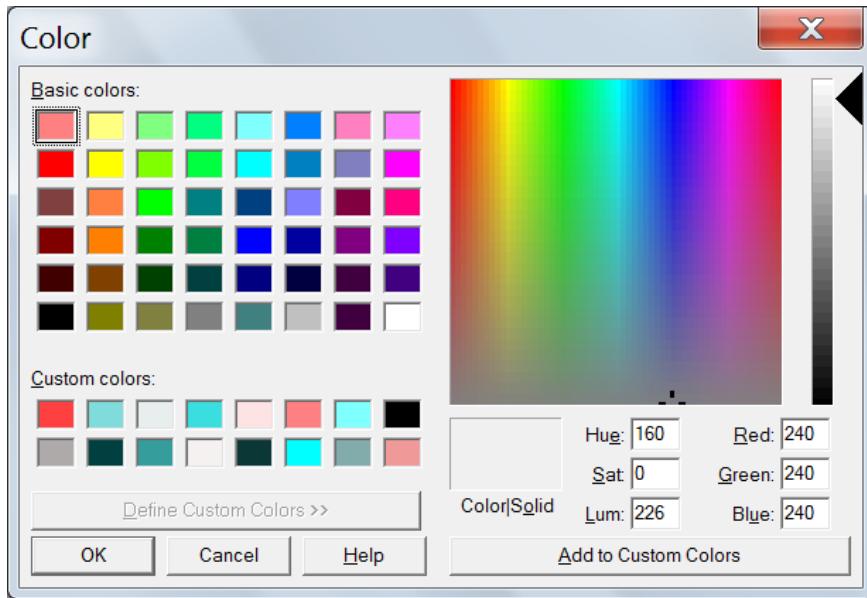
A number between 0 and 255 that represents the red component of the color.

Examples:

```
x = Color.GetRedRatio(Colors.Blue)    ' 0  
y = Color.GetRedRatio(Colors.Red)      ' 255  
c = Color.FromRGB(120, 30, 100)  
z = c.RedRatio                      ' 120
```

Color.ShowDialog(initialColor) As Color

Shows the color dialog to allow the user to select a color.



For simplicity, each control has the [ChooseForeColor](#) and [ChooseBackColor](#) methods to show the dialog and change the fore and back colors of that control directly.

Parameter:

- **initialColor:**

The color that will be selected when the dialog is opened.

Returns:

- If the user selected a color and clicks the OK button, this method will return the selected color.
- If the user canceled the color dialog, this method returns an empty string "".

Examples:

Add a button on the form and add this code to its OnClick event handler, to allow the user to change the back color of the from:

```
selectedColor = Color.ShowDialog(Me.BackColor)
```

```
If selectedColor <> "" Then  
    Me.BackColor = selectedColor  
EndIf
```

Note that the above code can be simplified to this single line of code:

```
Me.ChooseBackColor()
```

Color.Transparency As Double

This is an extension property of color variables. It gets the transparency percentage of the color.

Note that this property is read only, and you must use the [Color.ChangeTransparency](#) method to change the color transparency.

Returns:

A 0 to 100 value that represents the transparency percentage of the color.

Examples:

```
C1 = Color.FromARGB(127, 35, 135, 235)
C2 = Color.ChangeAlpha(C1, 128)
C3 = C1.ChangeAlpha(0)
TextWindow.WriteLine({
    Color.GetTransparency(Colors.AliceBlue), '0
    C1.Transparency,                      '50
    C2.Transparency,                      '50
    C3.Transparency                       '100
})
```

The Colors Type

Defines all known color names, and all system colors.

You can use any of them as in the following example:

```
textBox1.ForeColor = Colors.Red
```

The following table summarizes the names of color properties, and their return values, and you can preview them in the "Color labels" project in the samples folder:

Name	Value	Red	Green	Blue
AliceBlue	#FFF0F8FF	240	248	255
AntiqueWhite	#FFFAEBD7	250	235	215
Aqua	#FF00FFFF	0	255	255
Aquamarine	#FF7FFFAD	127	255	212
Azure	#FFF0FFFF	240	255	255
Beige	#FFF5F5DC	245	245	220
Bisque	#FFFFE4C4	255	228	196
Black	#FF000000	0	0	0
BlanchedAlmond	#FFFFEBCD	255	235	205
Blue	#FF0000FF	0	0	255
BlueViolet	#FF8A2BE2	138	43	226
Brown	#FFA52A2A	165	42	42
BurlyWood	#FFDEB887	222	184	135
CadetBlue	#FF5F9EA0	95	158	160
Chartreuse	#FF7FFF00	127	255	0
Chocolate	#FFD2691E	210	105	30
Coral	#FFFF7F50	255	127	80
CornflowerBlue	#FF6495ED	100	149	237
Cornsilk	#FFFFF8DC	255	248	220
Crimson	#FFDC143C	220	20	60
Cyan	#FF00FFFF	0	255	255

Name	Value	Red	Green	Blue
DarkBlue	#FF00008B	0	0	139
DarkCyan	#FF008B8B	0	139	139
DarkGoldenrod	#FFB8860B	184	134	11
DarkGray	#FFA9A9A9	169	169	169
DarkGreen	#FF006400	0	100	0
DarkKhaki	#FFBDB76B	189	183	107
DarkMagenta	#FF8B008B	139	0	139
DarkOliveGreen	#FF556B2F	85	107	47
DarkOrange	#FFFF8C00	255	140	0
DarkOrchid	#FF9932CC	153	50	204
DarkRed	#FF8B0000	139	0	0
DarkSalmon	#FFE9967A	233	150	122
DarkSeaGreen	#FF8FBC8F	143	188	143
DarkSlateBlue	#FF483D8B	72	61	139
DarkSlateGray	#FF2F4F4F	47	79	79
DarkTurquoise	#FF00CED1	0	206	209
DarkViolet	#FF9400D3	148	0	211
DeepPink	#FFF1493	255	20	147
DeepSkyBlue	#FF00BFFF	0	191	255
DimGray	#FF696969	105	105	105
DodgerBlue	#FF1E90FF	30	144	255
FireBrick	#FFB22222	178	34	34
FloralWhite	#FFFFFFA0	255	250	240
ForestGreen	#FF228B22	34	139	34
Fuchsia	#FFFF00FF	255	0	255
Gainsboro	#FFDCDCDC	220	220	220
GhostWhite	#FFF8F8FF	248	248	255
Gold	#FFFFD700	255	215	0
Goldenrod	#FFDAA520	218	165	32

Name	Value	Red	Green	Blue
Gray	#FF808080	128	128	128
Green	#FF008000	0	128	0
GreenYellow	#FFADFF2F	173	255	47
Honeydew	#FFF0FFF0	240	255	240
HotPink	#FFFF69B4	255	105	180
IndianRed	#FFCD5C5C	205	92	92
Indigo	#FF4B0082	75	0	130
Ivory	#FFFFFFF0	255	255	240
Khaki	#FFF0E68C	240	230	140
Lavender	#FFE6E6FA	230	230	250
LavenderBlush	#FFFFFFF5	255	240	245
LawnGreen	#FF7CFC00	124	252	0
LemonChiffon	#FFFFFFACD	255	250	205
LightBlue	#FFADD8E6	173	216	230
LightCoral	#FFF08080	240	128	128
LightCyan	#FFE0FFFF	224	255	255
LightGoldenrodYellow	#FFFAFAD2	250	250	210
LightGray	#FFD3D3D3	211	211	211
LightGreen	#FF90EE90	144	238	144
LightPink	#FFFFB6C1	255	182	193
LightSalmon	#FFFFA07A	255	160	122
LightSeaGreen	#FF20B2AA	32	178	170
LightSkyBlue	#FF87CEFA	135	206	250
LightSlateGray	#FF778899	119	136	153
LightSteelBlue	#FFB0C4DE	176	196	222
LightYellow	#FFFFFFE0	255	255	224
Lime	#FF00FF00	0	255	0
LimeGreen	#FF32CD32	50	205	50
Linen	#FFFAF0E6	250	240	230

Name	Value	Red	Green	Blue
Magenta	#FFFF00FF	255	0	255
Maroon	#FF800000	128	0	0
MediumAquamarine	#FF66CDAA	102	205	170
MediumBlue	#FF0000CD	0	0	205
MediumOrchid	#FFBA55D3	186	85	211
MediumPurple	#FF9370DB	147	112	219
MediumSeaGreen	#FF3CB371	60	179	113
MediumSlateBlue	#FF7B68EE	123	104	238
MediumSpringGreen	#FF00FA9A	0	250	154
MediumTurquoise	#FF48D1CC	72	209	204
MediumVioletRed	#FFC71585	199	21	133
MidnightBlue	#FF191970	25	25	112
MintCream	#FFF5FFFA	245	255	250
MistyRose	#FFFFE4E1	255	228	225
Moccasin	#FFFFE4B5	255	228	181
NavajoWhite	#FFFFDEAD	255	222	173
Navy	#FF000080	0	0	128
None	None			
OldLace	#FFFDF5E6	253	245	230
Olive	#FF808000	128	128	0
OliveDrab	#FF6B8E23	107	142	35
Orange	#FFFFA500	255	165	0
OrangeRed	#FFFF4500	255	69	0
Orchid	#FFDA70D6	218	112	214
PaleGoldenrod	#FFEEE8AA	238	232	170
PaleGreen	#FF98FB98	152	251	152
PaleTurquoise	#FFAFEEEE	175	238	238
PaleVioletRed	#FFDB7093	219	112	147
PapayaWhip	#FFFFEFD5	255	239	213

Name	Value	Red	Green	Blue
PeachPuff	#FFFFDAB9	255	218	185
Peru	#FFCD853F	205	133	63
Pink	#FFFC0CB	255	192	203
Plum	#FFDDA0DD	221	160	221
PowderBlue	#FFB0E0E6	176	224	230
Purple	#FF800080	128	0	128
Random	Returns a random color			
Red	#FFFF0000	255	0	0
RosyBrown	#FFBC8F8F	188	143	143
RoyalBlue	#FF4169E1	65	105	225
SaddleBrown	#FF8B4513	139	69	19
Salmon	#FFFA8072	250	128	114
SandyBrown	#FFF4A460	244	164	96
SeaGreen	#FF2E8B57	46	139	87
Seashell	#FFFFFFEE	255	245	238
Sienna	#FFA0522D	160	82	45
Silver	#FFC0C0C0	192	192	192
SkyBlue	#FF87CEEB	135	206	235
SlateBlue	#FF6A5ACD	106	90	205
SlateGray	#FF708090	112	128	144
Snow	#FFFFFFFAFA	255	250	250
SpringGreen	#FF00FF7F	0	255	127
SteelBlue	#FF4682B4	70	130	180
Tan	#FFD2B48C	210	180	140
Teal	#FF008080	0	128	128
Thistle	#FFD8BF8D	216	191	216
Tomato	#FFFF6347	255	99	71
Transparent	#00FFFFFF	255	255	255
Turquoise	#FF40E0D0	64	224	208

Name	Value	Red	Green	Blue
Violet	#FFEE82EE	238	130	238
Wheat	#FFF5DEB3	245	222	179
White	#FFFFFF	255	255	255
WhiteSmoke	#FFF5F5F5	245	245	245
Yellow	255	255		0
YellowGreen	#FF9ACD32	154	205	50

There are 2 special colors that need your attention:

Colors.None As Color

Returns no color!

Use this value when you don't want to draw the background color of a control or a shape, or the outline color of a shape.

Colors.Transparent As Color

Returns a transparent white color (#00FFFFFF).

Note that there is a difference between Colors.None and Colors.Transparent:

- The None color deletes the surface of the graphic, so, it doesn't respond to mouse and keyboard events, which are delivered to the underneath control.
- The Transparent color keeps the surface of the graphic but you can see through it, while it still responds to mouse and keyboard events.

The Color type also contains properties that return system colors, which are defined on the user's PC in his Windows themes. Using

these colors allows you to make your application interface use the user themes.

For example, You can set the back color of the form to the system control-color like this:

```
Me.BackColor = Colors.SystemControl
```

All system colors start with the `System` prefix so you can easily find them in the auto-completion list.

System colors properties are listed below:

Colors.SystemActiveBorder As Color

Gets the color of active window's border as defined on the user system.

Colors.SystemActiveCaption As Color

Gets the background color of the active window's title bar, as defined on the user system.

Colors.SystemActiveCaptionText As Color

Gets the color of the text in the active window's title bar, as defined on the user system.

Colors.SystemAppWorkspace As Color

Gets the color of the application workspace, as defined on the user system.

Colors.SystemControl As Color

Gets the face color of a three-dimensional display element, as defined on the user system.

Colors.SystemControlDarkShadow As Color

Gets the dark shadow color of a three-dimensional display element, as defined on the user system.

Colors.SystemControlHighlight As Color

Gets the highlight color of a three-dimensional display element, as defined on the user system.

Colors.SystemControlLight As Color

Gets the light color of a three-dimensional display element, as defined on the user system.

Colors.SystemControlShadow As Color

Gets the shadow color of a three-dimensional display element, as defined on the user system.

Colors.SystemControlText As Color

Gets the color of text in a three-dimensional display element, as defined on the user system.

Colors.SystemDesktop As Color

Gets the color of the desktop, as defined on the user system.

Colors.SystemGradientActiveCaption As Color

Gets the right side color in the gradient of an active window's title bar, as defined on the user system.

Colors.SystemGradientInactiveCaption As Color

Gets the right side color in the gradient of an inactive window's title bar, as defined on the user system.

Colors.SystemGrayText As Color

Gets the color of disabled text, as defined on the user system.

Colors.SystemHighlight As Color

Gets the background color of selected items, as defined on the user system.

Colors.SystemHighlightText As Color

Gets the color of the text of selected items, as defined on the user system.

Colors.SystemHotTrack As Color

Gets the color used to designate a hot-tracked item, as defined on the user system.

Colors.SystemInactiveBorder As Color

Gets the color of an inactive window's border, as defined on the user system.

Colors.SystemInactiveCaption As Color

Gets the background color of an inactive window's title bar, as defined on the user system.

Colors.SystemInactiveCaptionText As Color

Gets the color of the text of an inactive window's title bar, as defined on the user system.

Colors.SystemInactiveSelectionHighlight As Color

Gets the color used on the user's system to highlight a selected item that is inactive.

Colors.SystemInactiveSelectionHighlightText As Color

Gets the color of an inactive selected item's text, as defined on the user system.

Colors.SystemInfo As Color

Gets the background color for the ToolTip control, as defined on the user system.

Colors.SystemInfoText As Color

Gets the text color for the ToolTip control, as defined on the user system.

Colors.SystemMenu As Color

Gets the color of a menu's background, as defined on the user system.

Colors.SystemMenuBar

Gets the background color for a menu bar, as defined on the user system.

Colors.SystemMenuHighlight As Color

Gets the color used to highlight a menu item, as defined on the user system.

Colors.SystemMenuText As Color

Gets the color of a menu's text, as defined on the user system.

 **Colors.SystemScrollBar As Color**

Gets the background color of a scroll bar, as defined on the user system.

 **Colors.SystemWindow As Color**

Gets the background color in the client area of a window, as defined on the user system.

 **Colors.SystemWindowFrame As Color**

Gets the color of a window frame, as defined on the user system.

 **Colors.SystemWindowText As Color**

Gets the color of the text in the client area of a window, as defined on the user system.

The ComboBox Type

Represents a ComboBox control, which is composed of a textbox and a dropdown listbox.

You can use the form designer to add a combo box to the form by dragging it from the toolbox.

It is also possible to use the [Form.AddComboBox](#) method to create a new combo box and add it to the form at runtime.

Note that the ComboBox control inherits all the properties, methods and events of the [Control](#) type.

Besides, the ComboBox control has some new members that are listed below:

ComboBox.AddItem(value) As Double

Adds an item at the end of the list.

Parameter:

- **value:**

The item you want to add to the list. You can send an array to add all its items.

Returns:

The index of the newly added item, or 0 if the operation failed.

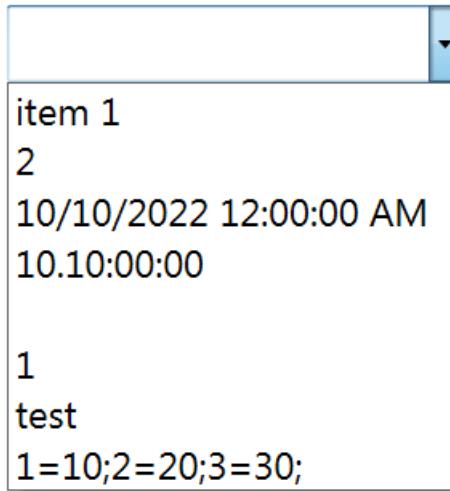
Examples:

Add a ComboBox on the form, and add the following code to the form global area:

```
ComboBox1.AddItem("item 1")
ComboBox1.AddItem(2)
```

```
ComboBox1.AddItem(#10/10/2022#)
ComboBox1.AddItem(#+10.10:00#)
ComboBox1.AddItem("")
ComboBox1.AddItem({1, "test", {10, 20, 30}})
```

When you run the program, the combo box will be populated with the items shown in the picture:



As you can see, the combo box can show strings, dates and durations (time spans). You can even add an empty item, like the fifth item, which is the result of adding empty string "" to the combo box.

It is also obvious that the array added with the last line of code is added to the combo box as the three items:

```
1
test
1=10;2=20;3=30;
```

Where the last one is the sVB string representation of the array {10, 20, 30}. This means that only first level items are added to the combo box as single items, but the AddItem method will not go deeper in the inner arrays. It just adds the inner array as a single item, which appears as you see, while

you can still read this item as an array. This is just the how the combo box displays it.

In fact, this is not a user friendly way, so, don't add an array as a single item.

ComboBox.AddItemAt(value, index)

Adds the given item to the list at the given index.

Parameters:

- **value:**

The item you want to add to the list. You can send an array to add all its items to the combo box starting from the given index.

- **index:**

The index you want to add the item at. The value of this index must be greater than 0 and less than list items count + 1, otherwise the item will not be added.

Returns:

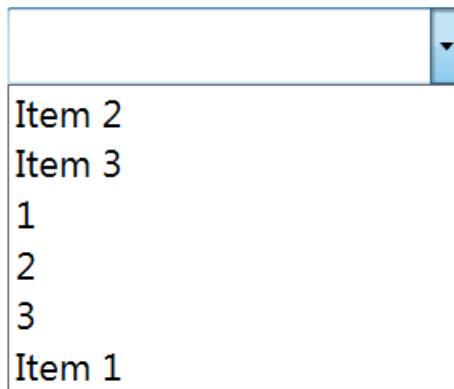
True if the item is successfully added at the given index, otherwise False.

Examples:

Add a ComboBox on the form, and add the following code to the form global area:

```
ComboBox1.AddItemAt("Item 1", 1)
ComboBox1.AddItemAt("Item 2", 1)
ComboBox1.AddItemAt("Item 3", 2)
ComboBox1.AddItemAt("Item 4", 5)
ComboBox1.AddItemAt({1, 2, 3}, 3)
```

When you run the program, the combo box will be populated with the items shown in the picture:



note that "Item 4" was not added, because we asked to add it outside the possible list indexes.

ComboBox.AllowEdit As Boolean

Set this property to False to prevent the user from writing in the textbox of the ComboBox. The default value is True.

Examples:

Add 3 combo boxes on the form, and write the following code in the form global area:

```
ComboBox1.AddItem({1, 2, 3})  
ComboBox2.AddItem({1, 2, 3})  
ComboBox3.AddItem({1, 2, 3})  
  
ComboBox2.AllowEdit = False  
ComboBox3.AllowEdit = False  
ComboBox3.BackColor = Colors.AliceBlue
```

When you run the program, the combo boxes will look like this:

The comboBox1 is editable, while comboBox2 and comboBox3 are not, as they don't display a textbox. You may dislike the 3D gray style of comboBox2, so, you can get rid of it just by changing the back color, as we did with comboBox3.

⌚ComboBox.ContainsItem(value) As Boolean

Checks whether or not the given item exists in the list.

Parameter:

- **value:**

The value of the item to

Returns:

True if the item exists, or False otherwise.

Examples:

Add a combo box on the form, and write the following code in the form global area:

```
ComboBox1.AddItem({1, 2, 3})  
TextWindow.WriteLine(ComboBox1.ContainsItem(2))  
TextWindow.WriteLine(ComboBox1.ContainsItem(5))
```

When you run the program, the text window will show the following results:

True

False

ComboBox.FindItem(value) As Double

Returns the index of the given item if it exists in the list, otherwise returns 0.

Parameter:

- **value:**

The item you want to find.

Returns:

The index of the item if found, or 0 otherwise.

Examples:

Add a combo box on the form, and write the following code in the form global area:

```
ComboBox1.AddItem({1, 2, 3})  
TextWindow.WriteLine(ComboBox1.FindItem(2))  
TextWindow.WriteLine(ComboBox1.FindItem(5))
```

When you run the program, the text window will show the following results:

2
0

ComboBox.FindItemAt(value, startIndex, endIndex) As Double

Returns the index of the given item if it exists in the list in the given index range, otherwise returns 0.

Parameters:

- **value:**

The item you want to find.

- **startIndex:**

The array index the search starts at.

- **endIndex:**

The array index the search ends at. If endIndex < startIndex, the search direction will be reversed to find the last index of the item.

Returns:

The index of the item if found, or 0 otherwise.

Examples:

Add a combo box on the form, and write the following code in the form global area:

```
ComboBox1.AddItem({1, 2, 3, 2, 4, 5})  
TextWindow.WriteLine(ComboBox1.FindItemAt(2, 1, 4))  
TextWindow.WriteLine(ComboBox1.FindItemAt(2, 3, 5))  
TextWindow.WriteLine(ComboBox1.FindItemAt(2, 5, 6))  
TextWindow.WriteLine(ComboBox1.FindItemAt(2, 6, 1))
```

When you run the program, the text window will show the following results:

```
2  
4  
0  
4
```

ComboBox.GetItemAt(index) As String

Returns the item that has the given index.

Parameter:

- **index:**

The index of the item. It should be greater than zero and not exceed the count of items, otherwise, this method will return an empty string.

Returns:

The value of the item if the index is valid, or "" otherwise.

Examples:

Add a combo box on the form, and write the following code in the form global area:

```
ComboBox1.AddItem({1, 2, 3})  
TextWindow.Write("[")  
TextWindow.Write(ComboBox1.GetItemAt(0) + ", ")  
TextWindow.Write(ComboBox1.GetItemAt(3) + ", ")  
TextWindow.WriteLine(ComboBox1.GetItemAt(5) + "])")
```

When you run the program, the text window will show:

[, 3,]

ComboBox.Items As Array

Gets an array containing the items of the ComboBox.

Examples:

Add a combo box on the form, and write the following code in the form global area:

```
ComboBox1.AddItem({1, 2, 3})  
items = ComboBox1.Items  
TextWindow.WriteLine(items.ToString())
```

When you run the program, the text window will show:

{1, 2, 3}

ComboBox.ItemsCount As Double

Gets the count of the items in the ComboBox.

Examples:

Add a combo box on the form, and write the following code in the form global area:

```
TextWindow.WriteLine(ComboBox1.ItemsCount)
ComboBox1.AddItem({1, 2, 3})
TextWindow.WriteLine(ComboBox1.ItemsCount)
```

When you run the program, the text window will show:

0

3

ComboBox.OnSelection

This event is fired when the selected item in the list is changed.

This event will also be fired when the user deselects all the items by entering a text in the textbox of the combo box that doesn't match any item.

Note that this is the default event for the ComboBox control, so, you can double-click any ComboBox on the form designer (say ComboBox1 for example), to get this event handler written for you in the code editor:

```
Sub ComboBox1_OnSelection()
```

```
EndSub
```

For more info, read about [handling control events](#).

Example:

This event is used in the "Validation" project in the samples folder, to re-validate the combo box when the selected item changed. You will find this code in that project:

```
Sub ComboBox1_OnSelection()
    ComboBox1.Validate()
EndSub
```

⌚ComboBox.RemoveAllItems()

Removes all the items from the combo box.

Example:

```
ComboBox1.RemoveAllItems()
```

⌚ComboBox.RemoveItem(value)

Searches for the given value in the list, and removes the first found item.

Parameter:

- **value:**

The item you want to remove. You can send an array to remove all its items.

Returns:

True if the item is successfully removed, or False if the item is not found.

Example:

Add a combo box on the form, and write the following code in the form global area:

```
ComboBox1.AddItem({2, 1, 2, 2, 2, 3, 4, 2, 5, 2})  
While ComboBox1.RemoveItem(2)  
Wend  
TextWindow.WriteLine(Array.ToString(ComboBox1.Items))
```

The above code shows you how to use the RemoveItem in a while loop to remove all items that have the value 2 from the combo box. After running this project, the text window will display:

{1, 3, 4, 5}

_COMBOBOX_RemoveItemAt(index)

Removes the list item that exists at the given index.

Parameter:

- **index:**

The index of the item you want to remove. You can send and array of indexes to remove them all, but be careful that removing an item changes the indexes of the following items, so it will be easier for you to arrange the indexes in a descending order, so that removing each item doesn't affect the indexes to be deleted next.

Returns:

True is the item is successfully removed, or False if the given index is < 1 or > list items count.

Examples:

Add a combo box on the form, and write the following code in the form global area:

```
ComboBox1.AddItem({1, 2, 2, 3, 4, 2, 5})  
ComboBox1.RemoveItemAt(6)  
ComboBox1.RemoveItemAt(2)  
TextWindow.WriteLine(Array.ToString(ComboBox1.Items))
```

The above code shows you how to use the RemoveItem in a while loop to remove all items that have the value 2 from the combo box.

Note that the above two RemoveAt calls can be done with only this single call:

```
ComboBox1.RemoveItemAt({6, 2})
```

After running this project, the text window will display:

```
{1, 2, 3, 4, 5}
```

ComboBox.SelectedIndex As Double

Gets or sets the index of the selected item in the ComboBox. Zero indicates that no item is selected.

Examples:

You can see the SelectedIndex property in action in the "Validation" project in the samples folder. It is used in the ComboBox1_OnLostFocus event handler to validate the combo box:

```
Sub ComboBox1_OnLostFocus()  
    If ComboBox1.SelectedIndex = 0 Then  
        ComboBox1.Error = "Please choose a number from the list"  
    Else  
        ComboBox1.Error = ""  
    EndIf  
EndSub
```

ComboBox.SelectedItem As String

Gets or sets the item that is currently selected in the ComboBox. If you set this property to a value that exists in the list (comparison is case-sensitive), the first item that has this value will be selected, otherwise the value is ignored, and the selected item doesn't change.

ComboBox.SetItemAt(index, value)

Sets the value of the item that exists at the given index in the list.

Parameters:

- **index:**

The index of the item. It should be greater than zero and not exceed the count of the items, otherwise, this method will return False.

- **value:**

The value to set to the item.

Returns:

True if the item is set, or False otherwise.

Examples:

Add a combo box on the form, and write the following code in the form global area:

```
ComboBox1.AddItem({1, 2, 3})
TextWindow.WriteLine(ComboBox1.SelectedItem)
TextWindow.WriteLine(ComboBox1.SetItemAt(0, 0))
TextWindow.WriteLine(ComboBox1.SetItemAt(3, 30))
TextWindow.WriteLine(ComboBox1.SetItemAt(4, 40))
TextWindow.WriteLine(Array.ToString(ComboBox1.Items))
```

When you run the program, the text window will show:

False
True
False
{1, 2, 30}

ComboBox.Text As String

Gets or sets the text that is displayed by the textbox of the ComboBox. If there is a selected item in the listbox, it will be displayed in the textbox, so the Text property will return the selected item.

If you set the Text property to a value that exists in the list, the first item that has this value will be selected.

You can also set the Text property to a value that doesn't exist in the list, but the result of this action will depend on the value of the [AllowEdit property](#):

- When AllowEdit = True:
the textbox will display the text.
- When AllowEdit = False:
the textbox will display an empty string "", but the Text property will keep the new value, and it can be displayed later if you changed the AllowEdit value to True.

In either case, no item will be selected in the list.

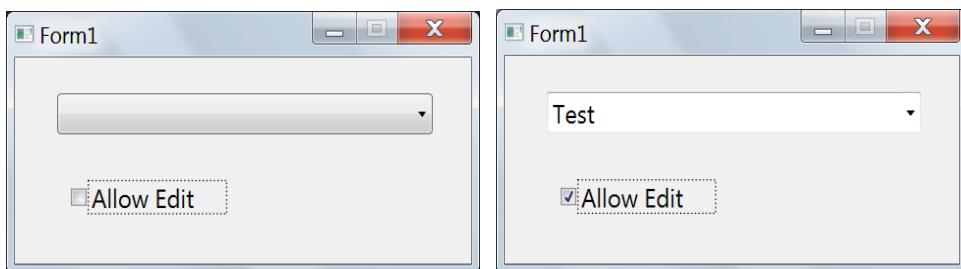
Examples:

Add a ComboBox and a CheckBox on the form, and write the following code in the form global area:

```
ComboBox1.Text = "Test"  
CheckBox1.Checked = True
```

```
Sub CheckBox1_OnCheck()  
    ComboBox1.AllowEdit = CheckBox1.Checked  
EndSub
```

Run the program, and see the effect of checking and unchecking the checkbox on the combo box. The difference is shown in the pictures:

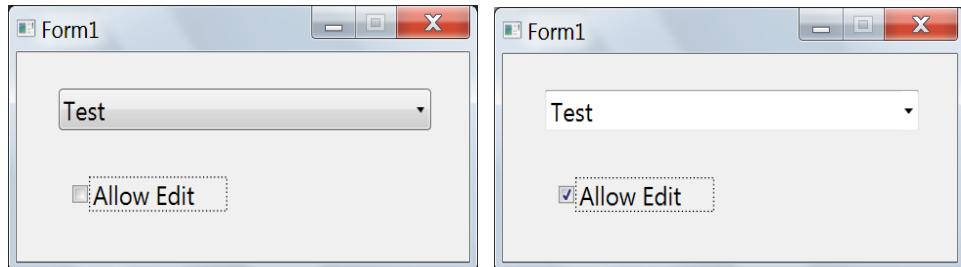


Stop the program, and add the following code line at the beginning of the previous code:

```
ComboBox1.AddItem {"abc", "Test", 100}
```

This will allow us to select the "Test" element when setting the text value to "Test".

Run the program, and see the effect of checking and unchecking the Checkbox on the combo box. The difference is shown in the pictures:



The Control Type

This is the parent control of other controls (like Form, Label, and Button controls). All controls inherit the properties, methods and events defined in this type, so you can use them with all controls. For example, You can use the Name property with different controls like this:

```
TextWindow.WriteLine(Me.Name)  
TextWindow.WriteLine(Button1.Name)  
TextWindow.WriteLine(Label1.Name)  
TextWindow.WriteLine(TextBox1.Name)
```

* **Animating Controls:**

The control class provides you with many animation methods, that you can use with any control to create lovely visual effects. These methods are:

1. [AnimateSize.](#)
2. [AnimatePos.](#)
3. [AnimateAngle.](#)
4. [AnimateColor.](#)
5. [AnimateTransparency.](#)

To see these methods in action, look at the "Animation 1, 2 and 3" projects in the samples folder.

It is important to note that in all these methods, the animation is executed asynchronously, so your app will continue running and the form will be responsive to the user while the control is being animated. Also, the lines of code next to the line that called the animation method, will be executed immediately after the animation starts. Pay a good attention to these notes:

- You can animate many controls at the same time.

- You can animate different properties of the same control at the same time.
- You can't apply more than one animation on the same control property at the same time, because when you start animating a property, it stops the current animation of this property and begins the new one. To solve this, you can delay the execution of the code until the animation ends, then start the new one. For example:

```
Form1.AnimateColor(Colors.Red, 1000)
Program.Delay(1000)
Form1.AnimateColor(Colors.Yellow, 2000)
```

The above code will animate the form backcolor to gradually change it from its current color to red during one second. The execution of code is delayed for one second to make sure that the animation is completed, then it starts a second animation to change the form back color gradually change it from red to yellow in two seconds.

- You should not change the animated property after starting the animation, because this will end the animation immediately. For example:

```
Form1.AnimateColor(Colors.Red, 1000)
Form1.BackColor = Colors.White
```

The above code will change the form back color to white immediately, and you will see no animation. So, you also need to delay executing the second command until the animation ends:

```
Form1.AnimateColor(Colors.Red, 1000)
Program.Delay(1000)
Form1.BackColor = Colors.White
```

- You now know how to stop the animation whenever you want 😊. Just add a cancel button, and in its OnClick event

handler change the property that is being animated to any value (See the " Cancel Animation" project in the samples folder).

- From the above notes, you can conclude that if you tried to read the animated property just after starting the animation, you will get its old value, hence you also need to wait until the animation ends to get the final value of the property. But you can also use a timer to display the intermediate values of that property while it is being animated. The " Cancel Animation" project in the samples folder shows you how to do that.

* **Handling Control Events:**

A windows application has a graphical user interface (GUI), which consists mainly one or more forms, with controls to get input from user and show results to him. The user inputs data by mouse and keyboard, so, it is essential for us to know what he is doing by these devices and when. We can know that from Events.

Events are messages that the Windows OS sends to the program to notify us that the user is doing something now, like clicking a button, pressing a key, or moving the mouse. We react to such events by executing the proper code that makes our application do its job. This is called Event Handling, and the code that responds to the event is called Event Handler, which is just a normal subroutine, that we tell sVB to call when the event is fired.

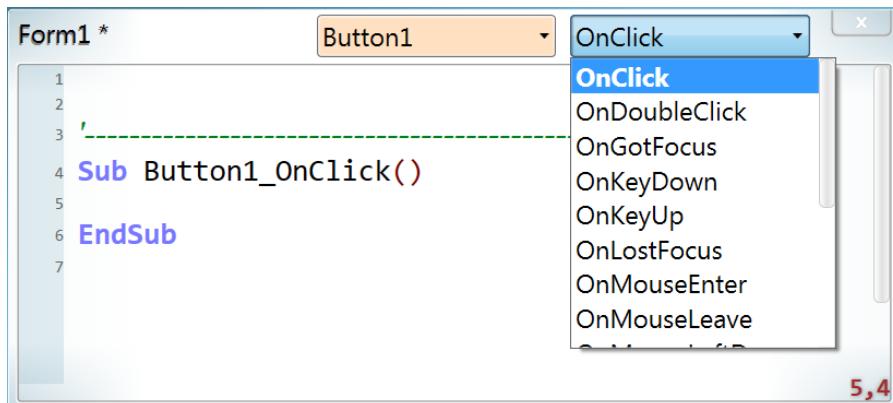
So, windows applications are mostly a bunch of event handlers that are waiting for the user to do something. This is why modern programming is called Event-driven programming.

In sVB, every control has its own events, to notify you about what the user is doing with it, like the OnTextChanged event that is fired when the user changes the textbox content.

All controls share some basic events like OnClick, OnMouseMove, and OnKeyDown, so, these events are defined in the Control type, because it is the parent of all controls, and they all inherit its members.

We will learn about these events in details in next paragraphs, but first, we need to learn how to handle events in sVB.

let's start by putting a textbox and a button on the form. Now double-click the button. This will open the code editor and insert the OnClick event handler in it:



The convention is to use the Control_Event name for the subroutine that handles the event, like the Button1_OnClick in our example. So, if you manually added the next subroutine to the code file, sVB will recognize it as an event handler for the TextBox1.TextChanged event:

```
Sub TextBox1_TextChanged()
    Me.Text = TextBox1.Text
```

EndSub

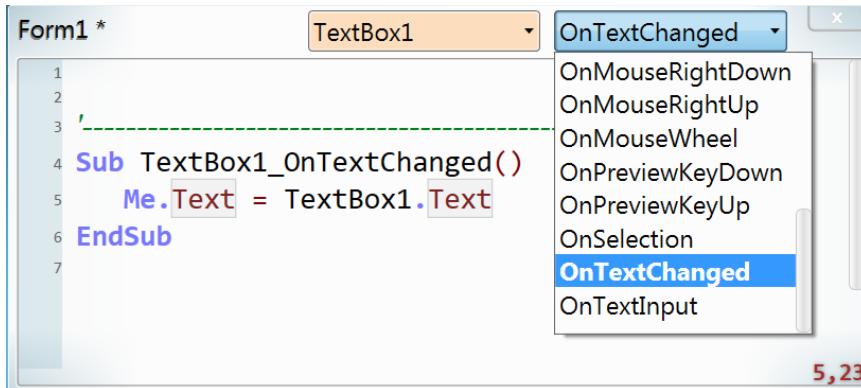
Now if you press F5 to run the program, every time you change the text written in the textbox, you will immediately see that text on the form's title bar.

But, you don't have to do this manually, as the text editor offers to do it for you. At the top of the code file, you see two drop down lists:

- The left list shows the form name and its controls.
- The right list shows the events available for the control that is currently selected in the first list.

So, you can simply select textBox1 from the left list, then select the OnTextChanged event from the right list, to get the handler added to the code for you.

Note that you can press 't' from the keyboard to select the OnTextChanged in the right list, instead of searching for it.



You can easily handle any event for any control.

But what if you want to write one subroutine to handle more than one event of the same control or the same event for many controls?

You may think of writing the code in one subroutine and call it from those event handlers, which is perfectly valid, but it will be unnecessarily verbose code.

Instead, we can tell sVB that only one subroutine can handle many events, but this can't be done by the code editor tools nor by naming convention, and we need to write some code to assign the handler to those events. This is exactly what is done in the `Mouse pos` project in the samples folder, where we told sVB that the subroutine called OnMouseMove will be used to handle the OnMouseMove event of the form and three other controls on it, by using this code:

```
TextBox1.OnMouseMove = OnMouseMove  
TextBox2.OnMouseMove = OnMouseMove  
Label1.OnMouseMove = OnMouseMove  
Me.OnMouseMove = OnMouseMove  
  
Sub OnMouseMove()  
    ' See the code in the app  
EndSub
```

When you use one handler to deal with events of different controls, you can use the Event.SenderControl property to get the control that fired the event. In general, there are three types that you will find useful when handling events to give you the necessary info about the event: [Event](#), [Mouse](#) and [Keyboard](#).

Note:

You can register only one handler to the event, whether by setting it by code or by using the naming convention. If you try to add more than one handler for the same event, only the last one will be used. But this applies only in the program domain, so you can add one more handler from an external library. For example, when you send a control to the Geometrics.AllowDrag method, it will handle the OnMouseMove and OnMouseLeftDown events of this control so that you can drag it on the form. Although, you

can still add a handler for OnMouseMove event and a handler for the OnMouseLeftDown event of this control in your program, which means that each of these events will call two handlers, one created by the Geometrics library, and the other is created by tour program.

The Control type has these members:

Control.Angle As Double

Gets or sets the rotation angle of the control. The angle is measured in degrees, and you can use negative values, and values greater than 360, but when you read this property, its value will be normalized to be in the range 0 to 259 degrees.

Note that you can change this angle by using the [rotation thumb](#) on the form designer.

Examples:

Add a ComboBox and a CheckBox on the form, and write the following code in the form global area:

```
ComboBox1.Angle = 400  
TextWindow.WriteLine(ComboBox1.Angle)  
CheckBox1.Angle = -45  
TextWindow.WriteLine(CheckBox1.Angle)
```

When you run the program, the text window will show:

40

315

Control.AnimateAngle(angle, duration)

Animates the control's rotation angle from its current value to the given new angle. That makes the control rotate around its center for the given duration.

For more details about how animation works and how you can control it, see the [Animating Controls](#) section.

Parameters:

- **angle:**

The new rotation angle, that the control will reach after the animation is finished.

- **duration:**

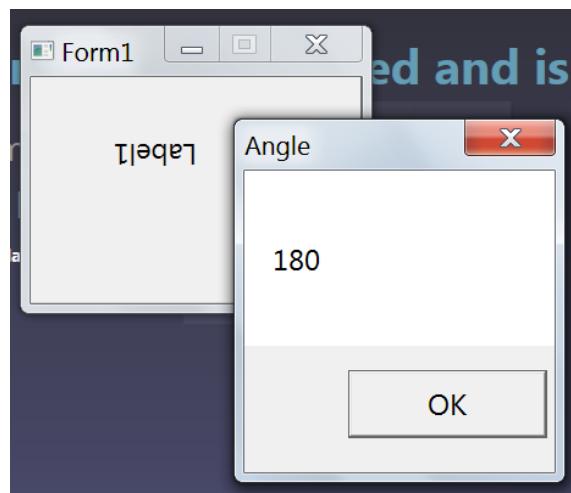
The time for the animation, in milliseconds.

Example:

Add a Label on the form, and write the following code in the form global area:

```
Label1.AnimateAngle(180, 1000)  
Program.Delay(1000)  
Forms.ShowMessage(Label1.Angle , "Angle")
```

When you run the program, the label will rotate clockwise for one second and stops at a 180 degree angle, then a message box will show the message "180".



Control.AnimateColor(newColor, duration)

Animates the control's back Color to a new color.

For more details about how animation works and how you can control it, see the [Animating Controls](#) section.

Parameters:

- **newColor:**

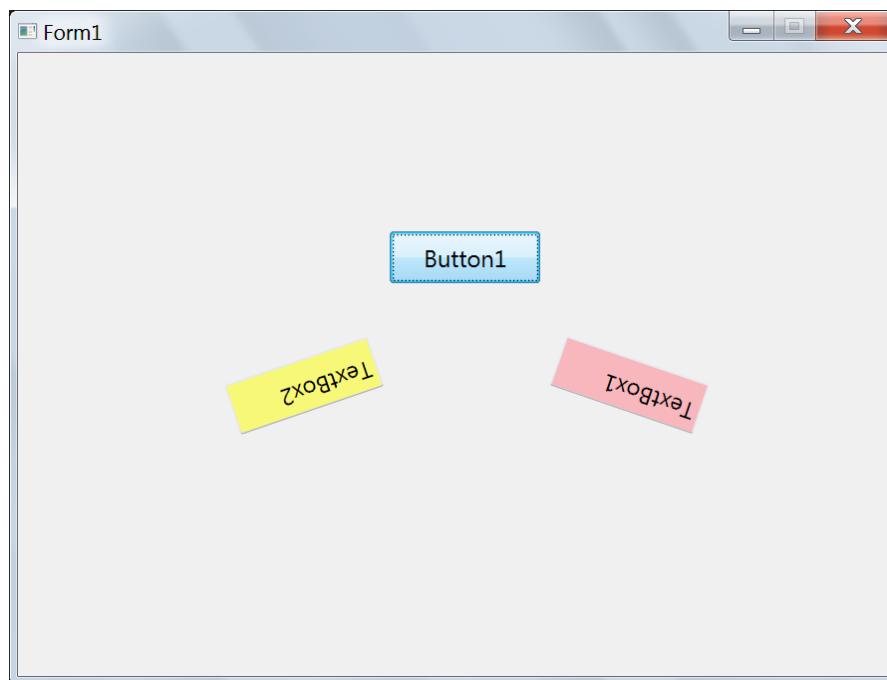
The new color to animate the control's back Color to.

- **duration:**

The time for the animation, in milliseconds.

Example:

You can see this method in action in the Animation2 project in the samples folder, where we apply many animations on the two text boxes to rotate and move them while their back colors are changing and their transparency is increasing.



Control.AnimatePos(x, y, duration)

Animates the control to a new position.

For more details about how animation works and how you can control it, see the [Animating Controls](#) section.

Parameters:

- **x:**

The x co-ordinate of the new position. The Control.Left property will reach this value after the animation ends.

- **y:**

The y co-ordinate of the new position. The Control.Top property will reach this value after the animation ends.

- **duration:**

The time for the animation, in milliseconds.

Example:

You can see this method in action in the Animation1 project in the samples folder, where we animate the angle and position of the two text boxes to rotate and move them.

Control.AnimateSize(width, height, duration)

Animates the control to a new size.

For more details about how animation works and how you can control it, see the [Animating Controls](#) section.

Parameters:

- **width:**

The new width. The Control.Width property will reach this value after the animation ends.

- **height:**

The new height. The Control.Height property will reach this value after the animation ends.

- **duration:**

The time for the animation, in milliseconds.

Example:

You can see this method in action in the Animation3 project, to animate the size of the ellipse shape.

Control.AnimateTransparency(transparency, duration)

Animates the control's back Color to a new transparency.

For more details about how animation works and how you can control it, see the [Animating Controls](#) section.

Parameters:

- **transparency:**

The new transparency to animate the transparency percentage of the back color to. Use a value between 0 and 100.

- **duration:**

The time for the animation, in milliseconds.

Example:

You can see this method in action in the Animation2 project in the samples folder, where we apply many animations on the two text boxes to rotate and move them while their back colors are changing and their transparency is increasing.

Control.BackColor As Color

Gets or sets the background color of the control.

Note that the auto-completion list will popup just after you type = to offer the names of the available colors, and you can type a few characters of the color name to filter the list, or use the mouse scroll or keyboard up and down arrows to move through the names.

```
Label1.BackColor = tr
```



Once you find the color you want, double-click it or press Enter to commit it. The name will be qualified by the `Colors.` prefix in the code editor, such as writing `Colors.Transparent` when you choose `Transparent`.

Note also that you can cancel the auto-completion list by pressing the Esc key. Now if you write any characters the auto-completion list will appear again but this time it will not offer color names, but the possible completion items for the current context, to allow you to choose a variable or a function name. If you want to view the font names again, delete any characters after = and press Ctrl+Space.

Example:

Add a ListBox on the form and write this code in the form global area:

```
ListBox1.AddItem({1, 2, 3})  
ListBox1.BackColor = Colors.Yellow
```

Control.Bottom As Double

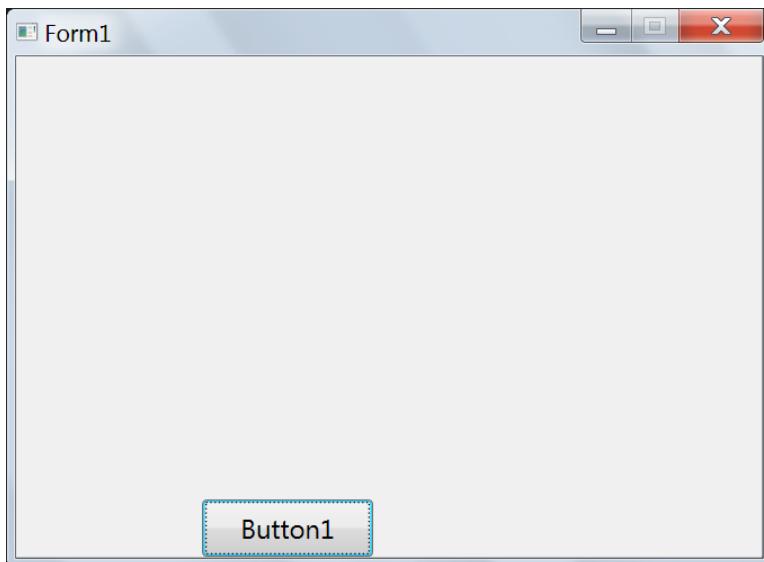
Gets or sets the y-pos of the bottom edge of control relative to top edge of its parent control (the form). If you use a negative value, the control will be out of the form.

Example:

Add a button on the form, double-click it, and write this code in its OnClick handler:

Button1.Bottom = Me.Height

When you run the program and click the button, the vertical position of the button will change so that its bottom edge touches the form's bottom edge.



Control.BringToFront()

Brings the current control to front of all other controls.

If the control is a form, it will be activated.

Example:

Add 2 textboxes and 2 labels on the form, and design it like this:



Switch to the code editor and write this code:

```
TextBox1.OnMouseDown = OnMouseDown
TextBox2.OnMouseDown = OnMouseDown
Label1.OnMouseDown = OnMouseDown
Label2.OnMouseDown = OnMouseDown

Sub OnMouseDown()
    control1 = Event.SenderControl
    If Keyboard.CtrlPressed Then
        control1.SendToBack()
    Else
        control1.BringToFront()
    EndIf
EndSub
```

Note that we used the OnMouseDown subroutine to handle the OnMouseDown event for the 4 controls, instead of repeating the same code 4 times in 4 different handlers with only the name of the control changed.

In the OnMouseDown subroutine, we used the Event.SenderControl property to get the control that fired the event, so we can bring it to front or send it to the back according to the state of the Ctrl key.

Now press F5 to run the project, and try this:

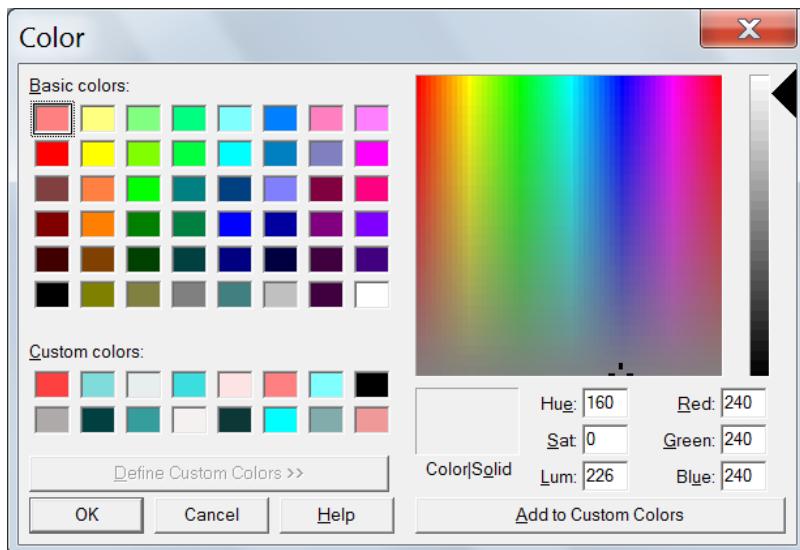
- Click any control to bring it to the front.
 - Press the Ctrl key and click any control to send it to the back of all other controls.
-

Control.CaptureMouse()

Makes the current control owns the mouse and its events, until you call the [ReleaseMouse](#) function.

Control.ChooseBackColor() As Color

Shows the color dialog to allow user to change the back color of the current control.



Returns:

The color that the user choose, or "" if he cancelled the operation.

In most cases, you will ignore this return value, unless you want to change the colors of many controls.

Example:

Add a TextBox and a Button on the form. Double-click the button and write this code in its OnClick event handler:

TextBox1.ChooseBackColor()

Press F5 to run the program, and click the button. This will show the colors dialog. Choose a color and click OK. The color you choose will be applied to the textbox back color.

Click the button to show the color dialog again and note that the textbox back color is initially selected. Choose a different color, but this time click the cancel button. The textbox back color will not change this time.

Control.ChooseFont() As Array

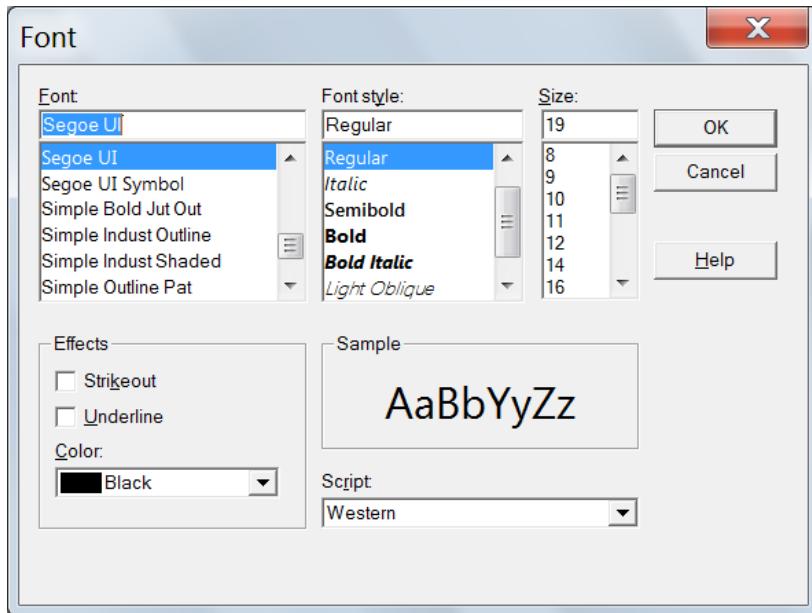
Shows the font dialog to allow user to change the font properties including the fore color of the current control.

Returns:

If user closes the font dialog by clicking it OK button, this method will return an array containing the font properties under the keys Name, Size, Bold, Italic, Underlined and Color.

In most cases, you will ignore this return value, unless you decided to change the font properties of many controls, hence you can set this return value to the Font property of each of these controls, or you can read individual keys from the returned arrays to set the values of individual properties of some controls if you wish.

If the user cancels the dialog, this method will return an empty string "".



Example:

Add a TextBox and a Button on the form. Double-click the button and write this code in its OnClick event handler:

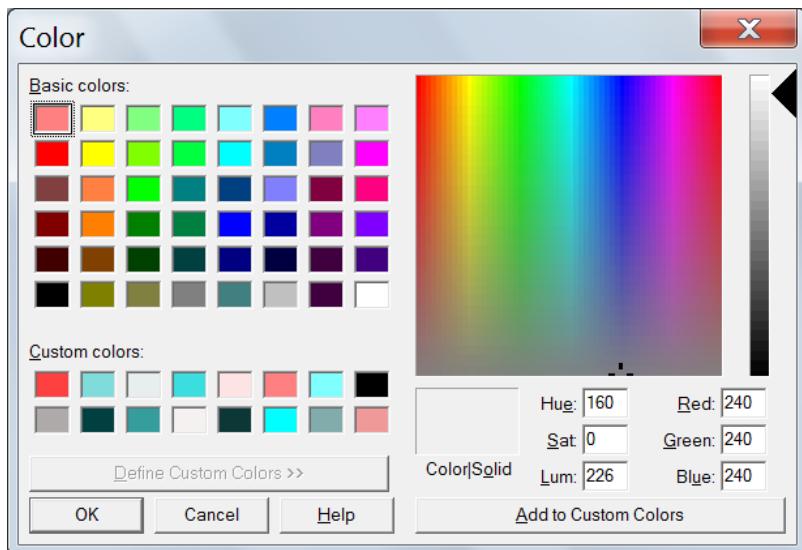
TextBox1.ChooseFont()

Press F5 to run the program, and click the button. This will show the fonts dialog. Choose font name and styles then click OK. The font properties you choose will be applied to the textbox FontName, FontBold, FontItalic, Underlined, and ForeColor properties.

Click the button to show the fonts dialog again and note that the textbox font properties values are initially selected. Choose different values, but this time click the cancel button. The textbox font will not change this time.

Control.ChooseForeColor() As Color

Shows the color dialog to allow user to change the fore color of the current control.



Returns:

The color that the user choose, or "" if he cancelled the operation.

In most cases, you will ignore this return value, unless you want to change the colors of many controls.

Example:

Add a TextBox and a Button on the form. Double-click the button and write this code in its OnClick event handler:

TextBox1.ChooseForeColor()

Press F5 to run the program, and click the button. This will show the colors dialog. Choose a color and click OK. The color you choose will be applied to the textbox fore color.

Click the button to show the color dialog again and note that the textbox fore color is initially selected. Choose a different

color, but this time click the cancel button. The textbox fore color will not change.

Control.Enabled As Boolean

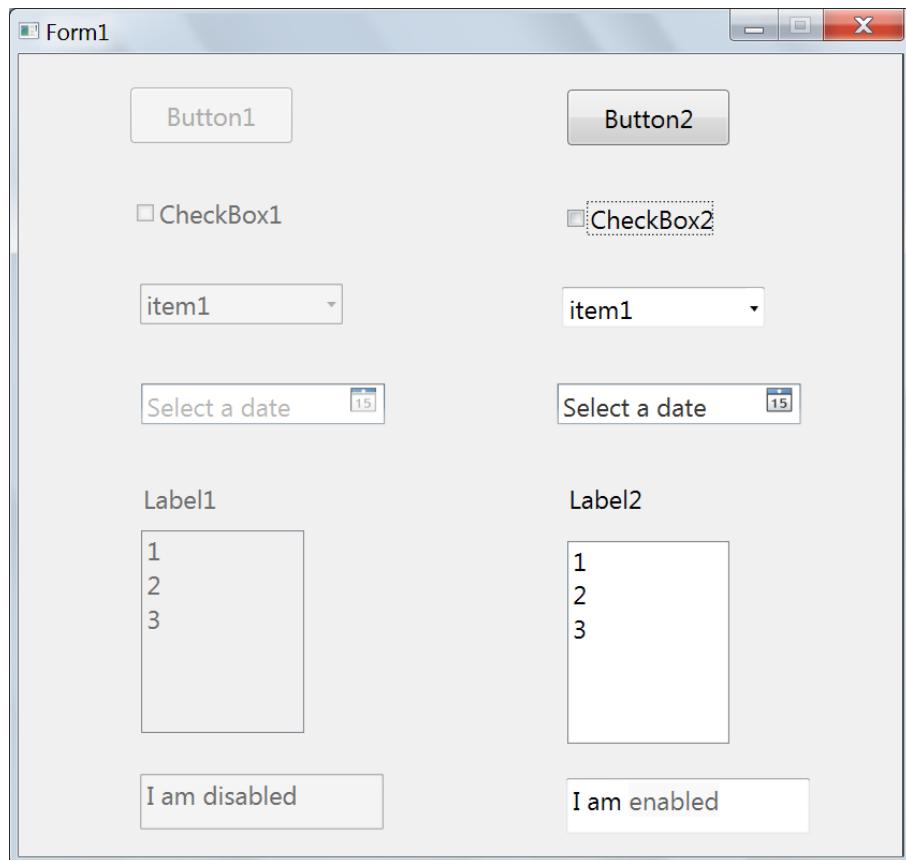
Gets or sets whether the control is enabled or disabled.

When it is True, user can interact with the control.

When it is False, the control is dimmed to indicate it is disabled, and user can't interact with it.

Example:

This picture shows you how controls look when they are disabled (at the left) and when they are enabled (at the right):



To try this, add a couple each control you see in the picture on the form, and write this code in the form global area:

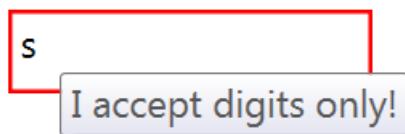
```
ComboBox1.Text = "item1"
ComboBox2.Text = "item1"
ListBox1.AddItem({1, 2, 3})
ListBox2.AddItem({1, 2, 3})
TextBox1.Text = "I am disabled"
TextBox2.Text = "I am enabled"

Button1.Enabled = False
CheckBox1.Enabled = False
ComboBox1.Enabled = False
DatePicker1.Enabled = False
Label1.Enabled = False
ListBox1.Enabled = False
TextBox1.Enabled = False
```

Control.Error As String

Gets or sets the error message for this control.

When you set the error message, the control will display a red border, and the error message will be shown as a tool tip when the mouse hovers over the control.



To reset the error, just set this property to an empty string.

The Error property is meant to be used to validate input controls before seeding the user input data to your program to process it. You can validate all controls in the button click event handler, but if the form contains many controls, putting all validation code together can unnecessarily complicate the code, so it is recommended in sVB to add the validation code in each control

[OnLostFocus](#) event handler, so user can get notified that he entered invalid value as soon as he leaves the control. For more information about validating controls, see the [Form.Validate](#) method.

Example:

You can see the Error property in action in the Validation project in the samples folder. For example, it is used in the OnLostFocus event handler of the TextBox2 to give an error if the user enters anything but positive numbers:

```
Sub TextBox2_OnLostFocus()
    n = TextBox2.Text
    If Text.IsNumeric(n) = False Then
        TextBox2.Error = "I accept digits only!"
    ElseIf n < 0 Then
        TextBox2.Error = "I don't accept negative numbers!"
    Else
        TextBox2.Error = ""
    EndIf
EndSub
```

Note that you must set the Error property to "" string when the input is valid, to reset the control to its normal style after the user fixed the input data. This is why we used the Else block in the above code.

[Control.FitContentHeight\(\)](#)

Changes the height of the control to fit its content height. This may be useful when you set WordWrap = True in label, textbox and button controls.

This is a one time change, and will not make the control height

auto-sized. If you want to make the height auto-sized, set the Height property to -1.

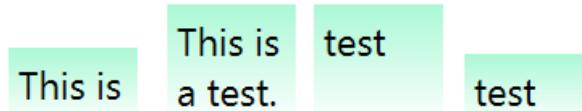
Example:

You can see this property in action in the "Fit size" project in the samples folder. The `Fit Height` button has this single line in its OnClick event handler:

Label1.FitContentHeight()

Run the program then click the `Long Text` button to make the text exceed the label width. You will not see the full text, because the label will not change its size to fit the text. Now click the `Fit Height` button, to make the label change its height to fit the text. But this is a one time change, so, if you click the `Short Text` button, the label size will not be changed although the text has shrunk. You can shrink the label height to fit this short text by clicking the `Fit Height` button again.

The four results are shown below:



Control.FitContentSize()

Changes the width and height of the control to fit its content size.

This is a one time change, and will not make the control width and height auto-sized. If you want to make them auto-sized, set both Width and Height properties to -1.

Example:

You can see this property in action in the " Fit size" project in the samples folder. The `Fit Size` button has this single line in its OnClick event handler:

Label1.FitContentSize()

Run the program then click these buttons and see what happens:

`Short Text`, `Fit Size`, `Long Text`, `Fit Height`, `Fit Width`, `Short Text`. Now the label looks like this:



Finally, click the `Fit Size` Button, to make the label look like this:



Control.FitContentSize()

Changes the width of the control to fit its content width. This is a one time change, and will not make the control width auto-sized. If you want to make the width auto-sized, set the Width property to -1.

Example:

You can see this property in action in the " Fit size" project in the samples folder. The `Fit Width` button has this single line in its OnClick event handler:

Label1.FitContentWidth()

Run the program then click these buttons and see what happens: `Long Text`, `Fit Height`, `Fit Width`.

Now the label looks like this:

This is a test.

Control.Focus()

Moves the focus to the control, so it becomes the active control that receives the keyboard keys.

Example:

You can see this method in action in the `Simple Calculator` project in the samples folder. It is used in the OnClick handler of the divide button to set the focus of txtNum2 when the user enters 0 in it, because division by zero is not possible:

```
Sub BtnDivide_OnClick
    If txtNum2.Text = 0 Then
        result = "Can't divide by zero."
        txtNum2.SelectAll()
        txtNum2.Focus()
    Else
        result = "Result = " + (
            txtNum1.Text / txtNum2.Text)
    EndIf
    lblResult.Text = result
EndSub
```

Control.Font As Array

Gets or sets an array containing the font properties under the keys: Name, Size, Bold, Italic, Underlined and Color, so you can use them as array indexers or dynamic properties. You don't have to set all these keys, as the missing keys will keep the corresponding font properties unchanged. But it is unlikely that you set these keys manually, as the font property is meant to be used in two cases:

1. Copying font properties from one control to another, by just using this single line of code:

Label1.Font = TextBox1.Font

2. Setting this property to the selected font returned by the [Desktop.ShowFontDialog method](#).

Otherwise, you should use the individual font properties, that are listed below.

Control.FontBold As Boolean

Gets or sets whether or not the font used to display the text of the control, is bold (heavy black).

Example:

Add a TextBox on the form, and write this line of code in the form global area:

TextBox1.FontBold = True

Run the program and write some characters in the TextBox. They will appear in bold font.

Hello

Control.FontItalic As Boolean

Gets or sets whether or not the font used to display the text of the control, is italic (oblique).

Example:

Add a TextBox on the form, and write this line of code in the form global area:

```
TextBox1.FontBoldItalic = True
```

Run the program and write some characters in the TextBox. They will appear in italic-style font.



Control.FontName As String

Gets or sets the font name used to display the text of the control. The auto-completion list will popup just after you type = to offer the names of the available system fonts, and you can type a few characters of the font name to filter the list, or use the mouse scroll or keyboard up and down arrows to move through the names.

```
TextBox1.FontName = o
```



Once you find the font you want, double-click it or press Enter to commit it. The name will be quoted in the code editor:

```
TextBox1.FontName = "Old Antic Bold"
```

Note also that you can cancel the auto-completion list by pressing the Esc key. Now if you write any characters the auto-completion list will appear again but this time it will not offer font names, but the possible completion items for the current context, to allow you to choose a variable or a function name. If you want to view the font names again, delete any characters after = and press Ctrl+Space. Or you can also type " or "" and once you write a character after the opening quote, the font names will appear in the completion list, and when you commit a name, the closing quote will be added if it is missing.



Control.FontSize As Double

Gets or sets the font size used to display the text of the control. This size is measured in points, like what you are used to in windows applications such as MS Word. You can use integer values between 1 and 100.

Note that font size depends also on the font type. For example, both next words have 36 points font size, but they have different font types ("Times New Roman" and "Segoe UI"), so they don't look equal:

Sample

Sample

Control.ForeColor As Color

The foreground color used to draw the text of the control.
Use values from the Color object, such as Color.Red.

Note that the auto-completion list will popup just after you type = to offer the names of the available colors, and you can type a few characters of the color name to filter the list, or use the mouse scroll or keyboard up and down arrows to move through the names.

```
Button1.ForeColor = blu
```



Once you find the color you want, double-click it or press Enter to commit it. The name will be qualified by the `Colors.` prefix in the code editor, such as writing `Colors.Blue` when you choose `Blue`.

Note also that you can cancel the auto-completion list by pressing the Esc key. Now if you write any characters the auto-completion list will appear again but this time it will not offer color names, but the possible completion items for the current context, to allow you to choose a variable or a function name. If you want to view the font names again, delete any characters after = and press Ctrl+Space.

Example:

Add a Button on the form and write this code in the form global area:

```
Button1.ForeColor = Colors.Blue
```

When you run the program, the button text will have a blue color.

Control.Height As Double

Gets or sets the height of the control.

If you want to use an auto-height to fit the control's content height, set this property to -1.

You can also limit the auto height not to exceed a certain height by setting the [MaxHeight](#) property.

Control.Left As Double

Gets or sets the x-pos of the left edge of the control relative to the left edge of its parent control (the form). If you use a negative value, the left edge of the control will be out of the form, and if the left gets more negative, the whole control may not appear on the form.

Example:

Add two buttons on the form and write this code:

```
Button2.Left = -Button1.Width  
  
Sub Button1_OnClick()  
    Button2.Left = Button2.Left + Button1.Width / 2  
EndSub
```

When you run the program, button2 will not appear on the form, because its left = - its width. When you click button1, the button2 left will increase by half its width, so, its half will appear on the form. When you click button1 again, button2 will fully appear on the form. You can continue clicking

button1 to increase the horizontal position of button2, until it goes out the right edge of the form and disappears.

Control.MaxHeight As Double

Get or sets the max height of the control. It is useful specially when you set the control's height to auto (Height = -1), to force the auto height not to exceed a max length.

Note that setting this property to 0 or a negative value will reset it (no max height).

Example:

Add a TextBox on the form and write this code in the form global area:

```
TextBox1.MaxHeight = 100  
TextBox1.Height = 500  
Me.ShowMessage(TextBox1.Height, "Height")  
TextBox1.MaxHeight = 0  
Me.ShowMessage(TextBox1.Height, "Height")
```

When you run the program, the message box will tell you that the textbox height is 100, which is the max height allowed. This means that you can't force the control to have a height greater than its max height. After you close the message box, the max height will be reset, and this will allow the textbox height to exceed 100, so, the second message box will show 500! This means that value you set to the Height property is preserved, but reading the Height property will always return the actual height of the control.

Control.MaxWidth As Double

The max width of the control. It is useful specially when you set the control's width to auto (Width = -1), to force the auto width not to exceed a max length.

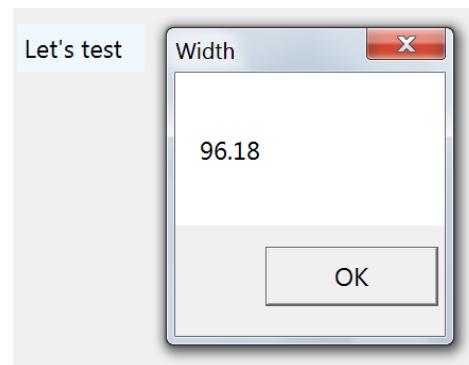
Note that setting this property to 0 or a negative value will reset it (no max width).

Example:

Add a Label on the form and write this code in the form global area:

```
Label1.MaxWidth = 100  
Label1.BackColor = Colors.AliceBlue  
Label1.Width = -1  
Label1.Text = "Let's test the effect of MaxWidth"  
Me.ShowMessage(Label1.Width, "Width")
```

When you run the program, the message box will tell you that the label width is 100 or a bit less:

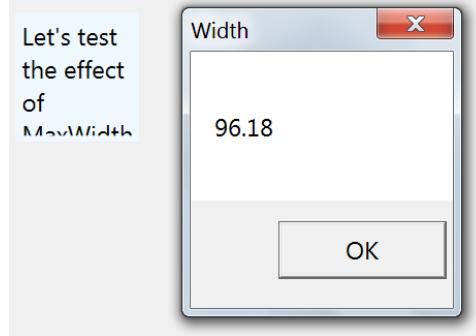


But why can the width be < 100? Can't the label width be exactly 100?

You can get the answer if you add this line of code at the start of the above code:

```
Label1.Height = 100
```

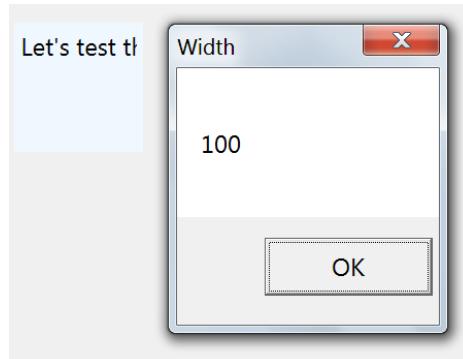
When you run the program, you will be able to see that the rest of test is wrapped over next lines. Each wrapped line starts with a whole word, so, each fragmented line ends with a space, and the label width is adjusted to fit the longest one of them.



So, if you want to have the full max width, turn off the word wrapping by adding this line of code at the start of the code:

Label1.WordWrap = False

When you run the program you will get what you expect (with the price of trimming the extra text out of the label area):



Control.MouseX As Double

Gets the mouse x-pos relative to the control. When mouse is over the control, this value lies between 0 and the control's width. A negative value means that the mouse pointer is outside

the left edge of the control.

Note that you can use the [Mouse.X](#) property to get mouse x-pos relative to the screen.

Example:

You can see this property in action in the `Mouse pos` project in the samples folder. It is used in the OnMouseMove event handler of each of the 3 controls and the form to show the mouse position relative to each of them:

```
TextBox2.Text = Text.Format(  
    "Pos relative to TextBox2: ([1], [2])",  
    {TextBox2.MouseX, TextBox2.MouseY}  
)  
  
Form1.Text = Text.Format(  
    "Pos relative to the form: ([1], [2])",  
    {Me.MouseX, Me.MouseY}  
)
```

Control.MouseY As Double

Gets the mouse y-pos relative to the control. When mouse is over the control, this value lies between 0 and the control's height. A negative value means that the mouse pointer is above the top edge of the control.

Note that you can use the [Mouse.Y](#) property to get mouse x-pos relative to the screen.

Example:

You can see this property in action in the `Mouse pos` project in the samples folder. It is used in the OnMouseMove event handler of each of the 3 controls and the form to show the

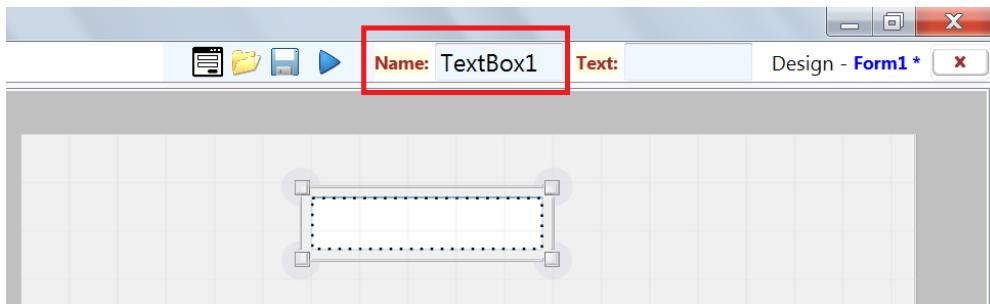
mouse position relative to each of them:

```
TextBox2.Text = Text.Format(  
    "Pos relative to TextBox2: ([1], [2])",  
    {TextBox2.MouseX, TextBox2.MouseY}  
)  
  
Form1.Text = Text.Format(  
    "Pos relative to the form: ([1], [2])",  
    {Me.MouseX, Me.MouseY}  
)
```

Control.Name As String

Gets the name of the control, which is a unique identifier that we use in code to refer to the control, like TextBox1 and BtnOk.

You can't change the control name at runtime. Use the designer to select the control then use the upper 'Name' textbox to change the name:



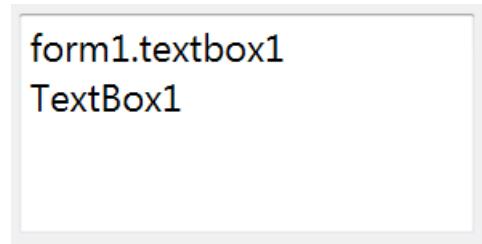
Note that the control variable is actually a string, that consists of the form name and the control name separated by a dot like `form1.textBox1`, while the name property of the control holds only its name without the name of the form.

Example:

Add a TextBox on the form and write this code in the form global area:

```
TextBox1.AppendLine(TextBox1)
TextBox1.AppendLine(TextBox1.Name)
```

When you run the program, the textbox will appear as in this picture:



⚡ Control.OnClick

This event is fired when user presses the left mouse-button on the control then releases it.

This is the default event for the Button and Label controls, so you can just double-click them on the form designer to get the handler for this event added for you, such as:

```
Sub Button1_OnClick()
```

```
EndSub
```

For more info, read about [handling control events](#).

⚡ Control.OnDoubleClick

This event is fired when user double-clicks the control.

For more info, read about [handling control events](#).

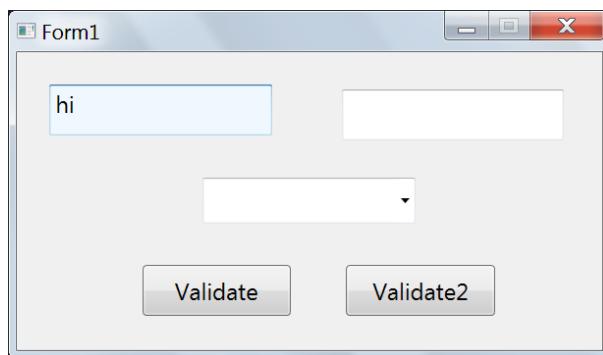
⚡ Control.OnGotFocus

This event is fired when the control gets the focus.

For more info, read about [handling control events](#).

Example:

You can see this event in action in the `Validation` project in the samples folder. It is used to give the focused control a different background. Both TextBox2 and ComboBox1 will use the OnGotFocus event handler of TextBox1, so that the TextBox1_OnGotFocus sub will handle the 3 controls:



```
TextBox2.OnGotFocus = TextBox1_OnGotFocus  
ComboBox1.OnGotFocus = TextBox1_OnGotFocus
```

```
'-----  
Sub TextBox1_OnGotFocus()  
    senderControl = Event.SenderControl  
    senderControl.BackColor = Colors.AliceBlue  
EndSub
```

Note that we need to give back the white back color to the control when we leave it, so, we need to use the OnLostFocus event for each control to do that, but in this time we will not use one handler for the 3 controls, because the Validation project uses the OnLostFocus for each control to validate its content, and the validation logic is different for each control, so if you take a look at that project, you can see that the OnLostFocus handler for each control checks the validation rules then resets its back color. For example:

```
Sub TextBox1_OnLostFocus()  
    If TextBox1.Text = "" Then
```

```
    TextBox1.Error = "Don't keep me empty!"  
Else  
    TextBox1.Error = ""  
EndIf  
    TextBox1.BackColor = Colors.White  
EndSub
```

Control.OnKeyDown

This event is fired when the user presses a keyboard key.

Use the [Keyboard](#) properties to get info about the key that fired the event and the state of the Ctrl, Shift, Alt and other special keys.

For more info, read about [handling control events](#).

Example:

You can see this event in action in the `Numeric TextBox` project in the samples folder. It is used to prevent the user from typing any thing but digits in the textbox. To do so, we use the Keyboard.LastKey property to get the key that raised the event, and make sure it is one of the digit keys, which are represented by the Keys properties Key.D0 to Key.D9. If the pressed key is out of this range, this we must prevent it by setting the Event.Handled to True to tell the textbox not to do any thing more which means it will not write the pressed key. This is the code of the OnKeyDown handler:

```
Sub TextBox1_OnKeyDown()  
    lastKey = Keyboard.LastKey  
    If lastKey < Keys.D0 Or lastKey > Keys.D9 Then  
        Event.Handled = True  
    EndIf  
EndSub
```

Control.OnKeyUp

This event is fired when the user releases a keyboard key that was pressed.

Use the [Keyboard](#) properties to get info about the key that fired the event and the state of the Ctrl, Shift, Alt and other special keys. For more info, read about [handling control events](#).

Control.OnLostFocus

This event is fired when the control loses the focus.

Use the event to validate the user input to make sure that he entered a valid data. For more information, see how to use the [Error Property](#) in this event handler to validate the control.

For more info, read about [handling control events](#).

Control.OnMouseEnter

This event is fired when the mouse pointer enters the control area.

Use the [Mouse](#) properties to get info about the mouse position and its buttons state.

For more info, read about [handling control events](#).

Control.OnMouseLeave

This event is fired when the mouse pointer leaves the control area.

Use the [Mouse](#) properties to get info about the mouse position and its buttons state.

For more info, read about [handling control events](#).

Control.OnMouseDown

This event is fired when user presses the left mouse-button down. Use the [Mouse](#) properties to get info about the mouse position and its buttons state.

For more info, read about [handling control events](#).

Control.OnMouseLeftUp

This event is fired when user releases the left mouse-button after it was pressed.

Use the [Mouse](#) properties to get info about the mouse position and its buttons state.

For more info, read about [handling control events](#).

Control.OnMouseMove

This event is fired repeatedly while the mouse pointer moves over the control. Use the [Mouse](#) properties to get info about the mouse position and its buttons state.

For more info, read about [handling control events](#).

Example:

You can see this event in action in the `FormShape` project in the samples folder. It is used to allow the user to drag the form by left-clicking at any point and move the mouse to a new position then release the mouse button. To do so, we must save the last point to displace the form by the deference between the mouse position and that point. This is the code:

```
LastX = 0  
LastY = 0  
  
Sub Form1_OnMouseMove()  
If Event.IsLeftButtonDown Then
```

```
    Me.Left = Me.Left + Mouse.X - LastX  
    Me.Top = Me.Top + Mouse.Y - LastY  
EndIf  
LastX = Mouse.X  
LastY = Mouse.Y  
EndSub
```

Control.OnMouseRightDown

This event is fired when user presses the right mouse-button down.

Use the [Mouse](#) properties to get info about the mouse position and its buttons state.

For more info, read about [handling control events](#).

Control.OnMouseRightUp

This event is fired when user releases the right mouse-button after it was pressed.

Use the [Mouse](#) properties to get info about the mouse position and its buttons state.

For more info, read about [handling control events](#).

Control.OnMouseWheel

This event is fired repeatedly while the user moves the mouse wheel.

Use the [Event.LastMouseWheelDirection](#) property to know if the wheel is moving up or down.

For more info, read about [handling control events](#).

Example:

Add a label on the form, change its back color and fore color to look like what you see in the next picture, and set its text to 0.



Switch to the code editor and add this code to increase or decrease the value of the label when the user moves the mouse wheel up or down respectively:

```
N = 0  
Sub Label1_OnMouseWheel()  
    N = N + Event.LastMouseWheelDirection  
    Label1.Text = N  
EndSub
```

Control.Padding As Double

Gets or sets the padding of the control, which is the internal space between the control edges and its content.

Example:

Add a label on the form, change its back color and border to look like what you see in the next picture.



By default, there is a 5 point padding between the label borders and its text. You can increase this padding to 25 by using this code:

```
Label1.Padding = 25  
Label1.FitContentSize()
```

Note that changing the padding may cause a part of the text to disappear if the height or the width or the label are small, so, we called the FitContentSize method to adjust the label size to fit the text with the new padding.



Note that the padding may bother you when you add an image on the label, so you can remove it by setting this property to 0:

```
Label1.Padding = 0
```

Control.ReleaseMouse()

Releases the mouse if the current control captured it by calling the [CaptureMouse](#) method.

Control.RemoveEventHandler()

Removes the handler of the given event of the current control. It affects only event handlers that were added by your program, and will not remove any handlers added by external sVB libraries. If you wrote a library that adds events handlers, you may provide a function that removes these handlers, so the user of your lib can call it. See the Example section.

Note that sVB provides a syntax to do the same job, by setting the event to **Nothing**, which is translated to a call to the RemoveEventHandler method when the program is compiled. So, this code:

```
TextBox1.OnTextChanged = Nothing
```

is equivalent (and will be compiled to) this code:

```
TextBox1.RemoveEventHandler("OnTextChanged")
```

It doesn't matter how you added the handler for the OnTextChanged event, as it will be removed whether you added it by setting the handler using code or by using a sub that follows the naming convention (TextBox1_OnTextChanged).

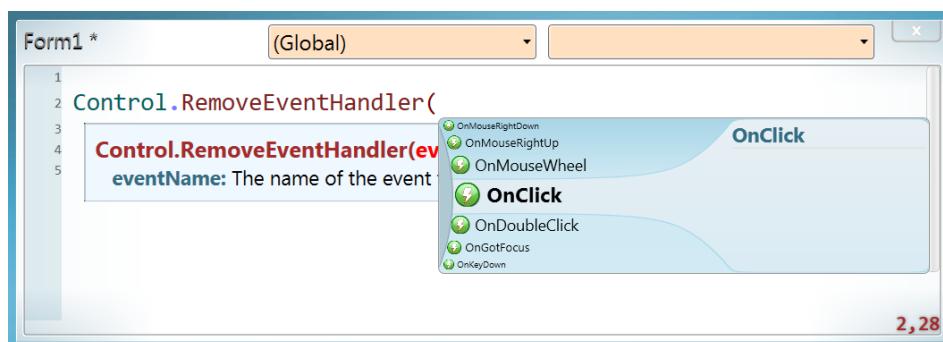
Note that **Nothing** works also with old Small Basic events like Timer.Tick, which can't be used with the RemoveEventHandler method:

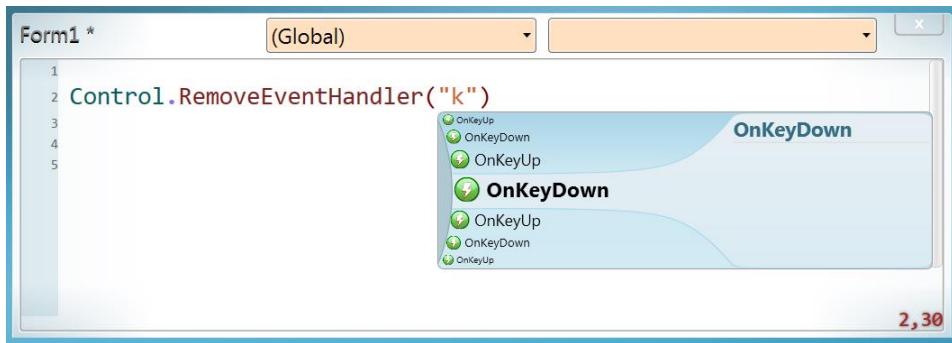
```
Timer.Tick = Nothing
```

Parameter:

- **eventName:**

The name of the event to remove its handler. You provide this name as string, but you don't need to worry about misspelling it, because the code editor will offer you a completion list with available events names.





Example:

The Geometrics lib (which is written with sVB, and its source code exists in the samples folder) has the AllowDrag method, which receives a control and registers two handlers for its OnMouseMove and OnMouseLeftDown events, to allow the user to drag it on the form.

Geometrics.AllowDrag(TextBox1)

You can't directly call the RemoveEventHandler method from your project to remove these handlers, but the Geometrics lib provides you with the PreventDrag method, which removes the handlers of OnMouseLeftDown events for the given control to disable the dragging capability.

Geometrics.PreventDrag(TextBox1)

This is the code used to do that, and you can find it in the Global.sb file in the Geometrics application in the sVB samples folder:

```
' Allows the user to drag the control by mouse
Sub AllowDrag(targetControl)
    targetControl.OnMouseMove = _OnMouseMove
    targetControl.OnMouseLeftDown = _OnMouseLeftDown
EndSub

' Prevents the user from dragging the control by mouse
Sub PreventDrag(targetControl)
    targetControl.RemoveEventHandler("OnMouseMove")
    targetControl.RemoveEventHandler("OnMouseLeftDown")
EndSub
```

Control.Right As Double

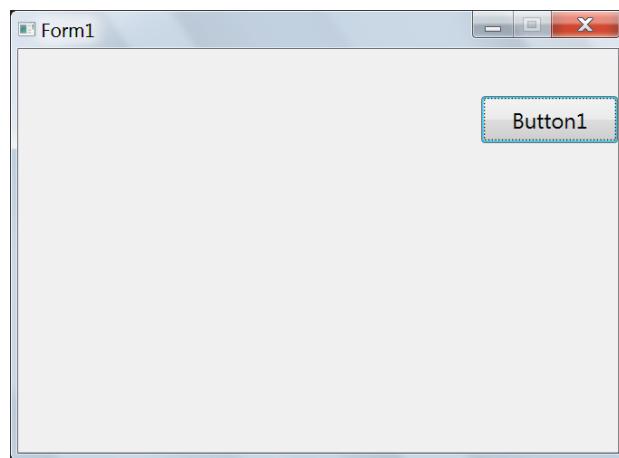
Gets or sets the x-pos of the right edge of control relative to the left edge of its parent control (the form). If you use a negative value, the control will be out of the form.

Example:

Add a button on the form, double-click it, and write this code in its OnClick handler:

```
Button1.Right = Me.Width
```

When you run the program and click the button, the horizontal position of the button will change so that its right edge touches the form's right edge.



Control.RightToLeft As Boolean

Gets or sets whether the control content is displayed from right to left. This is useful when you display right to left languages like Arabic.

Control.Rotate(angle)

Rotates the control by the specified angle.

Parameter:

- **angle:**

The angle in degrees to rotate the shape by. It will be added to the current value of the [Control.Angle](#) property.

Example:

Add a ProgressBar and a Label on the form and write this code:

```
ProgressBar1.Rotate(120)  
ProgressBar1.Rotate(-30)  
Label1.Text = ProgressBar1.Angle
```

When you run the program, the label will show 90, and you will have a vertical ProgressBar.



90

Control.SendToBack()

Sends the current control to the back of all other controls.

This method has no effect on forms.

See the example given for the [BringToFront](#) method.

Control.SetResourceDictionary(fileName)

Sets the resource dictionary of the control by loading it from the given file. The resource dictionary contains styles for the control and its child controls. This allows you to design advanced styles with other tools like VS.NET, save them to a file as a resource dictionary, then use this method to load it.

This is an advanced topic, that needs knowledge about XAML and WPF, which probably you don't have, but it allows you to make use of styles and themes defined for WPF, UWP or WinUI3 (which you can easily find by a quick search on google, or by

asking someone who has knowledge about these products). This will help you to make a beautiful design, by changing how controls look and even work!

Notes:

- This method is most useful when called from a form, so that the styles affect all target control types if exists on the form. In this case, styles must have no keys (names), to affect all child controls of the targeted types.
- In the styles file, add styles to target controls, but avoid to target the window (form), because controls actually are placed on a canvas on the form, and many properties of the window are set directly to the canvas not to the form. So, it is not easy to write a style to change the background on the form (not the canvas).
- Styles may fail to affect some properties like the Button.Text property, because there is no such property in wpf, and sVB just adds a TextBlock to the Button.Content. So, use the styles to do what you can't do with sVB code, and do the rest with code if styles failed to do it. Otherwise, you may use the style to change the control template so that you can force the control to look exactly like you want.
- If the styles file uses any images, make sure to copy these images to project directory, and fix the image files paths used in the styles file to be relative and to start with `\\`, like "\\sample.jpg", otherwise the image will not be recognized in sVB.

Parameter:

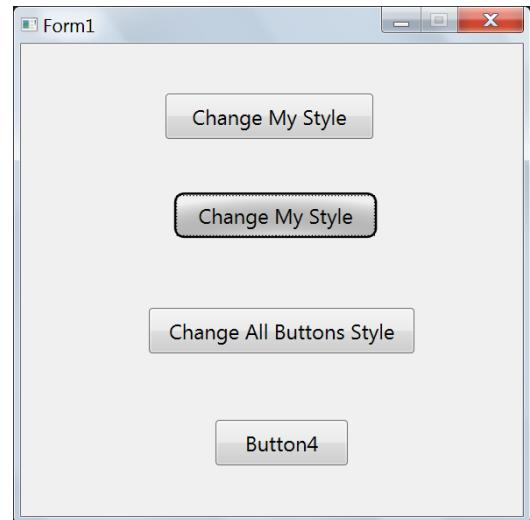
- **fileName:**

The file that contains the resource dictionary. It is actually a xaml file, but it is better to change its extension to ` `.style` not

to be confused with form design files that have the ` `.xaml` extension. Style files should be saved in the project directory, and you should use the file name without the full path (Like "rs.style"). Don't worry about copying style files to the project bin folder, as sVB will copy all .xaml and .style files to it when you run the program.

Examples:

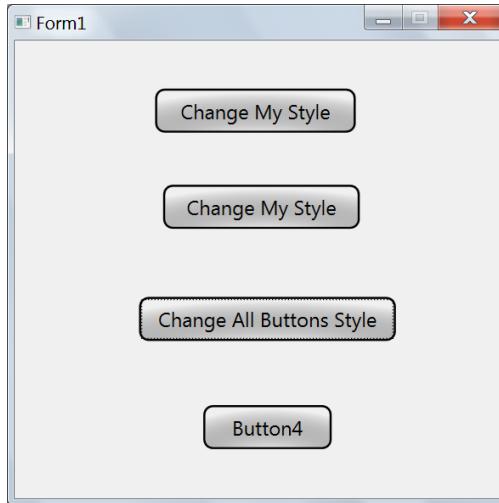
You can see this method in action in the `Custom Styles` project in the samples folder. This project contains a style file named `RoundCorner.style`, that changes the button style to have rounded edges and changes its back color. You can apply this style on a single button by loading this file to its resource dictionary. This is exactly what the second `Change My Style` button does when you click it:



Button3.SetResourceDictionary("RoundCorner.style")

You can also apply this style to all buttons on the form at once, by loading this file to the form resource dictionary. This is exactly what the `Change All Buttons Style` button does when you click it:

Me.SetResourceDictionary("RoundCorner.style")



Warning:

If you use a style that changes the control template, this may effect some of its properties, so they may stop working. As an example, in the above project, the style we applied on the buttons changes the button template to use a gradient brush to draw its background, which totally ignores the value of its `BackgroundColor` property (that we use in sVB by the name `BackColor`). So, after applying this style, applying this code will has no effect on buuton1:

```
Button1.BackColor = Colors.Red
```

So, you should create the styles carefully, not to disable control properties, or if you do, add all the desired effects in the style, so that you don't need to change the appearance of the control by code.

Control.SetStyle(fileName, styleKey)

Sets the style of the control by loading it from the given file. This allows you to design advanced styles with other tools like VS.NET, save them to a file as a resource dictionary, then use this method to load it.

Note that this method is similar to the [SetResourceDictionary](#) method, but the SetStyle method has a second parameter that receives the key (name) of the style, so it is useful when you want to apply a style on only one control. But if the style has no name, you can call SetResourceDictionary from the control to apply the style on it, but this can also apply another styles on the control if the resource dictionary has many styles targeting the same control type.

For more info, see the [SetResourceDictionary](#) method notes and examples.

Parameters:

- **fileName:**

The style file that contains the resource dictionary.

- **styleKey:**

The key of style. The resource dictionary can contain many styles targeting the same control. This method will search for the style that have the given key, and apply it if found.

Examples:

You can see this method in action in the `Custom Styles` project in the samples folder. This project contains a style file named `RoundCorner2.style`, that have a style with the key `RoundedButton`. This style changes the button to have rounded edges and changes its back color. You can apply this style on a single button by loading this file using the SetStyle method. This is exactly what the first `Change My Style` button does when you click it:

```
Button1.setStyle("RoundCorner2.style",
    "RoundedButton")
```

```
Button1.ForeColor = Colors.Yellow
```



Note that if click the `Change All Buttons Style` button first, Button1 will have the gray style, but when you click Button1 it will get the green style. This tells you that the last style you apply on the control wins and forces its values on the control. This also true when you change some properties from code after you apply the styles. The last value you set to the property will overwrite any old value.

Now lets do something interesting: What do you think of displaying a spherical button like this?



To do it, use the form designer to resize Button1 to be a 3x3 cm square, then open the `RoundCorner2.style` file in notepad, and change the value of the Border.CornerRadius property to 60:

```
<Border x:Name="border" CornerRadius="60"
    BorderBrush="Black" BorderThickness="2">
```

and save the file. Now when you run the program and click Button1, it will be turned to a spherical button.

Control.Tag

This is an extra property to store any value you want that is related to the control. You can store multiple values by putting them in an array or a dynamic object and set it to this property.

Control.ToolTip As String

Gets or sets the message to display as a tool tip when you hover over the control.

Example:

Add a button on the form, and use this code:

```
Button1.ToolTip = "Click me!"
```

Press F5 to run the program, and hover by the mouse over the button for a while (without moving the mouse). The button will show the tool tip as in the picture:



Control.Top As Double

Gets or sets the y-pos of the top edge of the control relative to the top edge of its parent control (the form). If you use a negative value, the top edge of the control will be out of the form, and if the top gets more negative, the whole control may not appear on the form.

Example:

Add two buttons on the form and write this code:

```
Button2.Top = -Button1.Height  
Sub Button1_OnClick()  
    Button2.Top = Button2.Top + Button1.Height / 2  
EndSub
```

When you run the program, button2 will not appear on the form, because its top = - its height. When you click button1, the button2 top will increase by half its height, so, its half will appear on the form. When you click button1 again, button2 will fully appear on the form. You can continue clicking button1 to increase the vertical position of button2, until it goes out the bottom edge of the form and disappears.

Control.TypeName

Gets the name of the control type, like TextBox, Label, and Button.

Example:

Add different types of controls on the form and write this code in the global area:

```
ForEach _Control In Me.Controls  
    TextWindow.WriteLine(_Control.TypeName)  
Next
```

When you run the program, the TextWindow will show the control names.

Control.Validate() As Boolean

Forces the control to raise the [OnLostFocus event](#) to apply any validation logic supplied by you in the OnLostFocus handler, then checks the [Error property](#) to see if the control has errors or not. In most cases you don't need to call this method explicitly, as it is called implicitly by the [Form.Validate](#) method which validates all controls of the form.

But you may call it in some cases, like when you need to force validate the combo box when the selected item changes. This is

exactly what happens in the Validation project in the samples folder:

```
Sub ComboBox1_OnSelection()
    ComboBox1.Validate()
EndSub
```

Returns:

True if the current control has no errors, or False otherwise.

Example:

You can validate all controls by just calling Me.Validate, but this methods stops if any control input is invalid, and doesn't validate next controls until the user fixed fixes the error and try again. You may prefer to validate all controls at once, then move the focus to the first invalid one, so the user see red borders around all invalid controls, and can hover over any one of them to see the error message in the tool tip. This will allow him to fix all errors at once. You can do this by writing a loop to call the Validate method for each control on the form. This code is used in the Validate button on the Validation project in the samples folder:

```
isValid = True
ForEach control1 In Me.Controls
    If control1.Validate() = False Then
        Sound.PlayBellRing()
        If isValid Then ' This is the first invalid control
            control1.Focus()
            isValid = False
        EndIf
    EndIf
Next
If isValid Then ' No errors found. Process the data
    Me.ShowMessage("OK", "")
EndIf
```

Control.Visible As Boolean

When it is True, the control is shown at the form.

When it is False, the control is hidden.

Example:

You can see this property in action in the "CheckBox Sample" project in the samples folder. It is used in the `ChkShowText_OnCheck` event handler to show or hide TextBox1, according to the state of the checkbox state:

```
Sub ChkShowText_OnCheck()
    state = ChkShowText.Checked
    If state Then ' Checked
        TextBox1.Visible = True
        TextBox1.Enabled = True
    ElseIf state = "" Then ' Indeterminate
        TextBox1.Visible = True
        TextBox1.Enabled = False
    Else ' Unchecked
        TextBox1.Visible = False
        TextBox1.Enabled = False
    EndIf
EndSub
```

Control.Width As Double

Gets or sets the width of the control.

If you want an auto-width to fit the control's content width, set this property to -1.

You can also limit the auto width by setting the [MaxWidth](#) property.

The Controls Type

The Controls object allows you to add controls on the [Graphics Window](#), and to move and interact with them.

In normal cases, you should avoid using this type, as it is a Small Basic legacy type, and use the sVB form designer to add controls on your forms instead.

The Controls type has these members:

Controls.AddButton(caption, left, top) As Button

Adds a new [Button control](#) to the graphics window at runtime.

Parameters:

- **caption:**

The text to display on the button.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

Returns:

The key of the button. sVB can deal with this key as an object of type [Button](#), so, you can access the Button properties, methods and events via it.

Example:

In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file:

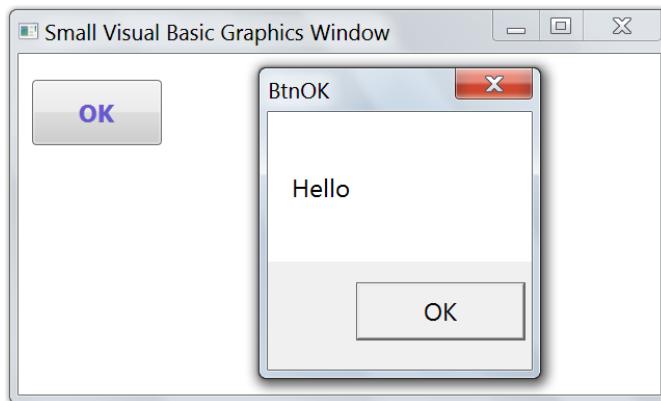
```

BtnOK = Controls.AddButton("OK", 10, 20)
BtnOK.Width = 100
BtnOK.Height = 50
BtnOK.FontSize = 14
BtnOK.OnClick = BtnOK_OnClick

Sub BtnOK_OnClick()
    Forms.ShowMessage("Hello", "BtnOK")
EndSub

```

Press F5 to run the program. The OK button will be displayed on the graphics window, and when you click it, it will show the "Hello" message box:



Controls.AddCheckBox(caption, left, top, checked) As CheckBox

Adds a new [CheckBox control](#) to the graphics window.

Parameters:

- **caption:**

The text to display on the CheckBox.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **checked:**

The value to set to the CheckBox.Checked property.

Returns:

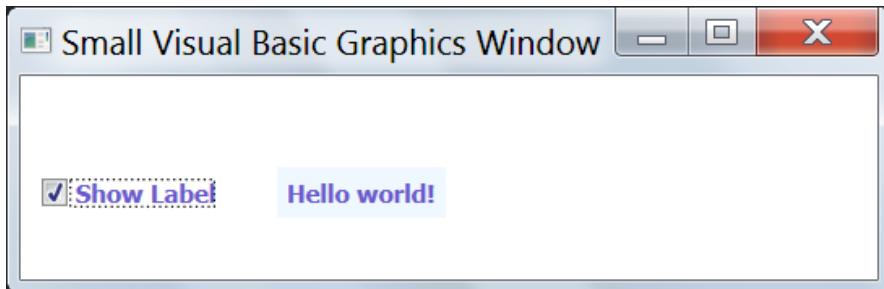
The key of the CheckBox. sVB can deal with this key as an object of type [CheckBox](#), so, you can access the CheckBox properties, methods and events via it.

Example:

In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file:

```
' Create a checkbox and handel its OnCheck event:  
CheckBox1 = Controls.AddCheckBox(  
    "Show Label", 10, 50, True)  
CheckBox1.OnCheck = CheckBox1_OnCheck  
  
' Create a label on the left of the checkbox:  
Label1 = Controls.AddLabel(  
    "Hello world!",  
    CheckBox1.Left + CheckBox1.Width + 30,  
    CheckBox1.Top)  
  
' Set the label's properties:  
Label1.BackColor = Colors.AliceBlue  
' Align the middles of label and checkbox horizntally:  
Label1.Top = Label1.Top -  
    (Label1.Height - CheckBox1.Height) / 2  
  
' The OnCheck handler hides or shows the label:  
Sub CheckBox1_OnCheck()  
    Label1.Visible = CheckBox1.Checked  
EndSub
```

Press F5 to run the program. The graphics window will display the checkbox and the label. Check and uncheck the checkbox to show and hide the label.



⚙️ **Controls.AddComboBox(left, top, width, height) As ListBox**

Adds a new [ComboBox control](#) to the graphics window.

Parameters:

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

Returns:

The key of the combo box. sVB can deal with this key as an object of type [ComboBox](#), so, you can use the ComboBox properties, methods and events via it.

Example:

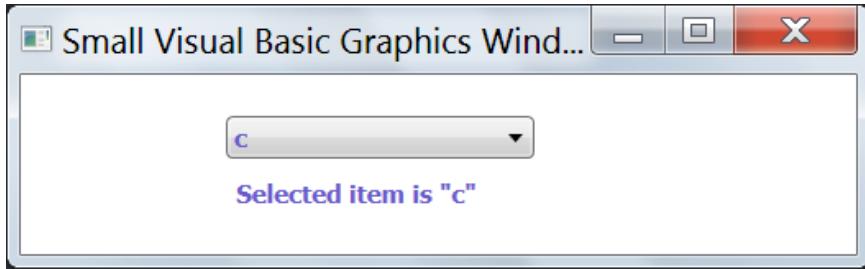
In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file:

```
ComboBox1 = Controls.AddComboBox(100, 20, 150, -1)
ComboBox1.AddItem({"a", "b", "c"})
ComboBox1.OnSelection = ComboBox1_Select
ComboBox1.SelectedIndex = 3

Sub ComboBox1_Select()
    Print(ComboBox1.SelectedItem)
EndSub

Sub Print(item)
    ' Erase any text previously written
    c = GraphicsWindow.BackgroundColor
    GraphicsWindow.BrushColor = c
    GraphicsWindow.FillRectangle(
        100, 45,
        GraphicsWindow.Width - 105,
        GraphicsWindow.Height - 40
    )
    ' Write the selected item:
    GraphicsWindow.BrushColor = Colors.Black
    GraphicsWindow.DrawText(
        105, 50,
        Text.Format(
            "Selected item is [1][2][1]",
            {Chars.Quote, item}
        )
    )
EndSub
```

Press F5 to run the program. The graphics window will display the combo box with the last item selected, and the selected item will be written under the combo box, and will be updated every time you change the selected item.



Controls.AddDatePicker(left, top, width, selectedDate) As DatePicker

Adds a new [DatePicker control](#) to the graphics window.

Parameters:

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **selectedDate:**

The date that will be selected in the control when it is displayed.

Returns:

The key of the DatePicker. sVB can deal with this key as an object of type [DatePicker](#), so, you can access the DatePicker properties, methods and events via it.

Example:

In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file:

```

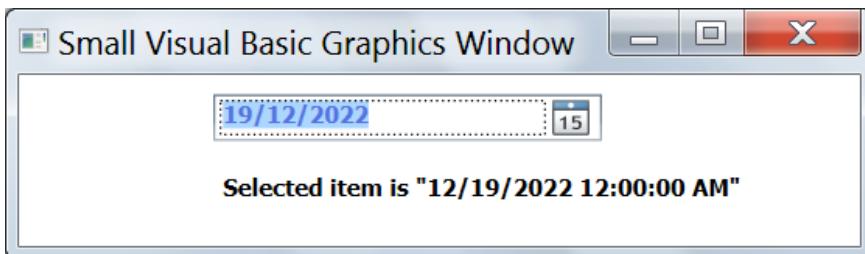
DatePicker1 = Controls.AddDatePicker(
    75, 10, 200, #12/18/2022#)
DatePicker1.OnSelection = DatePicker1_OnSelection

Sub DatePicker1_OnSelection()
    Print(DatePicker1.SelectedDate)
EndSub

```

Also add the Print subroutine used in the previous example.

Press F5 to run the program. Every time you change the selected date, it will be displayed under the date picker.



0Controls.AddLabel(caption, left, top) As Label

Adds a new Label control to the graphics window.

Parameters:

- **caption:**

the text to display on the label.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

Returns:

The key of the label. sVB can deal with this key as an object of type Label, so, you can access the Label properties, methods and events via it.

Example:

In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file:

```
Label1 = Controls.AddLabel("Hi there!", 10, 20)
Label1.OnMouseEnter = Label1_OnMouseEnter
Label1.OnMouseLeave = Label1_OnMouseLeave

Sub Label1_OnMouseEnter()
    Label1.BackColor = Colors.AliceBlue
EndSub

Sub Label1_OnMouseLeave()
    Label1.BackColor = Colors.Transparent
EndSub
```

Press F5 to run the program. The graphics window will display the "Hi there" label, and when the mouse pointer enters its area, its back color will change, where you can confirm that its width and height fits its content (because we used -1 for both). When the mouse pointer leaves the label area, its back color will be transparent again. You can repeat that as many times as you want!

Controls.AddListBox(left, top, width, height) As ListBox

Adds a new [ListBox control](#) to the graphics window.

Parameters:

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

Returns:

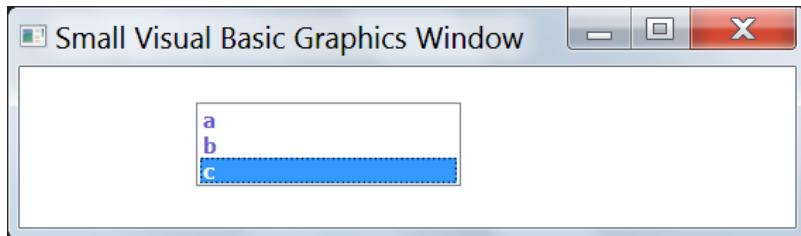
The key of the list box. sVB can deal with this key as an object of type [ListBox](#), so, you can access the ListBox properties, methods and events via it.

Example:

In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file:

```
ListBox1 = Controls.AddListBox(100, 20, 150, -1)
ListBox1.AddItem {"a", "b", "c"}
ListBox1.SelectedIndex = 3
```

Press F5 to run the program. The graphics window will display the list box with the last item selected.



① **Controls.AddProgressBar(left, top, width, height, minimum, maximum) As ProgressBar**

Adds a new [ProgressBar control](#) to the graphics window.

Parameters:

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

- **minimum:**

The progress minimum value

- **maximum:**

The progress maximum value. Use 0 if the max value is indeterminate. This will display an endlessly moving marquee.

Returns:

The key of the ProgressBar. sVB can deal with this key as an object of type [ProgressBar](#), so, you can access the ProgressBar properties, methods and events via it.

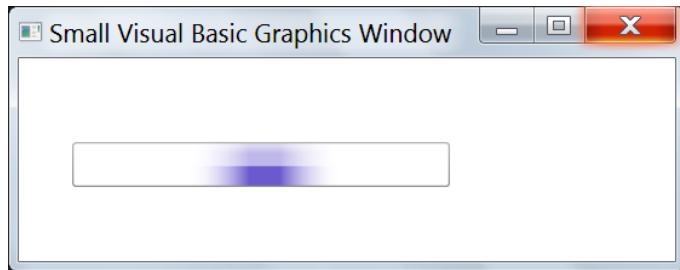
Example:

In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file:

```
ProgressBar1 = Controls.AddProgressBar(  
    35, 55, 250, 30,  
    0, 0  
)
```

Press F5 to run the program. The graphics window will display

the progress bar with an endlessly moving marquee.



Controls.AddRadioButton(**caption**, **left**, **top**, **groupName**, **checked**) As RadioButton

Adds a new [RadioButton control](#) to the graphics window.

Parameters:

- **caption:**

The text to display on the RadioButton

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **groupName:**

The name of the group to add the button to.

- **checked:**

The value to set to the Checked property

Returns:

The key of the RadioButton. sVB can deal with this key as an object of type [RadioButton](#), so, you can access the RadioButton properties, methods and events via it.

Example:

In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file: It creates 2 groups of radio buttons, each contains two buttons. It also adds a label to show the chosen position:

```
' Radio buttons group1
RdoTop = Controls.AddRadioButton(
    "Top", 35, 30, "group1", False)
RdoBottom = Controls.AddRadioButton(
    "Bottom", 235, 30, "group1", True)

' Radio buttons group2
RdoLeft = Controls.AddRadioButton(
    "Left", 35, 80, "group2", True)
RdoRight = Controls.AddRadioButton(
    "Right", 235, 80, "group2", False)

LblPos = Controls.AddLabel("", 35, 120)

' Only one event handler for all radio buttons:
RdoTop.OnCheck = OnCheck
RdoBottom.OnCheck = OnCheck
RdoLeft.OnCheck = OnCheck
RdoRight.OnCheck = OnCheck

' Call the handler once to handle initial radio buttons
' states which are set by the AddRadioButton method
OnCheck()

Sub OnCheck()
    pos = ""
    If RdoTop.Checked Then
        pos = "Top"
    Else
        pos = "Bottom"
    EndIf
```

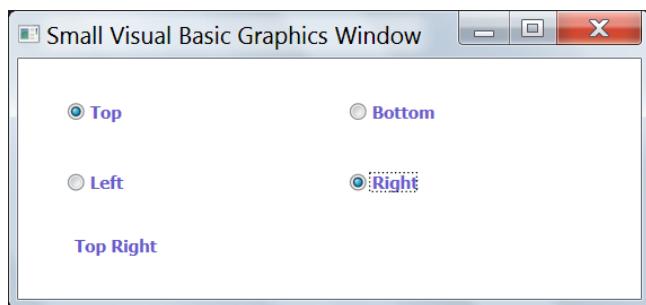
```

If RdoLeft.Checked Then
    pos = pos + " Left"
Else
    pos = pos + " Right"
EndIf

Controls.Text = pos
EndSub

```

Press F5 to run the program. The graphics window will display the 4 radio buttons with the "Bottom" and "Left" radio buttons selected. The label will show the chosen position, and will be updated every time you change the selected radio buttons.



Controls.AddScrollBar(left, top, width, height, minimum, maximum, value) As ScrollBar

Adds a new [ScrollBar control](#) to the graphics window.

Parameters:

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

- **minimum:**

The scrollbar minimum value

- **maximum:**

The scrollbar maximum value.

- **value:**

The scrollbar current value.

Returns:

The key of the Scrollbar. sVB can deal with this key as an object of type [Scrollbar](#), so, you can access the Scrollbar properties, methods and events via it.

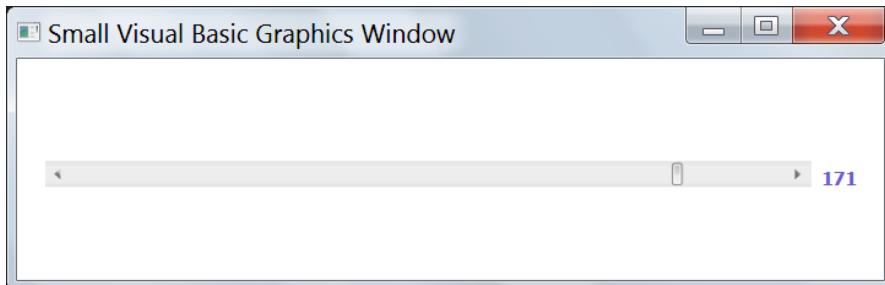
Example:

In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file:

```
ScrollBar1 = Controls.AddScrollBar(
    260, -175, -1, 500,
    0, 200, 50
)
' Rotate the scroll bar to be horizontal
ScrollBar1.Angle = -90
ScrollBar1.OnScroll = ScrollBar1_OnScroll
LblValue = Controls.AddLabel("50", 520, 65)

Sub ScrollBar1_OnScroll()
    LblValue.Text = Math.Round(ScrollBar1.Value)
EndSub
```

Press F5 to run the program. The graphics window will display the scroll bar, and the label will show its value (which is indicated by the thumb position). The label will be updated every time you change the thumb position of the scroll bar.



Controls.AddSlider(left, top, width, height, minimum, maximum, value, tickFrequency) As Slider

Adds a new [Slider control](#) to the graphics window.

Parameters:

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

- **minimum:**

The slider minimum value.

- **maximum:**

The slider maximum value.

- **value:**

The slider current value.

- **tickFrequency:**

The distance between slide ticks.

Returns:

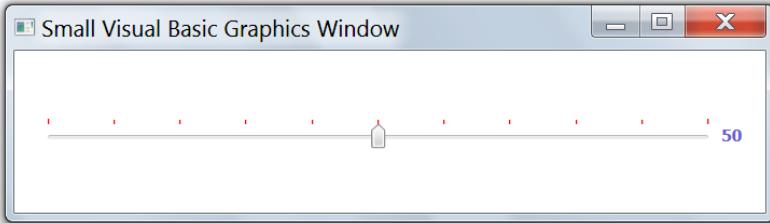
The key of the Slider. sVB can deal with this key as an object of type [Slider](#), so, you can access the Slider properties, methods and events via it.

Example:

In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file:

```
Slider1 = Controls.AddSlider(  
    20, 50, 500, 30,  
    0, 100, 50, 10  
)  
  
Slider1.ForeColor = Colors.Red  
Slider1.SnapToTick = True  
Slider1.OnSlide = Slider1_OnSlide  
LblValue = Controls.AddLabel("50", 520, 50)  
  
Sub Slider1_OnSlide()  
    LblValue.Text = Math.Round(Slider1.Value)  
EndSub
```

Press F5 to run the program. The graphics window will display the slider, and the label will show its value (which is indicated by the thumb position). The label will be updated every time you slide the thumb on the slider.



Controls.AddMultiLineTextBox(left, top)

Adds a multi-line textbox to the graphics window at the specified position, which allows the user to insert text into new lines by pressing Enter from keyboard.

Note that sVB kept this method for backward compatibility with Small Basic, but its job can be done by the Controls.AddTextBox method, where you can set the MultiLine property of the textbox to true, as we will show in the next example.

Parameters:

- **left:**

The x co-ordinate of the text box.

- **top:**

The y co-ordinate of the text box.

Returns:

The key of the TextBox. sVB can deal with this key as an object of type [TextBox](#), so, you can access the TextBox properties, methods and events via it.

Controls.AddTextBox(left, top) As TextBox

Adds a new [TextBox control](#) to the graphics window.

Parameters:

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

Returns:

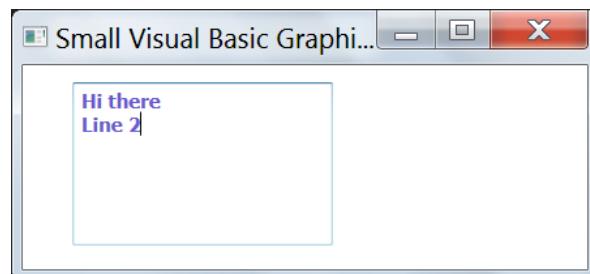
The key of the TextBox. sVB can deal with this key as an object of type [TextBox](#), so, you can access the TextBox properties, methods and events via it.

Example:

In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file:

```
TextBox1 = Controls.AddTextBox(30, 10)
TextBox1.Text = "Hi there"
' Make the text multi-line
TextBox1.MultiLine = True
TextBox1.Height = 100
```

Press F5 to run the program. The graphics window will display the textbox as shown in the picture:



Controls.AddTimer(interval) As WinTimer

Adds a new [WinTimer control](#) to the graphics window in runtime. This allows you to create as many timers as you need, which is not possible when you use the [Small Basic Timer](#).

Parameters:

- **interval:**

The delay time in milliseconds between ticks.

Returns:

The key of the timer. sVB can deal with this key as an object of type [WinTimer](#), so, you can access the WinTimer properties, methods and events via it.

Example:

In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file:

```
T1 = Controls.AddTimer(1000)
T1.OnTick = T1_OnTick

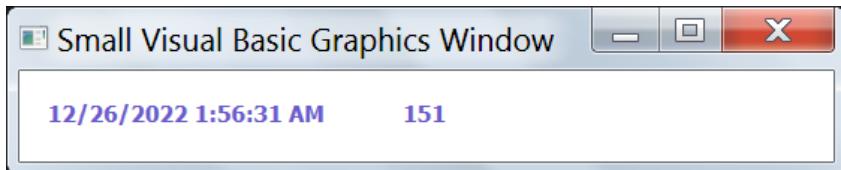
T2 = Controls.AddTimer(100)
T2.OnTick = T2_OnTick

Lb11 = Controls.AddLabel("", 10, 10)
Lb12 = Controls.AddLabel("", 200, 10)
T1_OnTick()

Sub T1_OnTick()
    Lb11.Text = Date.Now
EndSub

Sub T2_OnTick()
    Lb12.Text = Date.GetMillisecond(Date.Now)
EndSub
```

Press F5 to run the program. The graphics window will display the two labels, the first displays the current date and time and is updated by the first timer every second, while the second displays the current milliseconds and is updated every 0.1 second:



Controls.AddToggleButton(**caption**, **left**, **top**, **width**, **height**) As ToggleButton

Adds a new [ToggleButton control](#) to the graphics window.

Parameters:

- **caption:**

The text to display on the toggle button.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

Returns:

The key of the ToggleButton. sVB can deal with this key as an

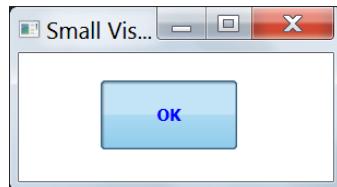
object of type [ToggleButton](#), so, you can access the ToggleButton properties, methods and events via it.

Example:

In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file:

```
ToggleButton1 = Controls.AddToggleButton(  
    "OK", 60, 50, 150, 50)  
ToggleButton1.ForeColor = Colors.Blue  
ToggleButton1.OnCheck = ToggleButton1_OnCheck  
ToggleButton1.Checked = True  
  
Sub ToggleButton1_OnCheck()  
    ToggleButton1.FontBold = ToggleButton1.Checked  
EndSub
```

Press F5 to run the program. The graphics window will display the toggle button, and it will be checked. Click it to toggle its check state, and notice that its font weight changes from Bold to normal and vice versa as its checked state changes.



⚡[Controls.ButtonClicked](#)

Raises an event when any button control is clicked. This is a Small Basic legacy event, and you should handle the Button.OnClick event for each button individually.

Controls.GetButtonCaption(buttonName)

Gets the current caption of the specified button. This is a Small Basic legacy method, and you should use the Button.Text property instead.

Parameter:

- **buttonName:**

The Button whose caption is requested.

Returns:

The current caption of the button.

Controls.GetTextBoxText(textBoxName)

Gets the current text of the specified TextBox. This is a Small Basic legacy method, and you should use the TextBox.Text property instead.

Parameter:

- **textBoxName:**

The TextBox whose text is requested.

Returns:

The text in the TextBox.

Controls.HideControl(controlName)

Hides an already added control. This is a Small Basic legacy method, and you should set the Control's Visible property to False instead.

Parameter:

- **controlName:**

The name of the control.

Controls.LastClickedButton

Gets the last Button that was clicked on the Graphics Window.

Controls.LastTypedTextBox

Gets the last TextBox, that text was typed into.

Controls.Move(control, x, y)

Moves the control with the specified name to a new position.

Parameters:

- **control:**

The name of the control to move.

- **x:**

The x co-ordinate of the new position.

- **y:**

The y co-ordinate of the new position.

Controls.Remove(controlName)

Removes the given control from the Graphics Window.

Parameter:

- **controlName:**

The name of the control to be removed.

Example:

In sVB, switch to the code tab and open a new small basic file (by clicking the New button on the toolbar or pressing Ctrl+N). Add this code to the new file:

```
Btn = Controls.AddButton("test", 10, 10)
Btn.OnClick = Click

Sub Click()
    Controls.Remove(Btn)
EndSub
```

Press F5 to run the program. The graphics window will display a button, and when you click it will be removed!

Controls.SetButtonCaption(buttonName, caption)

Sets the caption of the specified button. This is a Small Basic legacy method, and you should use the Button.Text property instead.

Parameters:

- **buttonName:**

The Button whose caption needs to be set.

- **caption:**

The new caption for the button.

Controls.SetSize(control, width, height)

Sets the size of the control.

Parameters:

- **control:**

The name of the control to be resized.

- **width:**

The width of the control.

- **height:**

The height of the control.

Controls.SetTextBoxLayout(textBoxName, text)

Sets the text of the specified TextBox. This is a Small Basic legacy method, and you should use the TextBox.Text property instead.

Parameters:

- **textBoxName:**

The TextBox whose text needs to be set.

- **text:**

The new text for the TextBox.

Controls>ShowControl(controlName)

Shows a previously hidden control. This is a Small Basic legacy method, and you should set the Control's Visible property to True instead.

Parameter:

- **controlName:**

The name of the control.

Controls.TextTyped

Raises an event when text is typed into any TextBox control. This is a Small Basic legacy event, and you should handle the TextBox.OnTextChanged event for each textbox instead.

The ControlTypes Type

Works as an enum that contains the names of winforms controls, to make it easy for you to check the control type name that is returned by the [Control.TypeName](#) property.

The ControlTypes type has these properties, which return the name of the corresponding control:

 **Button**

 **CheckBox**

 **ComboBox**

 **Control**

 **DatePicker**

 **Form**

 **Label**

 **ListBox**

 **MainMenu**

 **MenuItem**

 **ProgressBar**

 **RadioButton**

 **ScrollBar**

 **Slider**

 **TextBox**

 **ToggleButton**

The Date Type

Provides methods to deal with date, time and durations.

Note that sVB supports [date literals](#), like:

```
#2/17/2022#
#05:22:00 AM#
#17 Feb 2022 05:22:00 PM#
```

and time span literals, like:

```
#+1.05:22:00#
#-15:22:00.1#
```

sVB considers all these literals to be of the Date type, so you can assign them to date variables, and send them as arguments to the Date type methods.

Note also that you can use the + and – operators to add and subtract two dates, and use the =, >, >=, <, and <= operators to compare two dates.

The Date type provides you with these members to deal with date, time, and time span values:

Date.Add(date, duration) As Date

Creates a new date by adding the given duration to the given date. Note that you can also use the + operator directly to add dates and durations.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **duration:**

A date representing the duration you want to add.

Returns:

A new date carrying the result. The input date will not be affected.

Examples:

```
D1 = #+3:10:00#
' Add a date and a positive duration
D2 = Date.Add(#1/1/2023#, D1)
TextWindow.WriteLine(D2)
TextWindow.WriteLine(#1/1/2023# + D1)

' Add a date and a negative duration
TextWindow.WriteLine(D2.Add(#-3:10:00#))
TextWindow.WriteLine(D2 + #-3:10:00#)

' Add two durations
TextWindow.WriteLine(D1.Add(#+1.1:20:50#))
TextWindow.WriteLine(D1 + #+1.1:20:50#)
```

The text window will show these results:

```
01/01/2023 03:10:00 AM
01/01/2023 03:10:00 AM
01/01/2023 12:00:00 AM
01/01/2023 12:00:00 AM
1.04:30:50
1.04:30:50
```

⌚ Date.AddDays(date, value) As Date

Creates a new date by adding the given days to the given date.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

The number of days you want to add. Use negative numbers to subtract days. You can also use decimal numbers like 1.5 days.

Returns:

A new date carrying the result. The input date will not be affected.

Examples:

```
D1 = #10 Jun 2022#
```

```
D2 = #-5:10:30#
```

```
TextWindow.WriteLine({  
    Date.AddDays(#1/13/2023#, 1),  
    Date.AddDays(#1-1-2023 12:20:15 PM#, 20),  
    Date.AddDays(#+1.5:15#, 30),  
    D1.AddDays(1.5),  
    D2.AddDays(100)  
})
```

The text window will show these results:

```
1/14/2023 12:00:00 AM  
1/21/2023 12:20:15 PM  
31.05:15:00  
6/11/2022 12:00:00 PM  
99.18:49:30
```

Date.AddHours(date, value) As Date

Creates a new date by adding the given hours to the given date.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

The number of hours you want to add. Use negative numbers to subtract hours. You can also use decimal numbers like 1.5 hours.

Returns:

A new date carrying the result. The input date will not be affected.

Examples:

```
D1 = #10 Jun 2022#
```

```
D2 = #-5:10:30#
```

```
TextWindow.WriteLine({  
    Date.AddHours(#1/13/2023#, 1),  
    Date.AddHours(#1-1-2023 12:20:15 PM#, 20),  
    Date.AddHours(#+1.5:15#, -30),  
    D1.AddHours(1.5),  
    D2.AddHours(100)  
})
```

The text window will show these results:

```
1/13/2023 1:00:00 AM  
1/2/2023 8:20:15 AM  
-00:45:00  
6/10/2022 1:30:00 AM  
3.22:49:30
```

Date.AddMilliseconds(date, value) As Date

Creates a new date by adding the given milliseconds to the given date.

Note that you can use the + operator directly to add ticks to the date, where: 1 ms = 10,000 ticks.

So, if you want to add 0.5 ms to date1 , you can just use:

date1 = date1 + 5000

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

The number of milliseconds to add. Use negative numbers to subtract milliseconds. You can also use decimal numbers like 1.5 if you want to have more accurate ticks.

Returns:

A new date carrying the result. The input date will not be affected.

Examples:

D1 = #10 Jun 2022#

D2 = #-5:10:30#

```
TextWindow.WriteLine({
    Date.AddMilliseconds(#1/13/2023#, 1),
    Date.AddMilliseconds(#1-1-2023 12:20:15 PM#, 20),
    Date.AddMilliseconds(#+1.05:15#, -30),
    D1.AddMilliseconds(1.5),
    D2.AddMilliseconds(100)
})
```

The text window will show these results:

```
01/13/2023 12:00:00.001 AM
01/01/2023 12:20:15.02 PM
1.05:14:59.97
06/10/2022 12:00:00.0015 AM
-05:10:29.9
```

⌚ Date.AddMinutes(date, value) As Date

Creates a new date by adding the given minutes to the given date.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

The number of minutes you want to add. Use negative numbers to subtract minutes. You can also use decimal numbers like 1.5 minutes.

Returns:

A new date carrying the result. The input date will not be affected.

Examples:

D1 = #10 Jun 2022#

D2 = #-5:10:30#

```
TextWindow.WriteLine({
    Date.AddMinutes(#1/13/2023#, 1),
    Date.AddMinutes(#1-1-2023 12:20:15 PM#, 20),
    Date.AddMinutes(#+1.5:15#, -30),
    D1.AddMinutes(1.5),
    D2.AddMinutes(100)
})
```

The text window will show these results:

```
01/13/2023 12:01:00 AM
01/01/2023 12:40:15 PM
1.04:45:00
06/10/2022 12:01:30 AM
-03:30:30
```

⌚ Date.AddMonths(date, value) As Date

Creates a new date by adding the given months to the given date.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

The number of months you want to add.

You can use negative numbers to subtract months.

There are two cases you should avoid:

1. When you deal with a date, don't add a decimal number, otherwise it will be rounded to the nearest integer.
2. When you deal with a duration, don't add months, because months have different day numbers. Otherwise, the total days of the given months will be calculated

assuming that a month = 30.4375 days in average, so that 12 months = 365.2425 days!

Returns:

A new date carrying the result. The input date will not be affected.

Examples:

D1 = #10 Jun 2022#

D2 = #-5:10:30#

```
TextWindow.WriteLine({  
    Date.AddMonths(#1/13/2023#, 1),  
    Date.AddMonths(#1-1-2023 12:20:15 PM#, 20),  
    Date.AddMonths(#+1.5:15#, -30),  
    D1.AddMonths(1.5),  
    D2.AddMonths(100)  
})
```

The text window will show these results:

```
02/13/2023 12:00:00 AM  
09/01/2024 12:20:15 PM  
-911.21:45:00  
08/10/2022 12:00:00 AM  
3043.12:49:30
```

⌚ Date.AddSeconds(date, value) As Date

Creates a new date by adding the given seconds to the given date.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

The number of seconds you want to add. Use negative numbers to subtract seconds. You can also use decimal numbers like 1.5 seconds.

Returns:

A new date carrying the result. The input date will not be affected.

Examples:

```
D1 = #10 Jun 2022#
```

```
D2 = #-5:10:30#
```

```
TextWindow.WriteLine({
```

```
    Date.AddSeconds(#1/13/2023#, 1),
```

```
    Date.AddSeconds(#1-1-2023 12:20:15 PM#, 20),
```

```
    Date.AddSeconds(#+1.5:15#, -30),
```

```
    D1.AddSeconds(1.5),
```

```
    D2.AddSeconds(100)
```

```
)
```

The text window will show these results:

```
01/13/2023 12:00:01 AM
01/01/2023 12:20:35 PM
1.05:14:30
06/10/2022 12:00:01.5 AM
-05:08:50
```

⌚ Date.AddYears(date, value) As Date

Creates a new date by adding the given years to the given date.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you

call this method as an extension method from a date variable.

- **value:**

The number of years you want to add.

There are two cases you should avoid:

1. When you deal with a date, don't add a decimal number, otherwise it will be rounded to the nearest integer.
2. When you deal with a duration, don't add years, because leap years have 366 days, while other years have 365 days. Otherwise, the total days of the given years will be calculated assuming that a year = 365.2425 days!

Returns:

A new date carrying the result. The input date will not be affected.

Examples:

```
D1 = #10 Jun 2022#
```

```
D2 = #-5:10:30#
```

```
TextWindow.WriteLine({  
    Date.AddYears(#1/13/2023#, 1),  
    Date.AddYears(#1-1-2023 12:20:15 PM#, 20),  
    Date.AddYears(#+1.5:15#, -30),  
    D1.AddYears(1.5),  
    D2.AddYears(100)  
})
```

The text window will show these results:

```
01/13/2024 12:00:00 AM  
01/01/2043 12:20:15 PM  
-10956.06:45:00  
06/10/2024 12:00:00 AM  
36524.18:49:30
```

Date.ChangeDay(date, value) As Date

Creates a new date based on the given date or duration, with the day(s) part set to the given value.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

An integer number in the range:

- (1-31) for dates.
- (0-10675199) for durations.

Otherwise this method will do nothing and return the input date without any change.

Note That:

- If you supply a decimal number, it will be rounded to the nearest integer.
- If you supply a negative number, its absolute value will be used.
- If the input date is a duration, the days part will be set to the same sign of the duration, so you don't have to worry about the sign yourself. Just focus on the value.

Returns:

A new date with the new day value. The input date will not change.

Examples:

D1 = #10 Jun 2022#

D2 = #-5:10:30#

```
TextWindow.WriteLine({
    Date.ChangeDay(#1/13/2023#, 1),
    Date.ChangeDay(#1-1-2023 12:20:15 PM#, 20),
    Date.ChangeDay(#+1.5:15#, -30),
    D1.ChangeDay(1.5),
    D2.ChangeDay(100)
})
```

The text window will show these results:

```
01/01/2023 12:00:00 AM
01/20/2023 12:20:15 PM
30.05:15:00
06/02/2022 12:00:00 AM
-100.05:10:30
```

⌚ Date.ChangeHour(date, value) As Date

Creates a new date based on the given date or duration with the hour(s) part set to the given value.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

A number between 0 and 23 that represents the new hour, otherwise this method will do nothing and return the input date without any change.

Note That:

- If you supply a decimal number, it will be rounded to the nearest integer.

- If you supply a negative number, its absolute value will be used.
- If the input date is a duration, the hours part will be set to the same sign of the duration, so you don't have to worry about the sign yourself. Just focus on the value.

Returns:

A new date with the new hour value. The input date will not change.

Examples:

```
D1 = #10 Jun 2022#
```

```
D2 = #-5:10:30#
```

```
TextWindow.WriteLine({  
    Date.ChangeHour(#1/13/2023#, 1),  
    Date.ChangeHour(#1-1-2023 12:20:15 PM#, 0),  
    Date.ChangeHour(#+1.5:15#, -12),  
    D1.ChangeHour(1.5),  
    D2.ChangeHour(23)  
})
```

The text window will show these results:

```
01/13/2023 01:00:00 AM  
01/01/2023 12:20:15 AM  
1.12:15:00  
06/10/2022 02:00:00 AM  
-23:10:30
```

Date.ChangeMillisecond(date, value) As Date

Creates a new date based on the given date or duration with the millisecond(s) part set to the given value.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

A number between 0 and 999 that represents the new millisecond, otherwise this method will do nothing and return the input date without any change.

Note That:

- If you supply a decimal number, it will be rounded to the nearest integer.
- If you supply a negative number, its absolute value will be used.
- If the input date is a duration, the milliseconds part will be set to the same sign of the duration, so you don't have to worry about the sign yourself. Just focus on the value.

Returns:

A new date with the new millisecond value. The input date will not change.

Examples:

D1 = #10 Jun 2022#

D2 = #-5:10:30#

```
TextWindow.WriteLine({  
    Date.ChangeMillisecond(#1/13/2023#, 1),  
    Date.ChangeMillisecond(#1-1-2023 12:20:15.100 PM#, 0),  
    Date.ChangeMillisecond(#+1.5:15#, -12),  
    D1.ChangeMillisecond(1.5),  
    D2.ChangeMillisecond(23)  
})
```

The text window will show these results:

```
01/13/2023 12:00:00.001 AM  
01/01/2023 12:20:15 PM  
1.05:15:00.012  
06/10/2022 12:00:00.002 AM  
-05:10:30.023
```

⌚ Date.ChangeMinute(date, value) As Date

Creates a new date based on the given date or duration with the minute(s) part set to the given value.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

A number between 0 and 59 that represents the new minute, otherwise this method will do nothing and return the input date without any change.

Note That:

- If you supply a decimal number, it will be rounded to the nearest integer.

- If you supply a negative number, its absolute value will be used.
- If the input date is a duration, the minutes part will be set to the same sign of the duration, so you don't have to worry about the sign yourself. Just focus on the value.

Returns:

A new date with the new minute value. The input date will not change.

Examples:

```
D1 = #10 Jun 2022#
```

```
D2 = #-5:10:30#
```

```
TextWindow.WriteLine({  
    Date.ChangeMinute(#1/13/2023#, 1),  
    Date.ChangeMinute(#1-1-2023 12:20:15 PM#, 0),  
    Date.ChangeMinute(#+1.5:15#, -12),  
    D1.ChangeMinute(1.5),  
    D2.ChangeMinute(23)  
})
```

The text window will show these results:

```
01/13/2023 12:01:00 AM  
01/01/2023 12:00:15 PM  
1.05:12:00  
06/10/2022 12:02:00 AM  
-05:23:30
```

Date.ChangeMonth(date, value) As Date

Creates a new date based on the given date with the month part set to the given value.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

A number between 1 and 12 that represents the month, otherwise this method will do nothing and return the input date without any change.

Note That:

- If you supply a decimal number, it will be rounded to the nearest integer.
- If you supply a negative number, its absolute value will be used.
- If the input date is a duration, it will be returned as it is, because durations don't contain a months part.

Returns:

A new date with the new month value. The input date will not change.

Examples:

D1 = #10 Jun 2022#

D2 = #-5:10:30#

D3 = #1-1-2023 12:20:15 PM#

```
TextWindow.WriteLine({
    Date.ChangeMonth(#1/13/2023#, 1),
    Date.ChangeMonth(#+1.5:15#, -12),
    D1.ChangeMonth(1.5),
    D2.ChangeMonth(6),
    D3.ChangeMonth(3)
})
```

The text window will show these results:

```
01/13/2023 12:00:00 AM
1.05:15:00
02/10/2022 12:00:00 AM
-05:10:30
03/01/2023 12:20:15 PM
```

⌚ Date.ChangeSecond(date, value) As Date

Creates a new date based on the given date or duration with the second(s) part set to the given value.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

A number between 0 and 59 that represents the new second, otherwise this method will do nothing and return the input date without any change.

Note That:

- If you supply a decimal number, it will be rounded to the nearest integer.

- If you supply a negative number, its absolute value will be used.
- If the input date is a duration, the seconds part will be set to the same sign of the duration, so you don't have to worry about the sign yourself. Just focus on the value.

Returns:

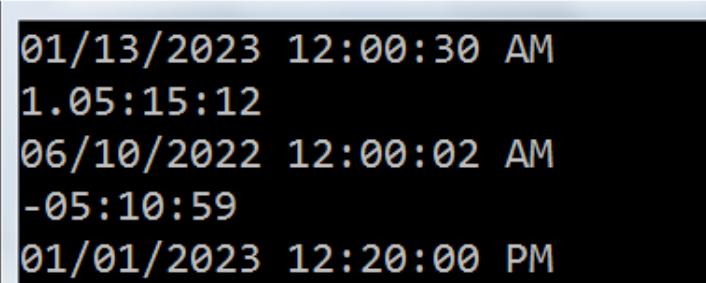
A new date with the new second value. The input date will not change.

Examples:

```
D1 = #10 Jun 2022#
D2 = #-5:10:30#
D3 = #1-1-2023 12:20:15 PM#
```

```
TextWindow.WriteLine({
    Date.ChangeSecond(#1/13/2023#, 30),
    Date.ChangeSecond(#+1.5:15#, -12),
    D1.ChangeSecond(1.5),
    D2.ChangeSecond(59),
    D3.ChangeSecond(0)
})
```

The text window will show these results:



```
01/13/2023 12:00:30 AM
1.05:15:12
06/10/2022 12:00:02 AM
-05:10:59
01/01/2023 12:20:00 PM
```

Date.ChangeYear(date, value) As Date

Creates a new date based on the given date with the year set to the given value.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

The new value of the year part.

Note That:

- If you supply a decimal number, it will be rounded to the nearest integer.
- If you supply a negative number, its absolute value will be used.
- If the input date is a duration, it will be returned as it is, because durations don't contain a years part.

Returns:

A new date with the new year value. The input date will not change.

Examples:

D1 = #10 Jun 2022#

D2 = #-5:10:30#

D3 = #1-1-2023 12:20:15 PM#

```
TextWindow.WriteLine({  
    Date.ChangeYear(#1/13/2023#, 2021),  
    Date.ChangeYear(#+1.5:15#, -2012),  
    D1.ChangeYear(2111.5),  
    D2.ChangeYear(1960),  
    D3.ChangeYear(2020)  
})
```

The text window will show these results:

```
01/13/2021 12:00:00 AM  
1.05:15:00  
06/10/2112 12:00:00 AM  
-05:10:30  
01/01/2020 12:20:15 PM
```

⌚ Date.CreateDuration(days, hours, minutes, seconds, milliseconds) As Date

Creates a time span from given duration parts.

Parameters:

- **days:**

The total days in the duration. For example, 1000 days approximately represents 2 years and 9 months. You can use negative values.

- **hours:**

The hours part of the duration. You can use negative values.

- **minutes:**

The minutes part of the duration. You can use negative values.

- **seconds:**

The seconds part of the duration. You can use negative values.

- **milliseconds:**

The milliseconds part of the duration. You can use negative values.

Returns:

A new duration calculated from the given parts.

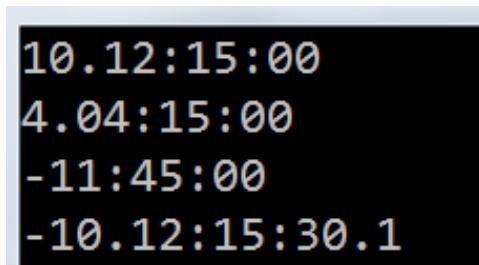
Note that if you supply a decimal value for any part it will be rounded. If you supply values that exceed the max value of any part (like 30 hours), the extra value will be added to the next part (30 hours = 1 days and 6 hours).

To have a negative duration, supply negative values for all parts. If you supply a mixture of positive and negative values, the duration will be calculated from these parts, and the result will not seem like the values you supplied.

Examples:

```
TextWindow.WriteLine({  
    Date.CreateDuration(10, 12, 15, 0, 0),  
    Date.CreateDuration(0, 100, 15, 0, 0),  
    Date.CreateDuration(0, -12, 15, 0, 0),  
    Date.CreateDuration(-10, -12, -15.4, -30, -100)  
})
```

The text window will show these results:



```
10.12:15:00  
4.04:15:00  
-11:45:00  
-10.12:15:30.1
```

Date.DateTime As String

Gets the date and time representation in the user local culture. This is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetDateTime](#) method. See more notes and examples there.

Date.Day As Double

Gets the day(s) part of the date (or duration).

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetDay](#) method. See more notes and examples there.

Date.DayName As String

Gets the local name of the week day of the date.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetDayName](#) method. See more notes and examples there.

Date.DayOfWeek As Double

Gets the day of the week of the date.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetDayOfWeek](#) method. See more notes and examples there.

Date.DayOfYear As Double

Gets the day number in the year of the date.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetDayOfYear](#) method. See more notes and examples there.

Date.ElapsedMilliseconds As Double

Calculates the difference in milliseconds between the start of the year 1900 and the date.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetElapsedMilliseconds](#) method. See more notes and examples there.

Date.EnglishDayName As String

Gets the English name of the week day of the date.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetEnglishDayName](#) method. See more notes and examples there.

Date.EnglishMonthName As String

Gets the English name of the month of the date.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetEnglishMonthName](#) method. See more notes and examples there.

Date.FromCulture(dateText, cultureName) As Date

Creates a new date from the given text if its format is a valid date format for the given culture.

Parameters:

- **dateText:**

The text that represents the date.

- **cultureName:**

The culture name used to format the date, like :

- "en-US" for English (United States) culture.
- "fr-Fr" for French (France) culture.
- "ar-EG" for Arabic (Egypt) culture.
- "ar-SA" for Arabic (Saudi Arabia) culture.

For the complete list of supported cultures, see the "Language tag" column in the [list of language/region names supported by Windows](#), but I think you can recognize the pattern from the above examples.

You can send empty string "" if you want to use the local culture of the user's system.

Returns:

A new date if the date text is in correct format for the given culture, otherwise an empty string.

Examples:

In this code, we will get 3 dates from 3 different cultures: Arabic of Egypt, Arabic of Saudi Arabia (which uses the Hegri calendar), and French of France:

```
TextWindow.WriteLine({  
    Date.FromCulture("2023 يناير 1", "ar-EG"),  
    Date.FromCulture("1444 رمضان 1", "ar-SA"),  
    Date.FromCulture("15 Janvier 2023", "fr-Fr")  
})
```

The text window will show these results:

```
01/01/2023 12:00:00 AM  
03/23/2023 12:00:00 AM  
01/15/2023 12:00:00 AM
```

⌚ Date.FromDate(year, month, day) As Date

Creates a new date from the given year, month and day values.

Parameters:

- **year**:

A positive number that represents the year.

- **month**:

A number between 1 and 12 representing the month.

- **day**:

A number between 1 and 31 representing the day. If you supply a 31 value to a 30-days month this will cause an error and this method will return an empty string. The same will happen if you supply wrong day (29, 30, 31) to February, with respect to leap years.

Returns:

If all given parts are within valid ranges, this method returns a new date that represents those parts, otherwise returns an empty string.

Example:

```
D1 = Date.FromDate(2023, 1, 16)
TextWindow.WriteLine(D1.EnglishMonthName) ' January
```

Date.FromDateTime(year, month, day, hour, minute, second, millisecond) As Date

Creates a new date from the given date and time values.

Parameters:

- **year**:

A positive number that represents the year.

- **month**:

A number between 1 and 12 representing the month.

- **day**:

A number between 1 and 31 representing the day. If you supply a 31 value to a 30-days month this will cause an error and this method will return an empty string. The same will happen if you supply wrong day (29, 30, 31) to February, with respect to leap years.

- **hour**:

A number between 0 and 23 representing the hour.

- **minute**:

A number between 0 and 59 representing the minute.

- **second**:

A number between 0 and 59 representing the second.

- **millisecond**:

A number between 0 and 999 representing the millisecond.

Returns:

If all given parts are within valid ranges, this method returns a new date that represents those parts, otherwise returns an empty string.

Example:

```
TextWindow.WriteLine(  
    Date.FromDateTime(2023, 2, 28, 0, 30, 11, 0))
```

The text window will show:

2/28/2023 12:30:11 AM

Date.FromTime(hour, minute, second, millisecond) As Date

Creates a new date from the given time values and the default date 1/1/0001.

Parameters:

- **hour**:

A number between 0 and 23 representing the hour.

- **minute**:

A number between 0 and 59 representing the minute.

- **second**:

A number between 0 and 59 representing the second.

- **millisecond**:

A number between 0 and 999 representing the millisecond.

Returns:

If all given parts are within valid ranges, this method returns a new date that represents those parts, otherwise returns an

empty string.

Example:

```
D1 = Date.FromTime(13, 30, 11, 0)
TextWindow.WriteLine(D1)                      ' 01:30:11 PM
TextWindow.WriteLine(D1.ShortDate)            ' 01/01/0001
```

Date.GetDateAndTime(date) As String

Gets the date and time string representation in the user local culture.

If you want to show date and time in the English culture, just show the value of the date variable directly.

If you want to show the date in any other culture, use the [Date.ToCulture](#) method.

If you want to show the date in any custom format, use the Date methods to get the date parts and use them to generate the format you want.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property (with the name DateAndTime).

Returns:

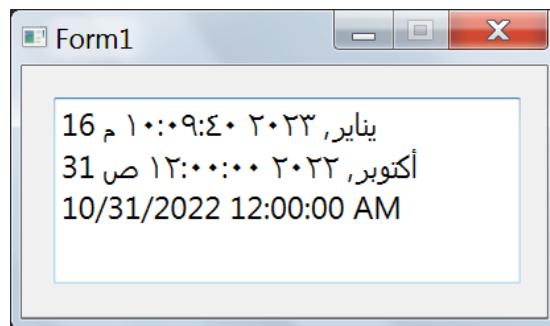
A string representing the long date and time.

Examples:

Add a TextBox on the form, and use this code:

```
TextBox1.AppendLine(Date.GetDateAndTime(Date.Now))
D = #10/31/2022#
TextBox1.AppendLine(D.DateAndTime)
TextBox1.AppendLine(D)
```

Press F5 to run the program. The textbox will show the two dates in your culture, and it will also show the second date in the short form with the English (United States) culture. This is how it looks on my PC with the Arabic(Egypt) culture (which will look better if the TextBox shows the text from right to left, which can be done manually by pressing Right Control + Right Shift, or programmatically by setting the TextBox.RightToLeft property to True):



⌚ Date.GetDay(date) As Double

Gets the day(s) part of the given date (or duration).

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "Day".

Returns:

A number between 1 and 31 that represents the day of the date, or a positive or negative number to represent the days part of the duration. If the input is not valid, this method returns an empty string.

Examples:

```
D1 = #10/31/2022#
D2 = #-100.00:30#
TextWindow.WriteLine({
    Date.GetDay(Date.Now) = Clock.Day, ' True
    D1.Day, ' 31
    D2.Day    '-100
})
```

⌚ Date.GetDayName(date) As String

Gets the local name of the week day of the given date.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "DayName".

Returns:

The name of the week day in the local language defined on the user system.

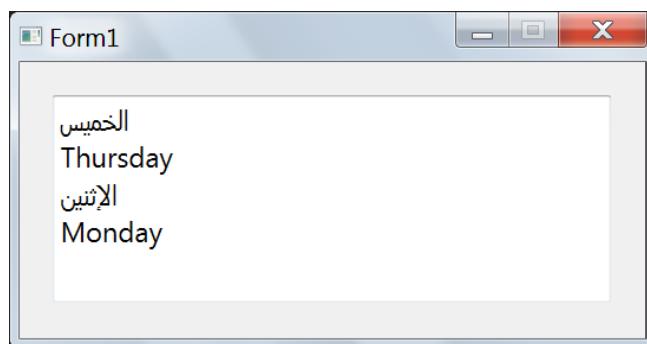
Examples:

Add a TextBox on the form, and use this code:

```
D = #10/31/2022#
```

```
TextBox1.AppendLines({  
    Date.GetDayName(Date.Now),  
    Date.GetEnglishDayName(Date.Now),  
    D.DayName,  
    D.EnglishDayName  
})
```

Press F5 to run the program. The textbox will show the names of the two days both in your local and English culture. This is how it looks on my PC with the Arabic(Egypt) culture:



⌚ Date.GetDayOfWeek(date) As Double

Gets the day number in the week for the given date.

Note that Sunday is the first day of the week.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "DayOfWeek".

Returns:

A number between 1 and 7 that Represents the day in the week.

Examples:

```
D = #10/31/2022#
TextWindow.WriteLine({
    D.DayOfWeek, '2
    Date.GetDayOfWeek(D.AddDays(1)), '3
    Date.GetDayOfWeek(D.AddDays(-1)) '1
})
```

⌚Date.GetDayOfYear(date) As Double

Gets the day number in the year for the given date.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "DayOfYear".

Returns:

A number between 1 and 366 that Represents the day in the year.

Examples:

```
D = #10/31/2022#
TextWindow.WriteLine({
    D.DayOfYear, '304
    Date.GetDayOfYear(D.AddDays(1)), '305
    Date.GetDayOfYear(D.AddDays(-1)) '303
})
```

Date.GetElapsedMilliseconds(date) As Double

Calculates the difference in milliseconds between the start of the year 1900 and the given date.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "ElapsedMilliseconds".

Returns:

The number of milliseconds that have elapsed since 1900 until the given date, or an empty string if the input is a duration.

Examples:

```
D1 = #1/1/1900 00:00:1#
TextWindow.WriteLine({
    Date.GetElapsedMilliseconds(#1/1/1900#), '0
    D1.ElapsedMilliseconds '1000
})
```

Date.GetEnglishDayName(date) As String

Gets the English name of the week day for the given date.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "EnglishDayName".

Returns:

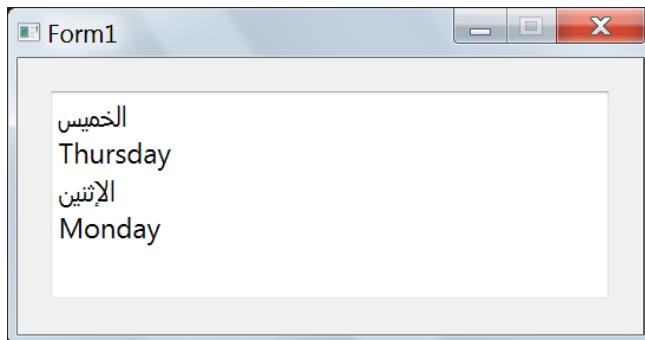
The name of the week day in English.

Examples:

Add a TextBox on the form, and use this code:

```
D = #10/31/2022#
TextBox1.AppendLines({
    Date.GetDayName(Date.Now),
    Date.GetEnglishDayName(Date.Now),
    D.DayName,
    D.EnglishDayName
})
```

Press F5 to run the program. The textbox will show the names of the two days both in your local and English culture. This is how it looks on my PC with the Arabic(Egypt) culture:



⌚ Date.GetEnglishMonthName(date) As String

Gets the English name of the month for the given date.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "EnglishMonthName".

Returns:

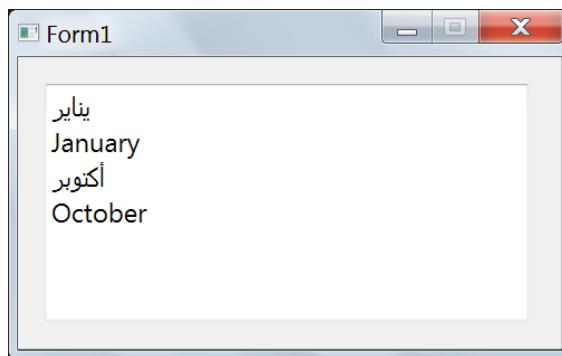
The name of the month in English.

Examples:

Add a TextBox on the form, and use this code:

```
D = #10/31/2022#
TextBox1.AppendLines({
    Date.GetMonthName(Date.Now),
    Date.GetEnglishMonthName(Date.Now),
    D.MonthName,
    D.EnglishMonthName
})
```

Press F5 to run the program. The textbox will show the names of the two months both in your local and English culture. This is how it looks on my PC with the Arabic(Egypt) culture:



⌚Date.GetHour(date) As Double

Gets the hour(s) part of the given date (or duration).

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "Hour".

Returns:

A number between 0 and 23 that represents the hour(s) part.

Examples:

```
D1 = #10/31/2022#
D2 = #-100.03:30#
TextWindow.WriteLine({
    Date.GetHour(Date.Now) = Clock.Hour, ' True
    D1.Hour, ' 0 (which is 12 AM)
    D2.Hour ' -3
})
```

Remember that when the duration is negative, all its parts are negative. This is why D2.Hour returns -3.

⌚Date.GetLongDate(date) As String

Gets the long form of the given date. The long date contains the month name in the user local culture instead of its number.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "LongDate".

Returns:

A string representing the long date.

Examples:

Add a TextBox on the form, and use this code:

```
TextBox1.AppendLine(Date.GetLongDate(Date.Now))
D = #10/31/2022#
```

```
TextBox1.AppendLine(D.LongDate)
TextBox1.Append(D.EnglishDayName + ", ")
TextBox1.Append(D.Day + " ")
TextBox1.Append(D.EnglishMonthName + " ")
TextBox1.Append(D.Year)
```

Press F5 to run the program. The textbox will show the two dates in your culture, then it will show the second date in the long form with a custom format that uses the English (United States) culture. This is how it looks on my PC with the Arabic(Egypt) culture (which will look better if the TextBox shows the text from right to left, which can be done manually by pressing Right Control + Right Shift, or programmatically by setting the TextBox.RightToLeft property to True):



⌚ Date.GetLongTime(date) As String

Gets the time part for the given date, which will include the seconds part and the day time part like AM/PM but in the user's local culture.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "LongTime".

Returns:

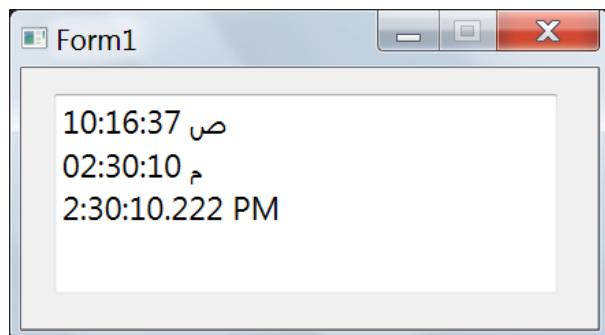
A string representing the long time.

Examples:

Add a TextBox on the form, and use this code:

```
TextBox1.AppendLine(Date.GetLongTime(Date.Now))
D = #10/31/2022 14:30:10.222#
TextBox1.AppendLine(D.LongTime)
H = D.Hour
T = "AM"
If H > 12 Then
    H = H - 12
    T = "PM"
EndIf
TextBox1.Append(H + ":")
TextBox1.Append(D.Minute + ":")
TextBox1.Append(D.Second + ".")
TextBox1.Append(D.Millisecond + " ")
TextBox1.Append(T)
```

Press F5 to run the program. The textbox will show the two long times in your culture, then it will show the second time in the long form with a custom format that uses the English (United States) culture. This is how it looks on my PC with the Arabic(Egypt) culture:



Date.GetMillisecond(date) As Double

Gets the millisecond(s) part of the given date (or duration).

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "Millisecond".

Returns:

A number between 0 and 999 that represents the millisecond(s) part.

Examples:

```
D1 = Date.GetMillisecond(Date.Now)
D2 = #10/31/2022 1:0:0.099#
D3 = #-100.03:30:0.5#
TextWindow.WriteLine(
    D1 <= Clock.Millisecond, ' True
    D2.Millisecond, ' 99
    D3.Millisecond ' -500
)
```

Remember that when the duration is negative, all its parts are negative. This is why D3.Millisecond returns -500.

Also note that Date.GetMillisecond(Date.Now) does exactly what Clock.Millisecond does, and both should return the same results, but you can't execute them both at the same time, because executing Date.GetMillisecond takes a few milliseconds. This is why we used \geq to compare the two values. If you want to ensure that are equal, capture Date.Now, then capture Clock.Millisecond which will have a few nano seconds more, hence when you compare the

milliseconds of both values they will be equal in most cases (but not always, since those elapsed nano seconds can sometimes advance the millisecond part!)

So, if you want to use = , this is the code you should try:

```
Now = Date.Now  
Ms1 = Clock.Millisecond  
Ms2 = Date.GetMillisecond(Now)  
D2 = #10/31/2022 1:0:0.099#  
D3 = #-100.03:30:0.5#  
TextWindow.WriteLine({  
    Ms1 = Ms2, ' True  
    D2.Millisecond, ' 99  
    D3.Millisecond ' -500  
})
```

Date.GetMinute(date) As Double

Gets the minute(s) part of the given date (or duration).

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "Minute".

Returns:

A number between 0 and 59 that represents the minute(s) part.

Examples:

```
D1 = #10/31/2022 1:7:0#  
D2 = #-100.03:30#
```

```
TextWindow.WriteLine({  
    Date.GetMinute(Date.Now) = Clock.Minute, ' True  
    D1.Minute, ' 7  
    D2.Minute ' -30  
})
```

Remember that when the duration is negative, all its parts are negative. This is why D2.Minute returns -30.

⌚ Date.GetMonth(date) As Double

Gets the month part of the given date.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "Month".

Returns:

A number between 1 and 12 that represents the month part of the given date, or 0 if the input is a duration.

Examples:

```
D1 = #10/31/2022 1:7:0#  
D2 = #-100.03:30#  
TextWindow.WriteLine({  
    Date.GetMonth(Date.Now) = Clock.Month, ' True  
    D1.Month, ' 10  
    D2.Month ' 0  
})
```

⌚ Date.GetMonthName(date) As String

Gets the local name of the month of the given date.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "MonthName".

Returns:

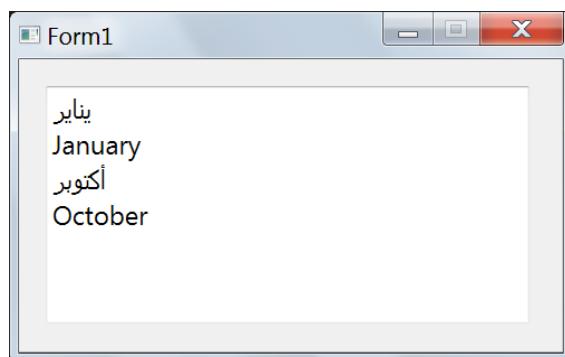
The name of the month in the local language defined on the user system.

Examples:

Add a TextBox on the form, and use this code:

```
D = #10/31/2022#
TextBox1.AppendLines(
    Date.GetMonthName(Date.Now),
    Date.GetEnglishMonthName(Date.Now),
    D.MonthName,
    D.EnglishMonthName
)
```

Press F5 to run the program. The textbox will show the names of the two months both in your local and English culture. This is how it looks on my PC with the Arabic(Egypt) culture:



⌚ Date.GetSecond(date) As Double

Gets the second(s) part of the given date (or duration).

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "Second".

Returns:

A number between 0 and 59 that represents the second(s) part.

Examples:

```
D1 = #10/31/2022 1:7:39#
D2 = #-100.03:30:01#
TextWindow.WriteLine({
    Date.GetSecond(Date.Now) = Clock.Second, ' True
    D1.Second, ' 39
    D2.Second ' -1
})
```

Remember that when the duration is negative, all its parts are negative. This is why D2.Second returns -1.

⌚ Date.GetShortDate(date) As String

Gets the short form of the given date, like 1/1/2020.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "ShortDate".

Returns:

A string representing the short date.

Examples:

```
D = #July 10, 2000#
TextWindow.WriteLine({
    Date.GetShortDate(#19 Sep 2020#), '19/09/2020
    D.ShortDate
})
```

⌚ Date.GetShortTime(date) As String

Gets the short time part of the given date. The time will include hours and minutes and AM or PM (in user's local culture), but not seconds.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "ShortTime".

Returns:

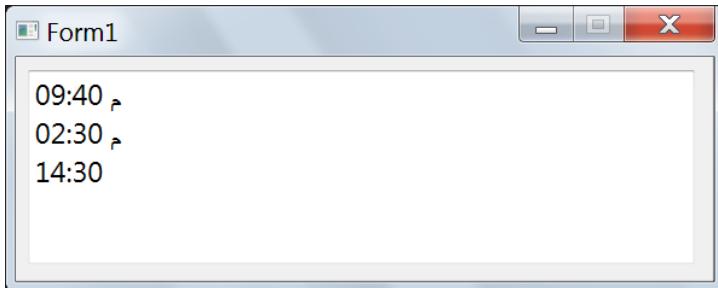
A string representing the short time.

Examples:

Add a TextBox on the form, and use this code:

```
TextBox1.AppendLine(Date.GetShortTime(Date.Now))
D = #10/31/2022 14:30:10.222#
TextBox1.AppendLine(D.ShortTime)
TextBox1.Append(D.Hour + ":")
TextBox1.Append(D.Minute)
```

Press F5 to run the program. The textbox will show the two short times in your culture, then it will show the second time in the short form with a custom format that uses the English (United States) culture. This is how it looks on my PC with the Arabic(Egypt) culture:



⌚ Date.GetTicks(date) As Double

Gets the total ticks in the given date or duration. In fact the date variable is just a numeric variable that carries the ticks of the date or the duration!

Note that one second contains 10 million ticks!

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method from a date variable as an extension property with the name "Ticks".

Returns:

An integer number that represents the total ticks.

Examples:

```
Ts1 = #+0.0:0:1#
Ts2 = #+0.0:0:0.5#
```

```

Print(Date.GetTicks(#12/31/2022#))
Print(Ts1.Ticks)
Print(Ts2.Ticks)

Sub Print(ticks)
    v = ticks / 1000000
    TextWindow.WriteLine(v + " million ticks")
EndSub

```

The text window will show these results:

```

638080416000 million ticks
10 million ticks
5 million ticks

```

Date.GetTotalDays(duration) As Double

Gets the total days in the given duration.

Parameter:

- **duration:**

The input duration. You can omit this argument if you call this method from a date variable as an extension property with the name "TotalDays".

Returns:

A number that represents the total days in the duration, which can contain a fraction of a day to represent the hours and other time parts if exist.

Examples:

```

D1 = #+00:00:01#
D2 = #-23:10#
D3 = #+1.00:00#

```

```
TextWindow.WriteLine({  
    Date.GetTotalDays(#-1000:10:30:12#),  
    D1.TotalDays,  
    D2.TotalDays,  
    D3.TotalDays  
})
```

The text window will show these results:

```
-1000.43763888889  
0.0000115740740740741  
-0.965277777777778  
1
```

⌚Date.GetTotalHours(duration) As Double

Gets the total hours in the given duration.

Parameter:

- **duration:**

The input duration. . You can omit this argument if you call this method from a date variable as an extension property with the name "TotalHours".

Returns:

A number that represents the total hours in the duration, which includes the hours part + the days part converted to hours. It can also contain a fraction of an hour to represent the minutes and other time parts if exist.

Examples:

```
D1 = #+00:00:01#  
D2 = #-23:10#
```

```

D3 = #+1.00:00#
TextWindow.WriteLine({
    Date.GetTotalHours(#-1000:10:30:12#),
    D1.TotalHours,
    D2.TotalHours,
    D3.TotalHours
})

```

The text window will show these results:

```

-24010.5033333333
0.000277777777777778
-23.16666666666667
24

```

Date.GetTotalMilliseconds(duration) As Double

Gets the total milliseconds in the given duration.

Parameter:

- **duration:**

The input duration. You can omit this argument if you call this method from a date variable as an extension property with the name "TotalMilliseconds".

Returns:

The total milliseconds in the duration, which is the result of dividing the whole duration ticks by 10000.

Examples:

```

D1 = #+00:00:01#
D2 = #-23:10#
D3 = #+1.00:00#

```

```
TextWindow.WriteLine({  
    Date.TotalMilliseconds(#-1000:10:30:12#),  
    D1.TotalMilliseconds,  
    D2.TotalMilliseconds,  
    D3.TotalMilliseconds  
})
```

The text window will show these results:

```
-86437812000  
1000  
-83400000  
86400000
```

⌚ Date.GetTotalMinutes(duration) As Double

Gets the total minutes in the given duration.

Parameter:

- **duration:**

The input duration. You can omit this argument if you call this method from a date variable as an extension property with the name "TotalMinutes".

Returns:

A number that represents the total minutes in the duration, which includes the minutes part + the days and hours parts converted to minutes. It can also contain a fraction of a minute to represent the seconds and milliseconds if exist.

Examples:

```
D1 = #+00:00:01#  
D2 = #-23:10#
```

```
D3 = #+00:30#
TextWindow.WriteLine({
    Date.GetTotalMinutes(#-1000:10:30:12#),
    D1.TotalMinutes,
    D2.TotalMinutes,
    D3.TotalMinutes
})
```

The text window will show these results:

```
-1440630.2
0.01666666666666667
-1390
30
```

⌚Date.GetTotalMonths(duration) As Double

Gets the approximate total months in the given duration.

Parameter:

- **duration:**

The input duration. You can omit this argument if you call this method from a date variable as an extension property with the name "TotalMonths".

Returns:

The approximate total months in the duration, assuming that there are 12 months in the year, and each year contains 365.2425 days. The resultant number can contain fractions to represent days and other time parts if exist.

Examples:

```
D1 = #+00:00:01#
D2 = #-23:10#
D3 = #+31.00:00#
TextWindow.WriteLine({
    Date.GetTotalMonths(#-1000:10:30:12#),
    D1.TotalMonths,
    D2.TotalMonths,
    D3.TotalMonths
})
```

The text window will show these results:

```
-32.8685877253023
0.000000380257053768347
-0.0317134382842802
1.01848049281314
```

⌚ Date.GetTotalSeconds(duration) As Double

Gets the total seconds in the given duration.

Parameter:

- **duration:**

The input duration. You can omit this argument if you call this method from a date variable as an extension property with the name "TotalSeconds".

Returns:

A number that represents the total seconds in the duration, which includes the seconds part + the days, hours and minutes parts converted to minutes. It can also contain a fraction of a second to represent the milliseconds part if exists.

Examples:

```
D1 = #+00:00:01#
D2 = #-23:10#
D3 = #+00:30#
TextWindow.WriteLine({
    Date.GetTotalSeconds(#-1000:10:30:12#),
    D1.TotalSeconds,
    D2.TotalSeconds,
    D3.TotalSeconds
})
```

The text window will show these results:

```
-86437812
1
-83400
1800
```

⌚ Date.GetTotalYears(duration) As Double

Gets the approximate total years in the given duration.

Parameter:

- **duration:**

The input duration. You can omit this argument if you call this method from a date variable as an extension property with the name "TotalYears".

Returns:

The approximate total years in the duration, assuming that each year contains 365.2425 days. The resultant number can contain fractions to represent months, days and other time parts if exist.

Examples:

```
D1 = #+00:00:01#
D2 = #-23:10#
D3 = #+366.00:00#
TextWindow.WriteLine({
    Date.GetTotalYears(#-1000:10:30:12#),
    D1.TotalYears,
    D2.TotalYears,
    D3.TotalYears
})
```

The text window will show these results:

```
-2.73904897710852
0.000000031688087814029
-0.00264278652369001
1.00205338809035
```

⌚ Date.GetYear(date) As Double

Gets the year of the given date.

Parameter:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

Returns:

A number representing the year part of the date, or 0 if the input is a duration.

Examples:

```
D1 = #10/31/2022 1:7:0#
D2 = #-100.03:30#
```

```
TextWindow.WriteLine({  
    Date.GetYear(Date.Now) = Clock.Year, ' True  
    D1.Year, ' 2022  
    D2.Year ' 0  
})
```

Date.Hour As Double

Gets the hour(s) part of the date (or duration).

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetHour](#) method. See more notes and examples there.

Date.LongDate As String

Gets the long form of the date. The long date contains the month name instead of its number.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetLongDate](#) method. See more notes and examples there.

Date.LongDateAndTime As String

Gets the short date and time representation.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetLongDateAndTime](#) method. See more notes and examples there.

Date.LongTime As String

Gets the full time part (including seconds and milliseconds) of the date.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetLongTime](#) method. See more notes and examples there.

Date.Millisecond As Double

Gets the millisecond(s) part of the date (or duration).

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetMillisecond](#) method. See more notes and examples there.

Date.Minute As Double

Gets the minute(s) part of the date (or duration).

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetMinute](#) method. See more notes and examples there.

Date.Month As Double

Gets the month of the date.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetMonth](#) method. See more notes and examples there.

Date.MonthName As String

Gets the local name of the month of the date.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetMonthName](#) method. See more notes and examples there.

Date.Negate(duration) As Date

Negatives the given duration.

Parameter:

- **duration:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

Returns:

- The negative value of the duration if it is positive.
- The positive value of the duration if it is negative.
- An empty string if the input is a date, because it can't be negated.

Examples:

```
D1 = #+10:15:00#
D2 = #-100.03:30#
TextWindow.WriteLine({
    Date.Negate(D1),
    D2.Negate()
})
```

The text window will show these results:

```
-10:15:00  
100.03:30:00
```

Date.Now As Date

Returns the current date and time as defined by the user system.

Date.Second As Double

Gets the second(s) part of the date (or duration).

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetSecond](#) method. See more notes and examples there.

Date.ShortDate As String

Gets the short form of the date, like 1/1/2020.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetShortDate](#) method. See more notes and examples there.

Date.ShortTime As String

Gets the short time part of the date. The time will include hours and minutes and AM or PM, but not seconds and milliseconds.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetShortTime](#) method. See more notes and examples there.

Date.Subtract(date, value) As Date

Creates a new date by subtracting the given value from the given date.

Parameters:

- **date:**

The input date or duration. You can omit this argument if you call this method as an extension method from a date variable.

- **value:**

The date or duration you want to subtract.

Returns:

A new date carrying the result. The input date will not be affected.

Note that:

- You can use the - operator directly to subtract dates.
- Subtracting two durations will return a duration.
- Subtracting two dates will return a duration.
- Subtracting a duration from a date will return a date, but the result will be invalid if the returned date is negative.
- Subtracting a date from a duration will return a meaningless value.

Examples:

D1 = #20 Feb 2023#

D2 = #+1.2:0#

D3 = #-12:00#

D4 = #+3.3:30#

```
TextWindow.WriteLine({
    Date.Subtract(D1, #1/1/2023#),
    D1 - #1/1/2023#,
    Date.Subtract(D1, D2),
    D1 - D2,
    D1.Subtract(D3),
    D1 - D3,
    D3.Subtract(D2),
    D3 - D2,
    D4.Subtract(D2),
    D4 - D2,
    D4.Subtract(D3),
    D4 - D3
})
```

The text window will show these results:

```
50.00:00:00
50.00:00:00
02/18/2023 10:00:00 PM
02/18/2023 10:00:00 PM
02/20/2023 12:00:00 PM
02/20/2023 12:00:00 PM
-1.14:00:00
-1.14:00:00
2.01:30:00
2.01:30:00
3.15:30:00
3.15:30:00
```

Date.Ticks As Double

Gets the total ticks in the date or duration.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetTicks](#) method. See more notes and examples there.

Date.TicksToDate(ticks) As Date

Creates a new date from the given ticks value. Note that one second contains 10 million ticks.

Parameter:

- **ticks:**

The total ticks of the date, which must be a positive number.

Note that any date variable is actually carrying the ticks of the date, so, you can send a date directly as an argument, which will make this method do nothing but returning the same date!

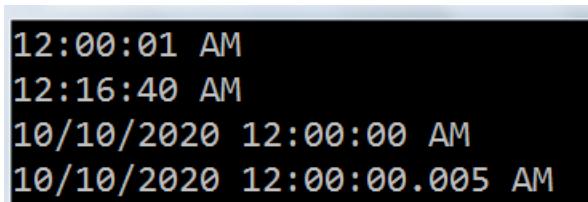
Returns:

A new date.

Examples:

```
TextWindow.WriteLine({  
    Date.TicksToDate(10000000),  
    Date.TicksToDate(10000000000),  
    Date.TicksToDate(#10 Oct 2020#),  
    Date.TicksToDate(#10 Oct 2020# + 50000)  
})
```

The text window will show these results:



```
12:00:01 AM  
12:16:40 AM  
10/10/2020 12:00:00 AM  
10/10/2020 12:00:00.005 AM
```

Date.TicksToDuration(ticks) As Date

Creates a new TimeSpan from the given ticks value. Note that one second contains 10 million ticks.

Parameter:

- **ticks:**

The total ticks of the duration, which can be positive or negative.

Note that any duration variable is actually carrying its ticks, so, you can use a duration directly as an argument, which will make this method do nothing but returning the same duration!

Returns:

A new duration.

Examples:

```
TextWindow.WriteLine({  
    Date.TicksToDuration(-10000000),  
    Date.TicksToDuration(10000000000),  
    Date.TicksToDuration(#+1.12:45#),  
    Date.TicksToDuration(#+1.12:45# + 50000)  
})
```

The text window will show these results:

```
-00:00:01  
00:16:40  
1.12:45:00  
1.12:45:00.005
```

Date.ToCulture(date, cultureName) As String

Gets the short date and short time string representation of the given date formatted with the given culture.

Parameters:

- **date:**

The input date. You can omit this argument if you call this method as an extension method from a date variable.

- **cultureName:**

The culture name used to format the date, like :

- "en-US" for English (United States) culture.
- "fr-Fr" for French (France) culture.
- "ar-EG" for Arabic (Egypt) culture.
- "ar-SA" for Arabic (Saudi Arabia) culture.

For the complete list of supported cultures, see the "Language tag" column in the [list of language/region names supported by Windows](#), but I think you can recognize the pattern from the above examples.

You can send empty string "" if you want to use the local culture of the user's system.

Returns:

A string represent the date in the given culture.

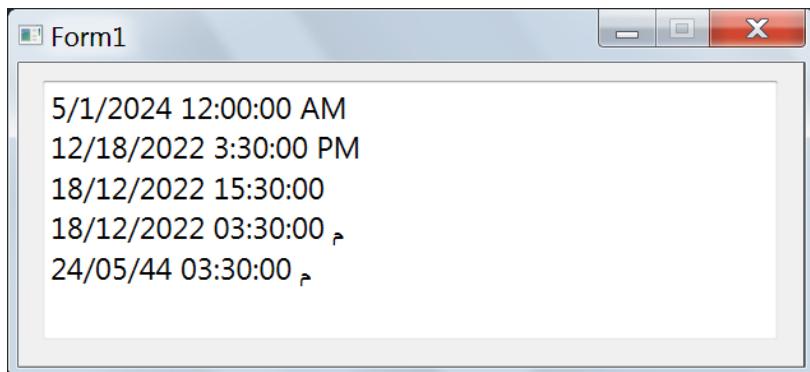
Examples:

Add a TextBox on the form, and use this code:

```
D = #12/18/22 15:30:00#
```

```
TextBox1.AppendLines({
    Date.ToCulture("1/5/2024", "en-US"),
    D.ToCulture("en-US"),
    D.ToCulture("fr-FR"),
    D.ToCulture("ar-EG"),
    D.ToCulture("ar-SA")
})
```

Press F5 to run the program. The textbox will show these results:



⌚ **Date.TotalDays As Double**

Gets the total days in the given duration.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetTotalDays](#) method. See more notes and examples there.

⌚ **Date.TotalHours As Double**

Gets the total hours in the given duration.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetTotalHours](#) method. See more notes and examples there.

Date.TotalMilliseconds As Double

Gets the total milliseconds in the given duration.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetTotalMilliseconds](#) method. See more notes and examples there.

Date.TotalMinutes As Double

Gets the total minutes in the given duration.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetTotalMinutes](#) method. See more notes and examples there.

Date.TotalMonths As Double

Gets the approximate total months in the given duration.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetTotalMonths](#) method. See more notes and examples there.

Date.TotalSeconds As Double

Gets the total seconds in the given duration.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetTotalSeconds](#) method. See more notes and examples there.

Date.TotalYears As Double

Gets the approximate total years in the given duration.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetTotalYears](#) method. See more notes and examples there.

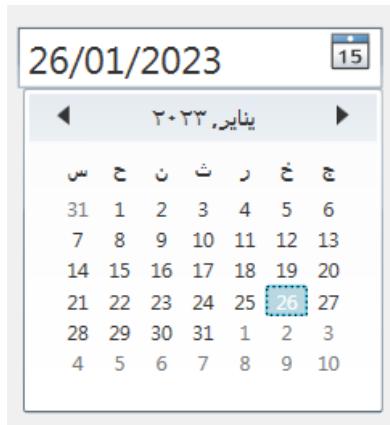
Date.Year As Double

Gets the year or the date.

Note that this is an extension property that can be accessed from a date variable, and its corresponding method is the [Date.GetYear](#) method. See more notes and examples there.

The DatePicker Type

Represents a DatePicker control, that allows the user to enter a date or pick it from the drop down calendar.



You can use the form designer to add a date picker to the form by dragging it from the toolbox.

It is also possible to use the [Form.AddDatePicker](#) method to create a new date picker and add it to the form at runtime.

Note that the DatePicker control inherits all the properties, methods and events of the [Control](#) type.

Besides, the DatePicker control has some new members that are listed below:

DatePicker.OnSelection

This event is fired when the selected date is changed.

Note that this is the default event for the DatePicker control, so, you can double-click any DatePicker on the form designer (say DatePicker1 for example), to get this event handler written for you in the code editor:

```
Sub DatePicker1_OnSelection()
```

```
EndSub
```

For more info, read about [handling control events](#).

Example:

Add a DatePicker and a TextBox on the form, and double-click the date picker to add its OnSelection event handler. Use this handler to make the textbox show the difference between the selected date and today, like this:

```
Sub DatePicker1_OnSelection()
    TextBox1.Text = DatePicker1.SelectedDate - Date.Now
EndSub
```

Press F5 to run the project, and choose some dates from the date picker. The textbox will show the positive and negative durations between now and those dates, like "09:32:52.5419105". You may not want to have the milliseconds part, so, you can use the [Text.IndexOf](#) method to search for the dot, then use the [Text.GetSubText](#) method to get the text before it. Or you can just get rid of the milliseconds part of the duration like this:

```
Sub DatePicker1_OnSelection()
    d = DatePicker1.SelectedDate - Date.Now
    TextBox1.Text = Date.ChangeMillisecond(d, 0)
EndSub
```

This will the textbox show results like "09:32:52"

DatePicker.SelectedDate As Date

Gets or sets the date that is selected by the DatePicker.

So, you can use this method to set the initial date selected in the date picker, and to read the date that is selected by the user, as we did in the last example.

The DemoLib Type

This is just a demo sample library created by VB.NET for sVB.
The source code of this library exists in the DemoLib project in the samples folder, and the DemoLib project shows you how to use this library.

Note that you can delete the DemoLib.dll and DempLib.xml from the sVB\bin\Lib folder if you done trying this demo library.

The DemoLib type has these members:

DemoLib.Decrease(delta) As Double

Decreases the value by the given delta.

Parameter:

- **delta:**

The amount of decrement.

Returns:

The new value.

DemoLib.Increase(delta) As Double

Increases the value by the given delta.

Parameter:

- **delta:**

The amount of increment.

Returns:

The new value.

DemoLib.IsPositive As Boolean

Returns True if the Value is positive

DemoLib.ToString() As String

Returns the Value as a string.

DemoLib.Value As Double

Gets or sets the current value.

The Desktop Type

This class provides methods to interact with the desktop of the user's PC.

The Desktop type has these members:

Desktop.FontNames As Array

Gets an array containing the font names defined on the user's system.

Example:

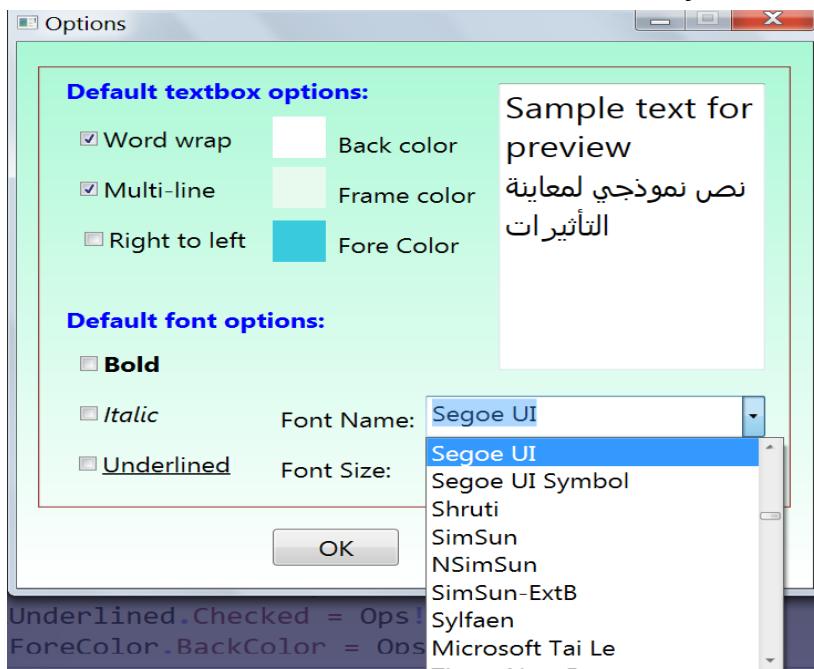
You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the Global.sb file to load the font names to the FontNames array:

FontNames = Desktop.FontNames

which is used in the FrmOps form to populate the CmbFont combo box with the font names:

CmbFont.AddItem(Global.FontNames)

This is how the fonts combo box looks on my PC:



Desktop.Height As Double

Gets the screen height of the primary desktop.

Desktop.SetWallPaper(fileOrUrl)

Sets the specified picture as the desktop's wallpaper.

Parameter:

- **fileOrUrl:**

The filename the picture. This file could be a local file or a network file or even an Internet URL.

Desktop.ShowFontDialog(font) As Array

Displays the font dialog to allow the user to choose font name, size and other font properties.

Parameter:

- **font:**

The initial font to show its properties in the dialog.

Returns:

An empty string "" if the user canceled the operation, otherwise an array containing the font properties under the keys Name, Size, Bold, Italic, Underlined and Color.

Example:

Add a TextBox on the form, and use this code to allow the user to change its font properties:

```
Font = Desktop.ShowFontDialog(TextBox1.Font)
If Font <> "" Then
    TextBox1.Font = Font
EndIf
```

Note that you can do the same job just by using the following single line of code:

```
TextBox1.ChooseFont()
```

Desktop.Width As Double

Gets the screen width of the primary desktop.

The DialogResults Type

Contains the names of common dialog results, which indicates the button that the user clicked on the dialog.

The DialogResults type has these properties:

Abort:

Returns the string "Abort".

Cancel:

Returns the string "Cancel".

Continue:

Returns the string "Continue".

Ignore:

Returns the string "Ignore".

No:

Returns the string "No".

OK:

Returns the string "OK".

Retry:

Returns the string "Retry".

Try:

Returns the string "Try".

Yes:

Returns the string "Yes".

Examples:

You can see this type in action in the "Show Dialogs" project in the samples folder. It is used in the OK button in the

FrmlInput form to set the value of the [Form.DialogResult](#) property:

```
Sub BtnOK_OnClick()
    Global.InputText = TxtInput.Text
    Me.DialogResult = DialogResult.OK
    Me.Close()
EndSub
```

It is also used in the Yes and No buttons in the FrmMsg form to set the value of the [Form.DialogResult](#) property:

```
Sub BtnYes_OnClick()
    Me.DialogResult = DialogResult.Yes
    Me.Close()
EndSub

Sub BtnNo_OnClick()
    Me.DialogResult = DialogResult.No
    Me.Close()
EndSub
```

It is also used in the "Show Msg" button to test the returning dialog result:

```
Sub BtnMsg_OnClick()
    result = Global MsgBox(
        "Confirm",
        "Do you want to save the changes?"
    )

    If result = DialogResult.Yes Then
        TextBox1.Text = "User accepted to save."
    ElseIf result = DialogResult.No Then
        TextBox1.Text = "User refused to save."
    Else
        TextBox1.Text = "User canceled operation."
    EndIf
EndSub
```

The Dialogs Type

Contains methods to show message box and input box dialogs. Note that this type is created as an external library (which is found in the "Bin\Lib\Dialogs" folder in the sVB directory). This library is created using sVB, and its source code exists in the "Dialog" project in the samples folder, so you can modify it and add more dialog forms to fit your needs.

For more information see: [Using sVB to create a code library](#).

The Dialogs type has these members:

Dialogs.InputBox(message) As String

Shows an input box dialog to allow the user to enter some text.

Parameter:

- **message:**

The message to show to the user.

Returns:

The text the user entered, or an empty string if the user cancelled the dialog.

Example:

This code uses an infinite loop to keep asking for the user's age until he enters a valid age or cancel:

```

While True
    Num = Dialogs.InputBox("What's your age?")
    If Num = "" Then
        Me.ShowMessage("You canceled!", "Sorry!")
        ExitLoop
    ElseIf Num.IsNumeric = False Then
        Me.ShowMessage(
            "please enter a valid number", "Error")
    ElseIf Num <= 0 Then
        Me.ShowMessage(
            "Age must be > 0. You are born!", "Error")
    Else
        Me.ShowMessage(
            Num + " years old! Well done!", "OK")
        ExitLoop
    EndIf
Wend

```

Dialogs.MsgBox(strTitle, strMessage) As DialogResult

Shows a message box dialog with Yes, No and Cancel buttons.

Parameters:

- **strTitle:**

The title to show on the message title bar.

- **strMessage:**

The message to show to the user.

Returns:

One of the [DialogResults](#) values that indicates the button that the user clicked.

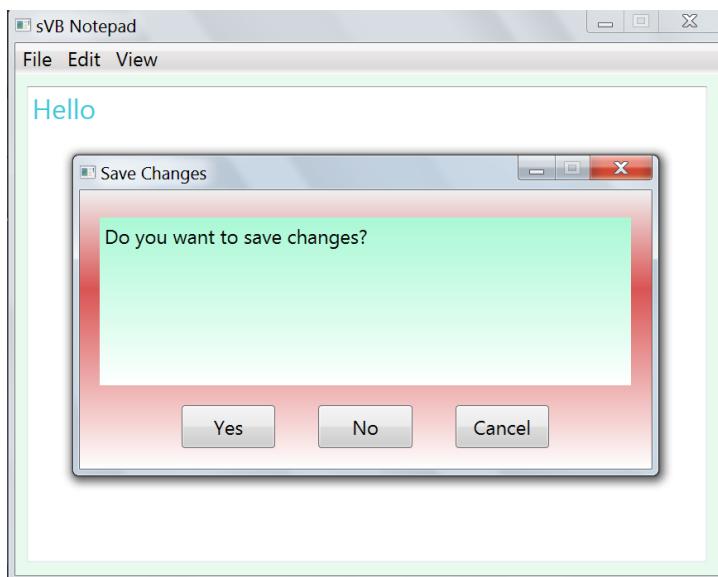
Example:

You can see this method in action in the "sVB Notepad" project in the samples folder. It is used the AskToSave function in the frmMain form to ask the user to save the changes before he closes the opened document:

```
Function AskToSave()
    If IsModified Then
        result = Dialogs(MsgBox(
            "Save Changes",
            "Do you want to save changes?")
    )

    If result = DialogResult.Yes Then
        If Save() = False Then
            Return False
        EndIf
    ElseIf result = DialogResult.Cancel Then
        Return False
    EndIf
EndIf

Return True
EndFunction
```



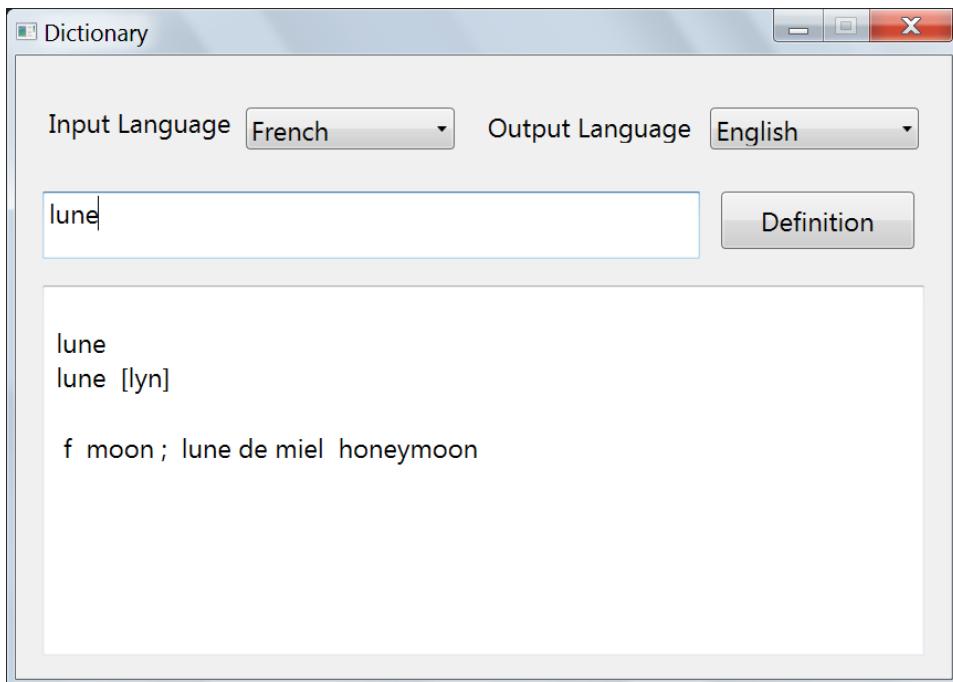
The Dictionary Type

Provides access to an online dictionary service, to allow you to get the definition of an English word in one of the following languages:

English, French, German, Italian, Japanese, Korean, SimplifiedChinese, Spanish and TraditionalChinese.

You can also get the English definition of French, German, Italian, Japanese, Korean, SimplifiedChinese, Spanish or TraditionalChinese words.

You can see the Dictionary type in action in the "Dictionary" project in the samples folder.



The Dictionary type has these members:

Dictionary.GetDefinition(word) As String

Gets the definition of a word in English. The same as GetDefinitionEnglishToEnglish.

Parameter:

- **word:**

The English word to define.

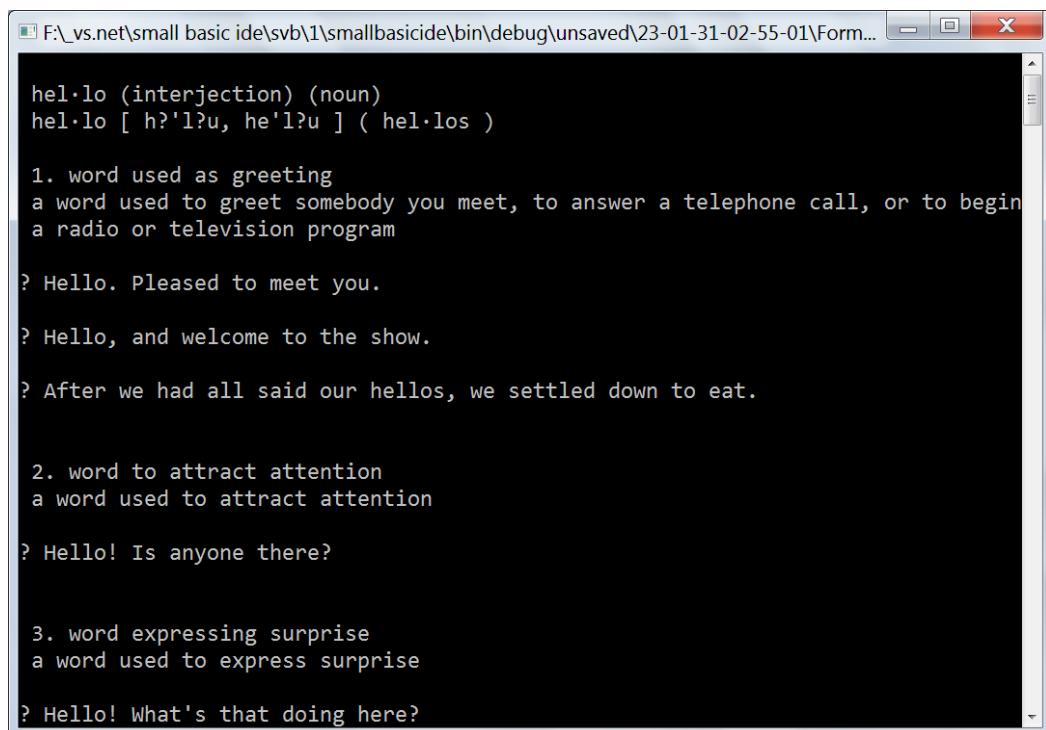
Returns:

The English definition(s) of the specified word.

Example:

```
TextWindow.WriteLine(  
    Dictionary.GetDefinition("Hello"))
```

The TextWindow will show the following results:



A screenshot of a Windows Text Window titled "F:_vs.net\small basic ide\svb\1\smallbasicide\bin\debug\unsaved\23-01-31-02-55-01\Form...". The window displays the definition of the word "Hello".

```
hel.lo (interjection) (noun)
hel.lo [ h?'l?u, he'l?u ] ( hel·los )

1. word used as greeting
a word used to greet somebody you meet, to answer a telephone call, or to begin
a radio or television program

? Hello. Pleased to meet you.

? Hello, and welcome to the show.

? After we had all said our hellos, we settled down to eat.

2. word to attract attention
a word used to attract attention

? Hello! Is anyone there?

3. word expressing surprise
a word used to express surprise

? Hello! What's that doing here?
```

Dictionary.GetDefinitionEnglishToEnglish(word) As String

Gets the definition of a word, English to English.

Parameter:

- **word:**

The English word to define.

Returns:

The English definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionEnglishToFrench(word) As String

Gets the definition of a word, English to French.

Parameter:

- **word:**

The English word to define.

Returns:

The French definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionEnglishToGerman(word) As String

Gets the definition of a word, English to German.

Parameter:

- **word:**

The English word to define.

Returns:

The German definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionEnglishToItalian(word) As String

Gets the definition of a word, English to Italian.

Parameter:

- **word:**

The English word to define.

Returns:

The Italian definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionEnglishToJapanese(word) As String

Gets the definition of a word, English to Japanese.

Parameter:

- **word:**

The English word to define.

Returns:

The Japanese definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionEnglishToKorean(word) As String

Gets the definition of a word, English to Korean.

Parameter:

- **word:**

The English word to define.

Returns:

The Korean definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionEnglishToSimplifiedChinese(word) As String

Gets the definition of a word, English to Simplified Chinese.

Parameter:

- **word:**

The English word to define.

Returns:

The Simplified Chinese definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionEnglishToSpanish(word) As String

Gets the definition of a word, English to Spanish.

Parameter:

- **word:**

The English word to define.

Returns:

The Spanish definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionEnglishToTraditionalChinese(word) As String

Gets the definition of a word, English to Traditional Chinese.

Parameter:

- **word:**

The English word to define.

Returns:

The Traditional Chinese definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionFrenchToEnglish(word) As String

Gets the definition of a word, French to English.

Parameter:

- **word:**

The French word to define.

Returns:

The English definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionGermanToEnglish(word) As String

Gets the definition of a word, German to English.

Parameter:

- **word:**

The German word to define.

Returns:

The English definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionItalianToEnglish(word) As String

Gets the definition of a word, Italian to English.

Parameter:

- **word:**

The Italian word to define.

Returns:

The English definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionJapaneseToEnglish(word) As String

Gets the definition of a word, Japanese to English.

Parameter:

- **word:**

The Japanese word to define.

Returns:

The English definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionKoreanToEnglish(word) As String

Gets the definition of a word, Korean to English.

Parameter:

- **word:**

The Korean word to define.

Returns:

The English definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionSimplifiedChineseToEnglish(word) As String

Gets the definition of a word, Simplified Chinese to English.

Parameter:

- **word:**

The Simplified Chinese word to define.

Returns:

The English definition(s) of the specified word.

Example:

You can see this method in action in the BtnDefinition_OnClick Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionSpanishToEnglish(word) As String

Gets the definition of a word, Spanish to English.

Parameter:

- **word:**

The Spanish word to define.

Returns:

The English definition(s) of the specified word.

Example:

You can see this method in action in the `BtnDefinition_OnClick` Event handler in the "Dictionary" project in the samples folder.

Dictionary.GetDefinitionTraditionalChineseToEnglish(word) As String

Gets the definition of a word, Traditional Chinese to English.

Parameter:

- **word:**

The Traditional Chinese word to define.

Returns:

The English definition(s) of the specified word.

Example:

You can see this method in action in the `BtnDefinition_OnClick` Event handler in the "Dictionary" project in the samples folder.

The Evaluator Type

The evaluator type allows you to evaluate mathematical expressions at runtime:

- Use the Expression Property to set the mathematical expression. The expression can contain only one input variable named X.
- Use the Evaluate method to calculate the value of the expression at the given value of x.

The Event type has these members:

Evaluator.Expression As Boolean

Gets or sets the mathematical expression that you want to evaluate. Use these rules to construct the mathematical expression:

1. The expression is case-insensitive.
2. Use X (or x) as the input variable, like: " $x^2 + x + 1$ ". The Evaluate method receives the value of x, and use it to calculate the value of the expression. You can call the Evaluate method as many times as you need to evaluate the expression at different values of x.
3. The expression can contain the following arithmetic operators:

Operator	Discretion	Example
$^$	Raises a number to the power of another	$x^2 + x^{0.5}$
*	Multiplies two numbers	$x * 4$
/	Divides two numbers	$1 / x$
Mod	Divides two numbers but	$x \text{ mod } 2$

%	returns the remainder	$x \% 2$
+	Adds two numbers	$x + 1$
-	Subtracts two numbers	$-x - 1$

4. The operators will be evaluated in the same order you see in the above table, unless you use parentheses to change that order. For example, $2^2 * 3$ will return 12, but $2^(2*3)$ will return 64.
 5. You can use the E and Pi constants from the [Math](#) type. For example: " $E^x + \sin(x * \text{Pi} / 180)$ "
 6. You can also use the methods of the [Math](#) type. For example: " $(2 + 3 * x) / \text{Sin}(\text{GetRadians}(x))$ ".
 7. For simplicity, you can use some abbreviations of some of these functions. For example: " $(2 + 3 * x) / \text{Sin}(\text{Rad}(x))$ ".
- The following table shows you the available abbreviations:

Abbreviation	For	Example
Rnd	GetRandomNumber	rnd(x)
Rand		rand(x + 1)
Random		Random(x)
GetRnd		getRnd(x)
GetRand		getRand(x + 1)
GetRandom		getRandom(x)
ASin	ArcSin	asin(x)
ACos	ArcCos	acos(x)
ATan	ArcTan	atan(x)
Pow	Power	pow(10, 2)
Ln	NaturalLog	Ln(x)
Sqrt	Squareroot	sqrt(x)
Round	Round2	round(x, 3)

Deg	GetDegrees	$\text{deg}(\text{ArcSin}(x))$
Degrees		$\text{degrees}(\text{atan}(x))$
Rad	GetRadians	$\text{Sin}(\text{rad}(x))$
Radians		$\text{cos}(\text{radians}(x))$

Evaluator.Evaluate(x) As Double

Calculates the value of the mathematical expression set to the Expression property, with the x input variable substituted with the given value.

Parameters:

- **x:**

The value to substitute the x input variable with.

Returns:

The value of the mathematical expression at the given x.

Note:

The value of the [Math.UseRadianAngles](#) property affects how the Sin, ArcSin, Cos, ArcCos, Tan and ArcTan functions are evaluated.

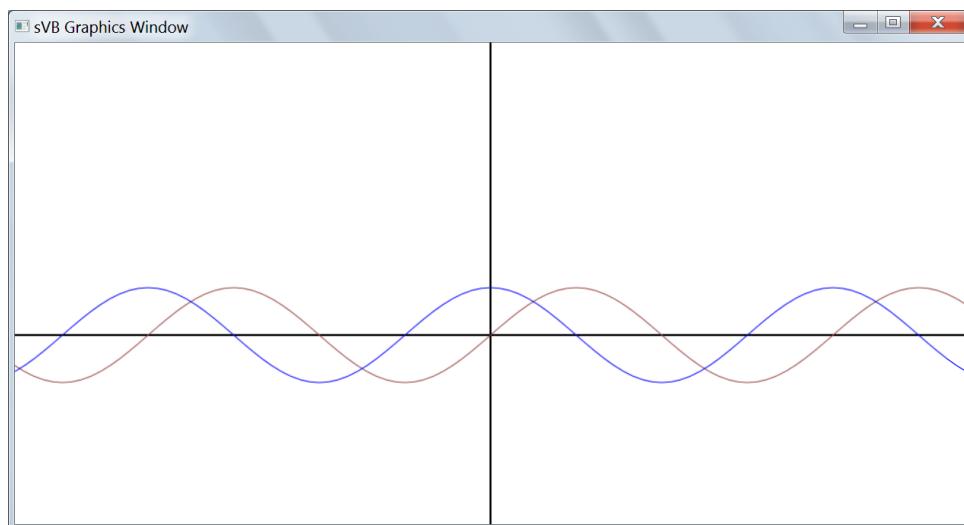
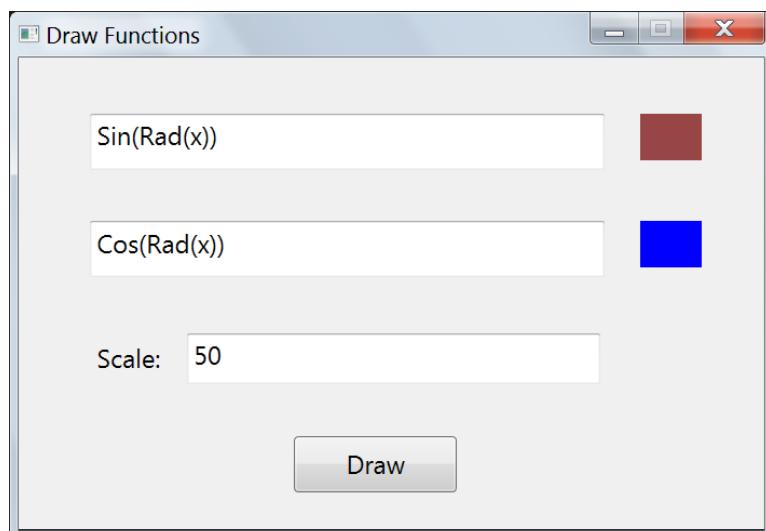
Examples:

You can hard code some expression like this:

```
Evaluator.Expression = "(x^2 + 1) * (-x + 5) / 4"
TextWindow.WriteLine({
    Evaluator.Evaluate(-1),
    Evaluator.Evaluate(0),
    Evaluator.Evaluate(37)
)
```

```
3  
1.25  
-10960
```

Or you can allow the user to provide any expression he wants, so you can evaluate it at some values. You can use this to create an application that draws functions provided by the user. This is exactly what the "Draw Functions" project in the samples folder does, so please take a look at.



Evaluator.LastErrors As String

Returns a string with details about last errors that happened while parsing and evaluating the current expression.

If the Evaluate function doesn't return the value you expect, you should check the errors to understand why.

The Event Type

Contains info about the last fired event. For more information about available events and how to use them, read the [handling control events](#) section.

The Event type has these members:

Event.Handled As Boolean

If this property is set to true inside an event handler sub, the event will be considered handled and will not be processed by the operating system anymore, which means canceling the event.

Examples:

If you want to cancel writing a key to the textbox, use `Event.Handled = True` inside the OnKeyDown event handler.

You can see this in action in the " Numeric TextBox" project in the samples project, where any pressed key is considered handled except the keys from Keys.D0 to Keys.D9 (which are the numeric digits 0-9). This allows the user to only write digits in the textbox.

```
Sub TextBox1_OnKeyDown
    lastKey = Keyboard.LastKey
    If lastKey < Keys.D0 Or lastKey > Keys.D9 Then
        Event.Handled = True
    EndIf
EndSub
```

Press F5 to run the project, and try to write digits and letters in the two textboxes. The first textbox will only accept digits.

Note that it will also reject the backspace, delete and arrow keys, because the condition we used only allows digits. You can allow those keys by adding more conditions to the If statement to check them (like Or LastKey = Keys.Back), or you can use another way as we do in the " Simple Calculator" project, where we use the OnTextInput event handler to cancel the input if the text input about to be written is not numeric. The OnTextInput event is fired only when there is something to write in the textbox, so the backspace, delete and arrow keys will not fire it, hence they will not be cancelled:

```
Sub TxtNum1_OnTextInput
    text = Event.LastTextInput
    If Text.IsNumeric(text) = False Then
        Sound.PlayBellRing()
        Event.Handled = True
    EndIf
EndSub
```

This works just fine, but we still can't write a decimal point (the dot) nor the minus sign! You can easily fix this by writing more conditions to check these two cases, taking into account that there can be only one decimal point, and one negative sign, and also that the negative sign can only appear at the very beginning of the number. You can see how it is done in the " Simple Calculator 3" project. The logic to check the input is written in the Validate subroutine, which is called from the OnTextInput event handlers for the two textboxes:

```
Sub Validate(numTextBox)
    c = Event.LastTextInput
    value = numTextBox.Text
    If Text.IsNumeric(c) = False Then
        If c = "-" Then
            If Text.StartsWith(value, "-") Then
                Sound.PlayBellRing()
                Event.Handled = True
            Else
                numTextBox.Select(1, 0)
            EndIf
        ElseIf c = "." Then
            If Text.Contains(value, ".") Then
                Sound.PlayBellRing()
                Event.Handled = True
            EndIf
        Else
            Sound.PlayBellRing()
            Event.Handled = True
        EndIf
    EndIf
EndSub
```

Event.IsLeftButtonDown As Boolean

Gets whether or not the left button is pressed.

Event.IsRightButtonDown As Boolean

Gets whether or not the right button is pressed.

Event.LastKey

Gets the last Key pressed on the keyboard. It can be used in the OnKeyDown, OnKeyUp, OnPreviewKeyDown and OnPreviewKeyUp event handlers to check the key that fired the event. Use the [Keys enum](#) members to check the key.

Example:

See the examples section of the [Event.Handled property](#).

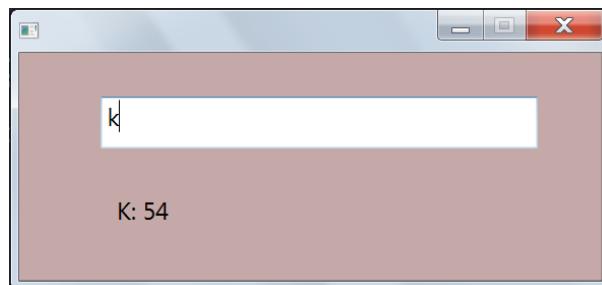
Event.LastKeyName As String

Returns the name of the last key pressed on the keyboard. Use this property to display the key name for the user, but not to check the pressed key, where Event.LastKey is better for this job.

Example:

You can see this property in action in the "Pressed keys" project in the samples folder. It is used in the TextBox1_OnKeyDown event handler to show the name of the pressed key among other info:

```
info = info + Keyboard.LastKeyName + ":"  
      + Keyboard.LastKey
```



Event.LastMouseWheelDirection As Double

Get a value that indicates the last mouse wheel movement direction, where:

- -1 means down.
- 1 means up.

You can use this property in the [Control.OnMouseWheel](#) or the [Form.OnPreviewMouseWheel](#) events handlers to know the direction of the mouse wheel. See examples there.

Event.LastTextInput As String

Returns the last text that was about to be written to the TextBox. This property is meant to be used with the TextBox.OnTextInput event to check the input text before it is written to the textbox.

Example:

See the examples section of the [Event.Handled property](#).

Event.MouseX As Double

Gets or sets the mouse cursor's x co-ordinate. It can be used in the Mouse event handlers to get the horizontal mouse position.

Event.MouseY As Double

Gets or sets the mouse cursor's y co-ordinate. It can be used in the Mouse event handlers to get the vertical mouse position.

Event.SenderControl As Control

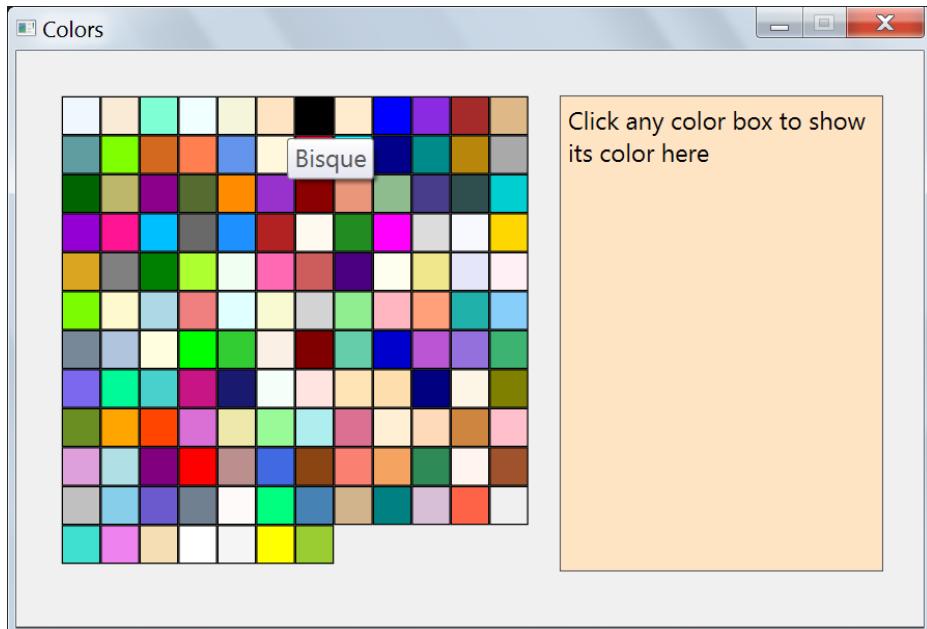
Gets the name of the control that raised the event.

This property is helpful when you use one subroutine to handle many events, to get the control that fired the event.

Example:

You can see this property in action in the "Color labels" project in the samples folder. It is used in the LblColour_OnClick event handler which handles the OnClick event for all the color labels, to get the label that the user clicked, so we can set its color as the back Color of the preview label:

```
Sub LblColour_OnClick()
    senderlabel = Event.SenderControl
    LblPreview.BackColor = senderlabel.BackColor
EndSub
```



The File Type

The File object provides methods to access, read and write information from and to a file on disk. Using this object, it is possible to save and open user date and application settings across multiple sessions of your program.

In the following sections, we will learn about the File type members. Note that most of the file methods return False if the operation failed, and you can test this value then use the File.LastError property to get the error message that describes what the operation failed.

File.AppendContents(filePath, contents) As Boolean

Opens the specified file and appends the given contents to the end of the file, just after the existing contents, without adding a new line or any separator between them.

Parameters:

- **filePath:**

The full path of the file to write to, like "c:\temp\settings.data".

- **contents:**

The contents to append to the end of the file.

When you send an array, its string representation will be written as a single string, without adding any new lines between elements. Use File.AppendLines to insert array elements in separate lines.

Returns:

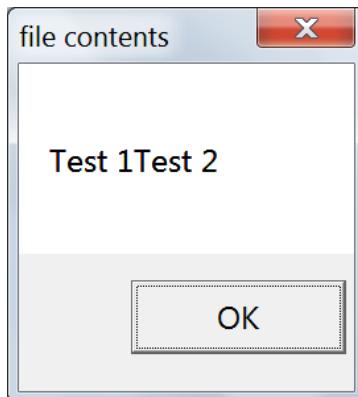
True if the operation was successful, or False otherwise.

Examples:

Add this code at the global area of the form:

```
FilePath = File.GetTemporaryFilePath()  
File.AppendContents(FilePath, "Test 1")  
File.AppendContents(FilePath, "Test 2")  
x = File.ReadContents(FilePath)  
Me.ShowMessage(x, "file contents")
```

When you run the project, you will get this message box:



File.AppendLines(filePath, lines) As Boolean

Opens the specified file and appends the given lines to the end of the file just after the existing contents, then adds a new line.

Parameters:

- **filePath:**

The full path of the file to write to, like "c:\temp\settings.data".

• **lines:**

An array to append its elements to the file, with a new line appended after each element. You can also send a single value (not an array) to append it to the end of the file with a new line appended after it.

Returns:

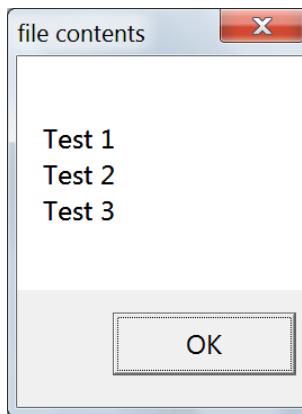
True if the operation was successful, or False otherwise.

Examples:

Add this code at the global area of the form:

```
FilePath = File.GetTemporaryFilePath()
File.AppendLines(FilePath, "Test 1")
File.AppendLines(FilePath, {"Test 2", "Test 3"})
x = File.ReadContents(FilePath)
Me.ShowMessage(x, "file contents")
```

When you run the project, you will get this message box:



File.CopyFile(sourceFilePath, destinationFilePath) As Boolean

Copies the specified source file to the destination file path. If the destination points to a location that doesn't exist, the method will attempt to create it automatically.

Existing files will be overwritten. Check if the destination file exists to avoid overwriting it.

Parameters:

- **sourceFilePath:**

The path of the file that needs to be copied like:
"c:\temp\settings.data".

- **destinationFilePath:**

You can send the destination folder path (without a file name), to copy the source file to. Note the difference between the following two paths:

c:\sVBTest\temp

c:\sVBTest\temp\

The first one is a file path with the name temp without any extension, but the second one is a directory path with the name temp. So, you have to add the path separator character after the directory name, otherwise it will be considered a file name!

You can also send a file path to this parameter, which allows you to copy the file with a new name. This means also you can use this method to rename the file, by creating a copy of the file in the same path but with a new file name, then deleting the source file.

Returns:

True if the operation was successful, or False otherwise.

Examples:

```
' Create a new directory  
File.CreateDirectory("c:\sVBTest")
```

```

' Create a new file
Source = "c:\sVBTTest\1.txt"
File.WriteAllText(Source, "Hello")

' Copy the file to a new directory with the same file name
Destination = "c:\sVBTTest\temp\
If File.CopyFile(Source, Destination) = False Then
    Me.ShowMessage(File.LastError, "Error")
EndIf

' Copy the file to the new directory with a new file name
Destination = "c:\sVBTTest\temp\2.txt"
If File.CopyFile(Source, Destination) = False Then
    Me.ShowMessage(File.LastError, "Error")
EndIf

' Copy the file to the same directory with a new file name
Destination = "c:\sVBTTest\3.txt"
If File.CopyFile(Source, Destination) = False Then
    Me.ShowMessage(File.LastError, "Error")
EndIf

' Delete the file
File.DeleteFile(Source)

```

File.CreateDirectory(directoryPath) As Boolean

Creates the specified directory. If any directory along the given path doesn't exist it will be created.

Parameter:

- **directoryPath:**

The full path of the directory to be created. Don't supply a file name because it will be considered as a directory name (including the extension). for example, sending the path "c:\sVBTTest\Folder1\File1.txt" to this method, will create a directory with the name "File1.txt" !

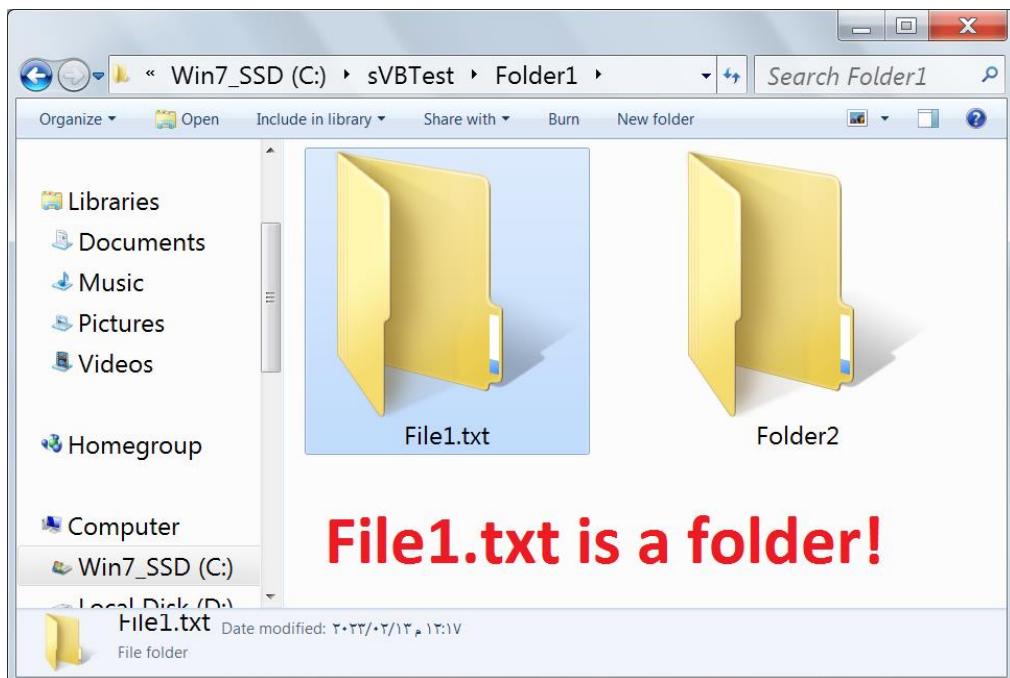
Returns:

True if the operation was successful, or False otherwise.

Examples:

```
DirName = "c:\sVBTest\Folder1\Folder2"  
If File.CreateDirectory(DirName) = False Then  
    Me.ShowMessage(File.LastError, "Error")  
EndIf
```

```
DirName = "c:\sVBTest\Folder1\File1.txt"  
If File.CreateDirectory(DirName) = False Then  
    Me.ShowMessage(File.LastError, "Error")  
EndIf
```



⚙️ **File.DeleteDirectory(directoryPath) As Boolean**

Deletes the specified directory and all its contents. Only the last directory on the path will be deleted, and be careful because all its child folders and files will be deleted too.

Parameter:

- **directoryPath:**

The full path of the directory to be deleted.

Returns:

True if the operation was successful, or False otherwise.

Example:

```
DirName = "c:\sVBTest\Folder1"  
If File.DeleteDirectory(DirName) = False Then  
    Me.ShowMessage(File.LastError, "Error")  
EndIf
```

File.DeleteFile(filePath) As Boolean

Deletes the specified file.

Parameter:

- **filePath:**

The destination location or the file path, like "c:\temp\settings.data".

Returns:

- True if the file is deleted or if it doesn't exist.
- False if the file exists but can't be deleted, for example because access to the file is denied at this moment.

Example:

```
FileName = "c:\sVBTest\file1.txt"  
If File.DeleteFile(FileName) = False Then  
    Me.ShowMessage(File.LastError, "Error")  
EndIf
```

File.GetDirectories(directoryPath) As Array

Gets the paths of all the directories in the specified directory path. Note that this operation is not recursive. This means it gets only the first level of subdirectories (the direct children of the given directory). It will not go deeper to get the children of those subdirectories, and you have to write more code if you want to get them.

Parameter:

- **directoryPath:**

The directory to look for its subdirectories.

Returns:

If the operation was successful, this will return an array containing the subdirectories. Otherwise, it will return an empty string "".

Example:

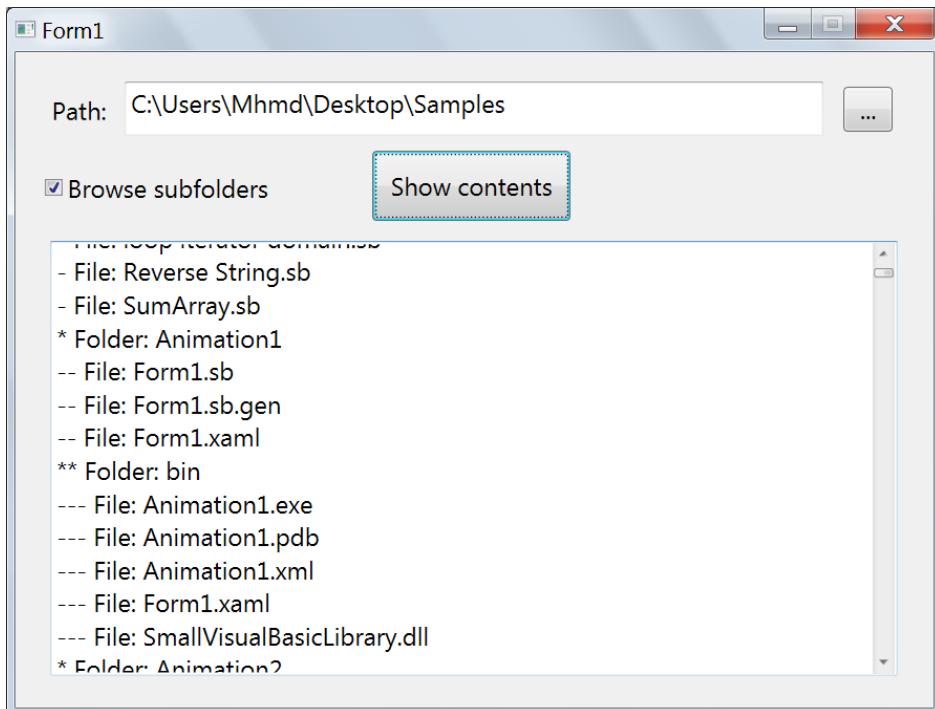
You can see this method in action in the "Show Directory Contents" project in the samples folder. It is used in the ShowContent function to get subdirectories of the directory sent to the function via the strDir parameter. We show those subdirectories names in the textbox, and if the ChkRecursive checkbox is checked by the user, we call the ShowContent function again to show the subdirectories for each of those subdirectories. This recursive call will end up showing all the directory tree in the textbox, and you should be careful not to apply this on the windows drive, because it contains thousands of directories and will take some time to show them. Instead, you may show the contents of the sVB folder.

This is a part of this function:

```

len = strDir.Length
ForEach strSubDir In File.GetDirectories(strDir)
    TxtContents.Append(indent + " Folder: ")
    dirName = strSubDir.SubTextToEnd(len + 2)
    TxtContents.AppendLine(dirName)
    If ChkRecursive.Checked Then
        ShowContent(strSubDir, level + 1)
    EndIf
Next

```



File.GetFiles(directoryPath) As Array

Gets the paths of all the files in the specified directory path.

Parameter:

- **directoryPath:**

The directory to look for its files.

Returns:

If the operation was successful, this will return the files as an array. Otherwise, it will return an empty string "".

Example:

You can see this method in action in the "Show Directory Contents" project in the samples folder. It is used in the ShowContent function to get files of the directory sent to the function via the strDir parameter, then we show those file names in the textbox.

This is a part of this function:

```
ForEach strFile In File.GetFiles(strDir)
    TxtContents.Append(indent + " File: ")
    fileName = strFile.SubTextToEnd(strDir.Length + 2)
    TxtContents.AppendLine(fileName)
```

[Next](#)

File.GetSettingsFilePath() As String

Gets the full path of the settings file for this program. The settings file name is based on the program's name with the extension .settings, and is present in the same location as the program. For example, if you have a program named myApp.exe in the path "C:\sample\MyApp\bin", this method will return the file name: "C:\sample\MyApp\bin\MyApp.settings".

You can use this settings file to save value related to your program before it is closed, so that you can be able to read them again from the settings file when it is opened again.

Returns:

The full path of the settings file specific for this program.

Example:

You can see this method in action in the "Show Directory Contents" project in the samples folder. It is used in the LoadSettings function in the Global.sb file, to load the default settings of the program:

```
_file = File.GetSettingsFilePath()  
settings = File.ReadLines(_file)
```

File.GetTemporaryFilePath() As String

Creates a new unique file name in the windows temporary directory and returns the full file path. This name consist of random letters and numbers, and you will get a different name in each call of this method.

Sometimes you want to store data to a file and read it back while your application is running, but you don't care about this data after the application is closed. This is when you use a temporary file to put such data, and you don't need to worry about deleting the file, because Windows delete temporary file after a period of time. If you want to keep your data, create a file in a normal directory, or use the [GetSettingsFilePath](#) method to get the path of the application's settings file.

Returns:

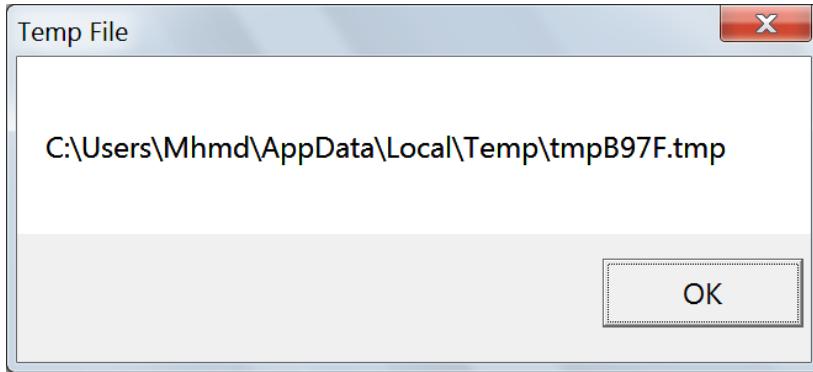
The full file path of the temporary file.

Example:

Add this code in the global area of the form:

```
tmpFile = File.GetTemporaryFilePath()  
Me.ShowMessage(tmpFile, "Temp File")
```

When I ran the above code on my PC, I got this:



⌚ **File.InsertLines(filePath, lineNumber, lines) As Boolean**

Opens the specified file and inserts the given lines at the specified line number.

This operation will not overwrite any existing content at the specified line. It will just insert the new lines before it.

Warning:

Don't use this method to write lines in the same file using a loop, otherwise it will have a bad impact on your application performance, because this operation involves copying all the file lines to a temp file to insert the given content at the given line number, which will be repeated in each call to this method is the loop!

In such case, add all the lines to one array first, call this method to insert it for once.

Parameters:

- **filePath:**

The full path of the file to read from, like "c:\temp\settings.data".

- **lineNumber:**

The line number of the text to insert. If the line number doesn't exist, the lines will be appended to the end of the file.

- **lines:**

An array to insert its items into the file, with a new line added after each item.

Returns:

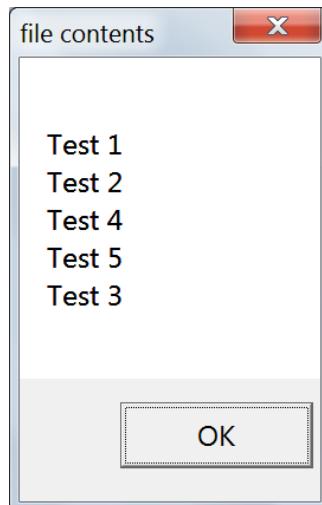
True if the operation was successful, or False otherwise.

Example:

Add this code at the global area of the form:

```
FilePath = File.GetTemporaryFilePath()
File.InsertLines(FilePath, 1, "Test 1")
File.InsertLines(FilePath, 2, {"Test 2", "Test 3"})
File.InsertLines(FilePath, 3, {"Test 4", "Test 5"})
x = File.ReadContents(FilePath)
Me.ShowMessage(x, "file contents")
```

When you run the project, you will get this message box:



File.LastError As String

Gets or sets the last file operation error message. This property is useful for finding out whey some method fails to execute.

Notes:

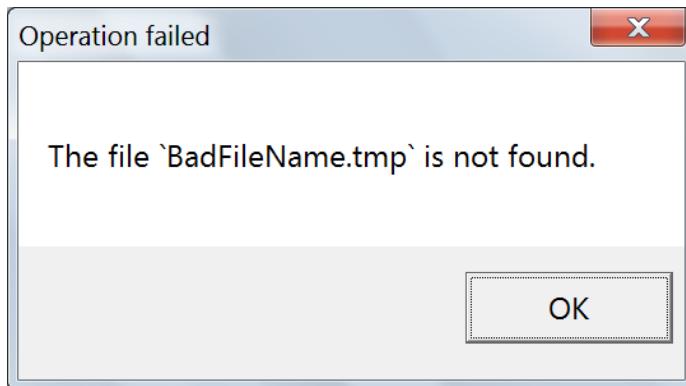
Most file methods return False when they fail, so, you can check the method return value, and show the error message to the user. On the other hand, a few methods like reading methods returns am empty string "" when they fail, but this is not a sufficient indication, because when you read from an empty file you will successfully get an empty string! So, your reliable failure indication in all cases is to test that the LastError property is not empty (<> "").

Example:

Add this code at the global area of the form:

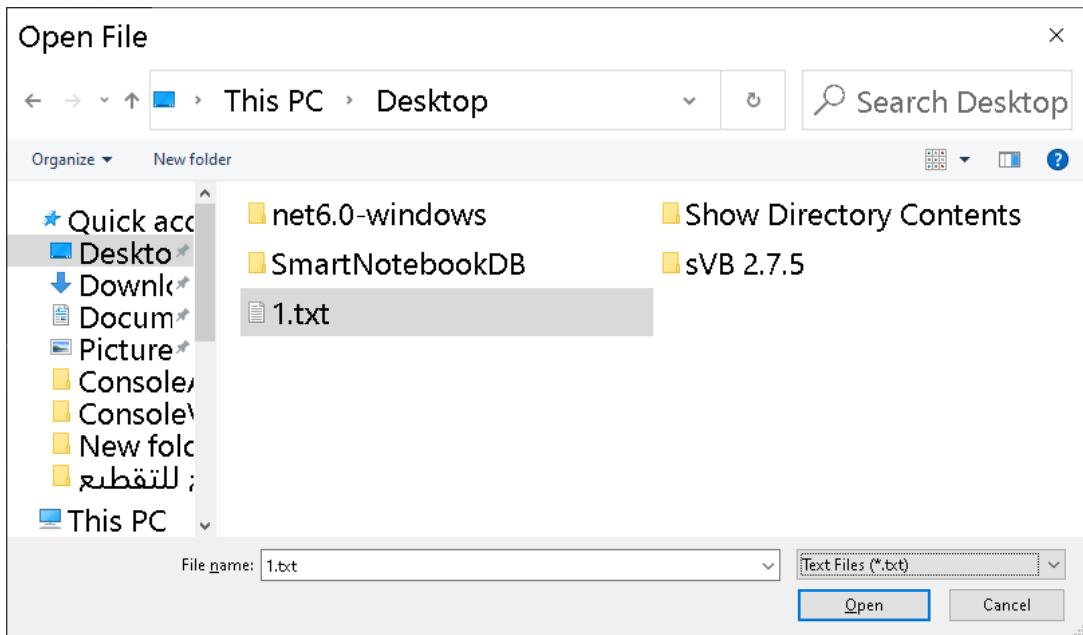
```
X = File.ReadContents("BadFileName.tmp")
If File.LastError <> "" Then
    Me.ShowMessage(File.LastError, "File Error")
EndIf
```

When you run the project, you will get this message box:



File.OpenFileDialog(extFilters) As String

Shows the open file dialog, to allow the user to select a file from his PC file system.



Parameter:

- **extFilters:**

Extension filters specify the file types you want to allow the dialog to view. Each filter consist of a description text and a list of file extensions.

You can use a standard string filter, where a | is used to separate between the filter parts, and ; is used to separate between file extensions, like:

"Text Files|*.txt|Images|*.bmp;*.jpg;*.gif|Doc|*.doc"

Or you can use an array of extension filters, with each extension filter itself is an array, where:

- the first Item describes the file type which ill be displayed in the dialog window in the file types dropdown list.

- and the next items contain one or more extension.
- If you will open only one file type category like images, you can send the extension filter directly as a one dimension array like: {"Images", "bmp", "jpg", "gif"}.
- otherwise, send an array of extension filters like:
 {{"Text Files", ".txt"}, {"Images", "bmp", "jpg", "gif"}}
- If you will open only a single extension, you can just use it as a single string like: "doc".
- You can mix the above rules, such as:

```
{
    {"Text Files", ".txt"},
    {"Images", "bmp", "jpg", "gif"},
    "doc"
}
```
- If you want to show all files, use "" or "*" as the extension, like: {"All Files", "*"}, or just use "*".

Returns:

The file name that the user selected, or an empty string "" if he canceled the operation.

Example:

Add a textbox and a button on the form, double-click the button and write this code in the Button1_OnClick event handler:

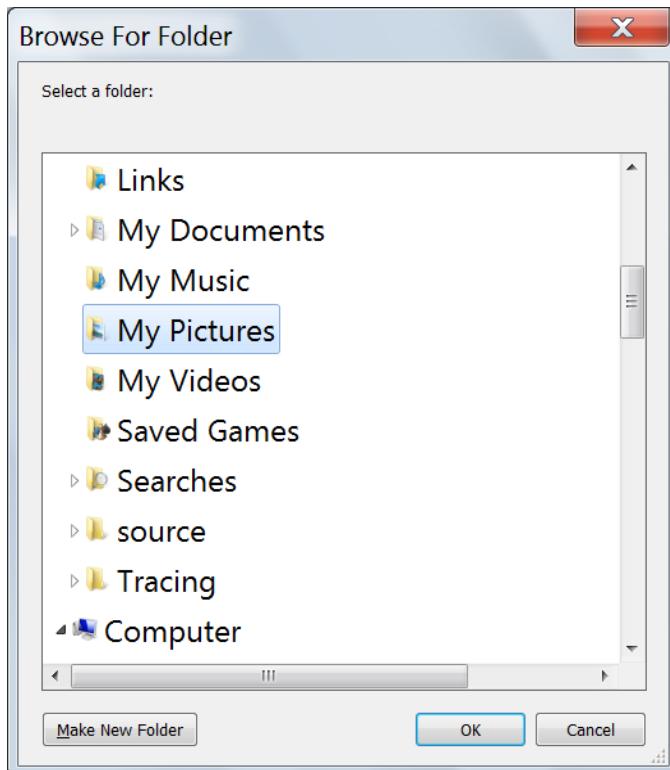
```
_file = File.OpenFileDialog(
    {"Text Files", ".txt"},
    {"Images", "bmp", "jpg", "gif"},
    "doc"
)
```

```
If _file <> "" Then  
    TextBox1.Text = File.ReadAllText(_file)  
EndIf
```

Press F5 to run the project, then click the button to show the open file dialog. The Text files filter will be selected initially, which allows you to select .txt files. You can browse to any folder on your system and change the extension filter to select images or docs. After you select a file, click the Open button to close the dialog and show its contents in the textbox.

File.OpenFileDialog(initialFolder) As String

Shows the folder browser dialog, to allow the user to select a folder from his PC file system.



Parameter:

- **initialFolder:**

The path of the folder you want to select in the folder dialog when it is opened. This parameter is important because the user can't paste any folder path into this dialog to start browsing from, and it will be frustrating to start browsing from the desktop every time, so, this parameter allows you to suggest the initial folder to start browsing from. If you send empty string "", sVB will try to help the user, by selecting the folder path that the user copied to the windows clipboard (if any), or the folder that the user selected in the dialog in the last time (if any).

Example:

You can see this method in action in the "Show Directory Contents" project in the samples folder. It is used in the BtnBrowse_OnClick event handler to allow the user to select the folder that he wants to show its files and folders in the textbox:

```
Sub BtnBrowse_OnClick()
    dir = File.OpenFileDialog(TxtPath.Text)
    If dir <> "" Then
        TxtPath.Text = dir
    EndIf
EndSub
```

File.ReadArray(filePath) As Array

Opens the specified file and reads its data as an array.

Parameter:

- **filePath:**

The full path of the file to read, like "c:\temp\settings.data".

Returns:

An array if the specified file content is a valid string representation of an array, or an empty string otherwise.

Example:

```
A = {1, 2, 3, 4}
Path = File.GetTemporaryFilePath()
If File.WriteAllArray(Path, A) Then
    TW.WriteLine(File.ReadAllText(Path))
EndIf
```

```
1
2
3
4
Press any key to close the window...
```

File.ReadContents(filePath) As String

Opens a file and reads the entire file contents. This method will be fast for small files that are less than 1 MB in size, but will start to slow down and will be noticeable for files greater than 10MB.

Parameter:

- **filePath:**

The full path of the file to read, like "c:\temp\settings.data".

Returns:

The entire contents of the file.

Example:

You can see this method in action in the "sVB Notepad" project in the samples folder. It is used in the MenuOpen_OnClick event handler to read the content if the opened file is a text file. This is a part of the code:

```
If fileName = "" Then
    Return
ElseIf Text.EndsWith(fileName.ToLowercase, ".txt") Then
    TxtEditor.Text = File.ReadContents(fileName)
    ops = Global.Ops
Else
    ' .....
EndIf
```

File.ReadLine(filePath, lineNumber) As String

Opens the specified file and reads the contents at the specified line number.

Don't use this method in a loop to read many lines from the same file, because it always starts reading from the first line of the file until it reaches the required line, which can have a heavy impact on performance in loops.

Instead, if the file size is less than 1 mega bytes, you can use the ReadLines method to read all the file lines then walk through it using the loop as you want.

Parameters:

- **filePath:**

The full path of the file to read from, like "c:\temp\settings.data".

- **lineNumber:**

The line number of the text to be read.

Returns:

The text at the specified line if exists, otherwise an empty string "". You should check the File.LastError property to know if the empty string results from an empty line or from a wrong line number or none existing file.

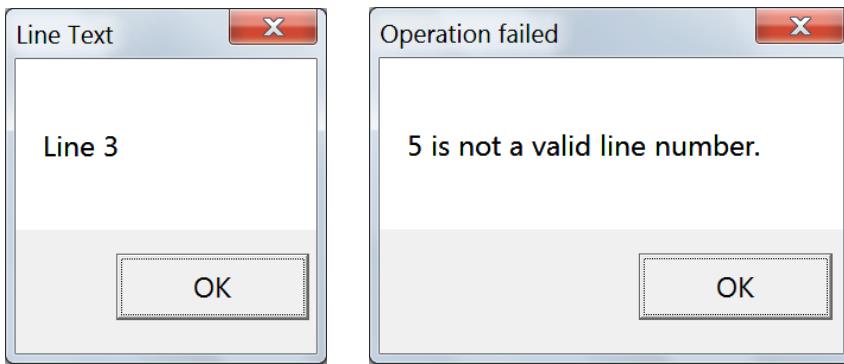
Examples:

When you run this code:

```
_File = "BadFileName.tmp"
File.WriteLine(
    _File,
    {"Line 1 ", "Line 2", "Line 3", "Line 4"}
)
ReadLine(3)
ReadLine(5)

Sub ReadLine(lineNumber)
    x = File.ReadLine(_File, lineNumber)
    If File.LastError = "" Then
        Me.ShowMessage(x, "Line Text")
    Else
        Me.ShowMessage(File.LastError, "Error!")
    EndIf
EndSub
```

You will get these two messages:



File.ReadLine(filePath) As Array

Opens the specified file and reads all its lines.

Parameter:

- **filePath:**

The full path of the file to read from, like "c:\temp\settings.data".

Returns:

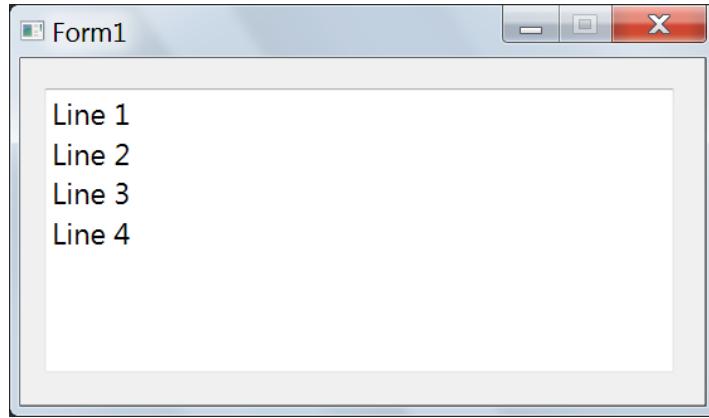
An array containing the lines of the specified file , or an empty string if the file is not found.

Examples:

Add a textbox on the form, then add this code in the global area of the form:

```
_File = "BadFileName.tmp"  
File.WriteAllText(_File, {"Line 1 ", "Line 2",  
"Line 3", "Line 4"})  
TextBox1.AppendLines(File.ReadLines(_File))
```

Press F5 to run the project. The textbox will show this result:



File.SaveFileDialog(fileName, extFilters) As String

Shows the save file dialog, to allow the user to enter the file name and choose a location in his PC file system to save the file to.



Parameters:

- **fileName:**

A suggested name to save the file with. The user can change this name in the dialog. You can use the full path of the file, to suggest the initial directory in the dialog, otherwise, the initial directory will be the last opened one.

- **extFilters:**

Extension filters specify the file types you want to allow the dialog to view. Each filter consist of a description text and a list of file extensions.

You can use a standard string filter, where a | is used to separate between the filter parts, and ; is used to separate between file extensions, like:

"Text Files|*.txt|Images|*.bmp;*.jpg;*.gif|Doc|*.doc"

Or you can use an array of extension filters, with each extension filter itself is an array, where:

- the first Item describes the file type which ill be displayed in the dialog window in the file types dropdown list.
- and the next items contain one or more extension.
- If you will open only one file type category like images, you can send the extension filter directly as a one dimension array like: {"Images", "bmp", "jpg", "gif"}.
- otherwise, send an array of extension filters like:
 {{"Text Files", ".txt"}, {"Images", "bmp", "jpg", "gif"}}
- If you will open only a single extension, you can just use it as a single string like: "doc".
- You can mix the above rules, such as:

```

{
    {"Text Files", ".txt"},
    {"Images", "bmp", "jpg", "gif"},
    "doc"
}

```

- If you want to show all files, use "" or "*" as the extension, like: {"All Files", "*"}, or just use "*".

Returns:

The file name that the user selected, or an empty string "" if he canceled the operation.

Example:

You can see this method in action in the "sVB Notepad" project in the samples folder. It is used in the SaveAs function in the frmMain.sb file to ask the user where to save the file:

```

Function SaveAs()
    fileName = File.SaveFileDialog(
        SavedFile,
        {
            {"Text", "txt"},
            {"Text with format", "twf"}
        }
    )
    If fileName = "" Then
        Return False
    Else
        SavedFile = fileName
        Return Save()
    EndIf
EndFunction

```

File.WriteAllText(filePath, array) As Boolean

Opens a file and replaces its data with the given array.

Parameters:

- **filePath:**

The full path of the file to read, like "c:\temp\settings.data".

- **array:**

The array to write its string representation into the specified file. Actually you can pass a string, a number or a date to this parameter and it will be saved to the file, but this will cause an error if you try to read it later with the ReadArray method.

Returns:

True if the operation was successful, or False otherwise.

Example:

```
A = {1, 2, 3, 4}
Path = File.GetTemporaryFilePath()
If File.WriteAllText(Path, A) Then
    TW.WriteLine(File.ReadAllText(Path))
Else
    TW.WriteLine(File.LastError)
EndIf
```

```
1
2
3
4
Press any key to close the window...
```

File.WriteAllText(filePath, contents) As Boolean

Opens a file and replaces its data with the given contents.

Parameters:

- **filePath:**

The full path of the file, like "c:\temp\settings.data".

- **contents:**

The contents to write into the specified file. You can send an array to write its elements to the file, each element in a line.

Returns:

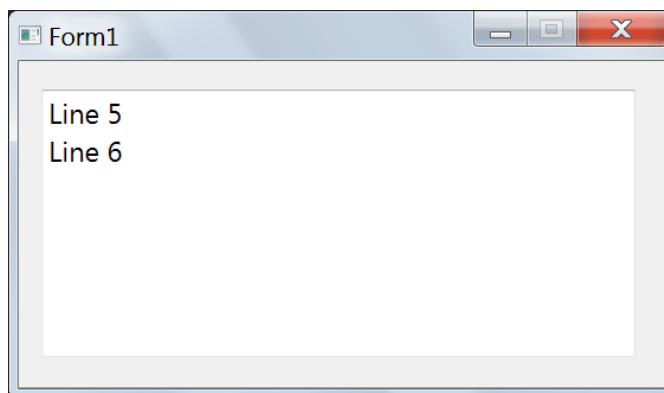
True if the operation was successful, or False otherwise.

Example:

Add a textbox on the form, and write this code in the global area of the form:

```
_File = File.GetTemporaryFilePath()
File.WriteAllText(_File,
    {"Line 1 ", "Line 2", "Line 3", "Line 4"})
TextBox1.AppendLines(File.ReadLines(_File))
File.WriteAllText(_File, {"Line 5 ", "Line 6"})
TextBox1.Text = ""
TextBox1.AppendLines(File.ReadLines(_File))
```

Press F5 to run the project. The textbox will look like this:



File.WriteLine(filePath, lineNumber, lines) As Boolean

Opens the specified file and writes the given lines starting from the specified line number.

This operation will overwrite any existing content at the corresponding file lines.

Don't use this method in a loop, as it will have a bad impact on performance, because it involves copying all file lines to a temp file to write the given content at the given line number, which will be repeated for every line you write using the loop!

Instead, add all lines to an array and write it for once.

Parameters:

- **filePath:**

The full path of the file, like "c:\temp\settings.data".

- **lineNumber:**

The line number to write the first item of the lines array at.

- **lines:**

An array to write its items to the file, each item at a line, starting at the given line number.

Returns:

True if the operation was successful, or False otherwise.

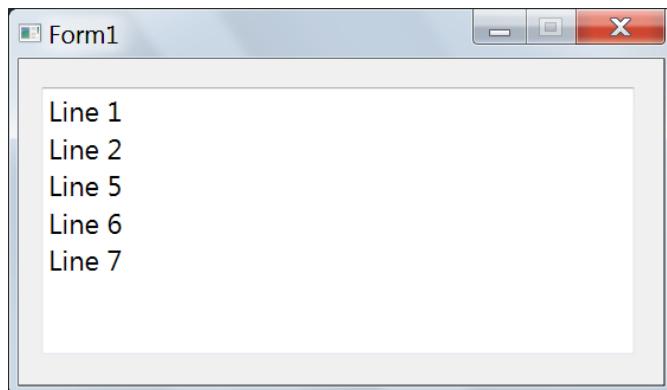
Example:

Add a textbox on the form, and write this code in the global area of the form:

```
_File = File.GetTemporaryFilePath()  
File.WriteLine(_File, 1,  
 {"Line 1 ", "Line 2", "Line 3", "Line 4"})
```

```
File.WriteLine(_File, 3,  
    {"Line 5 ", "Line 6", "Line 7"})  
TextBox1.AppendLines(File.ReadLines(_File))
```

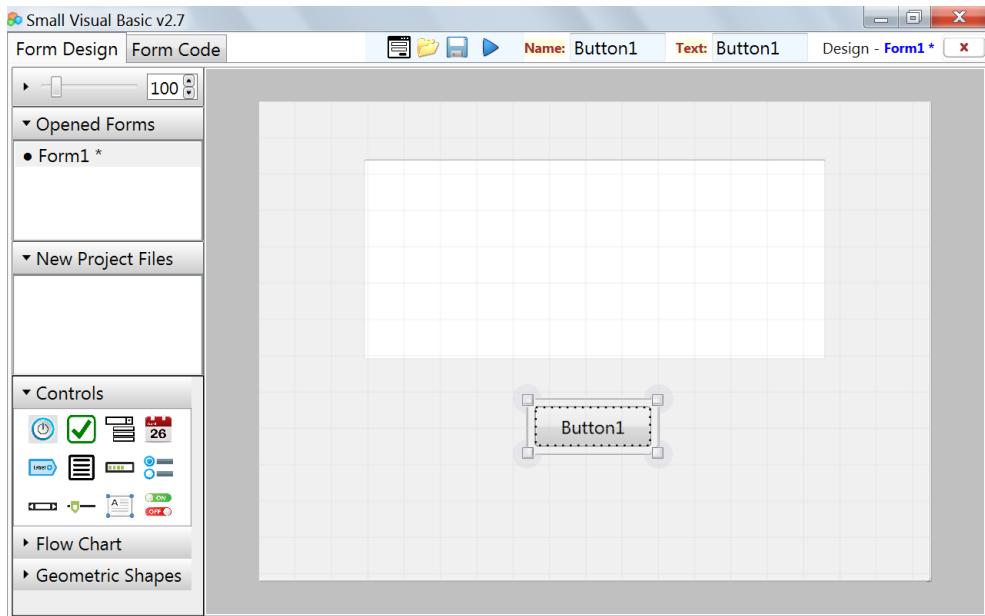
Press F5 to run the project. The textbox will look like this:



The Form Type

Represents the Form control, which is a window that can display other controls like labels, text boxes and buttons which the user interacts with.

It is easier to use the form designer to add a new form and save it to a XAML file.



It is also possible to create a new form at run time, by calling the [Forms.AddForm](#) method. It is also possible to call the [Forms.LoadForm](#) method to load a form from the XAML file that contains its design.

You can display the form by calling the [Form.Show](#) method.

The Form control inherits all the properties, methods and events of the [Control](#) type.

Besides, the Form control has some new members that are listed below:

Me:

You can use "Me" to refer to the current Form, so if you deal with a form named Form1, you can use both the following statements to change its title:

Form1.Text = "My name is Form1"

Me.Text = "I am also Form1"

Form.AddButton(buttonName, left, top, width, height) As Button

Adds a new [Button control](#) to the form at runtime.

It is easier to use the form designer to create controls and adjust their locations and sizes in design time, so, don't use this method unless you really need to.

Parameters:

- **buttonName:**

A unique name for the new Button.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

Returns:

The key of the button in the formula {formName.buttonName} like "form1.button1". sVB can deal with this key as an object

of type [Button](#), so, you can access the Button properties, methods and events via it.

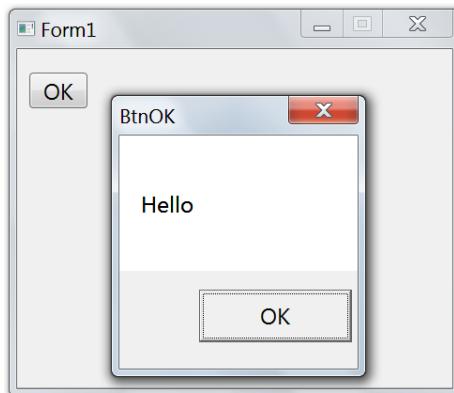
Example:

Add this code to the form global area.

```
BtnOK = Me.AddButton("Button1", 10, 20, 50, 30)  
BtnOK.Text = "OK"  
BtnOK.OnClick = BtnOK_OnClick
```

```
Sub BtnOK_OnClick()  
    Me.ShowMessage("Hello", "BtnOK")  
EndSub
```

Press F5 to run the program. The OK button will be displayed on the form, and when you click it, it will show the "Hello" message box:



Form.AddCheckBox(checkBoxName, left, top, text, checked) As CheckBox

Adds a new [CheckBox control](#) to the form.

It is easier to use the form designer to create controls and adjust their locations and sizes in design time, so, don't use this method unless you really need to.

Parameters:

- **checkBoxName:**

A unique name for the new CheckBox.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **text:**

The text to display on the CheckBox.

- **checked:**

The value to set to the CheckBox.Checked property.

Returns:

The key of the CheckBox in the formula

{formName.checkBoxName} like "form1.checkbox1".

sVB can deal with this key as an object of type [CheckBox](#), so, you can access the CheckBox properties, methods and events via it.

Example:

Add this code to the form global area.

```
' Create a checkbox and handle its OnCheck event:  
CheckBox1 = Me.AddCheckBox("CheckBox1", 10, 50,  
    "Show Label", True)  
CheckBox1.OnCheck = CheckBox1_OnCheck  
  
' Create a label on the left of the checkbox:  
Label1 = Me.AddLabel(  
    "Label1",  
    CheckBox1.Left + CheckBox1.Width + 30,  
    CheckBox1.Top, -1, -1)
```

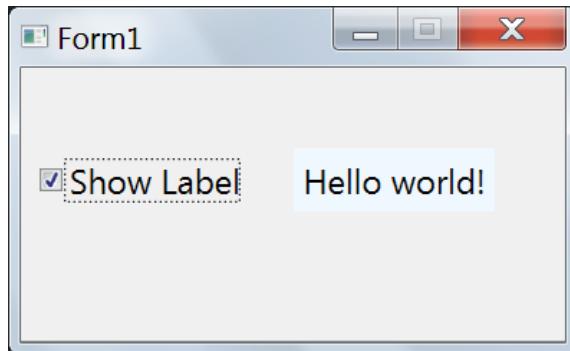
```

' Set the label's properties:
Label1.BackColor = Colors.AliceBlue
Label1.Text = "Hello world!"
' Align the middles of label and checkbox horizontally:
Label1.Top = Label1.Top -
    (Label1.Height - CheckBox1.Height) / 2

' The OnCheck handler hides or shows the label:
Sub CheckBox1_OnCheck()
    Label1.Visible = CheckBox1.Checked
EndSub

```

Press F5 to run the program. The form will display the checkbox and the label. Check and uncheck the checkbox to show and hide the label.



Form.AddComboBox(comboBoxName, left, top, width, height) As ListBox

Adds a new ComboBox control to the form.

It is easier to use the form designer to create controls and adjust their locations and sizes in design time, so, don't use this method unless you really need to.

Parameters:

- **comboBoxName:**

A unique name for the new combo box.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

Returns:

The key of the combo box in the formula
{formName.comboBoxName} like "form1.comboBox1".

sVB can deal with this key as an object of type [ComboBox](#), so,
you can use the ComboBox properties, methods and events
via it.

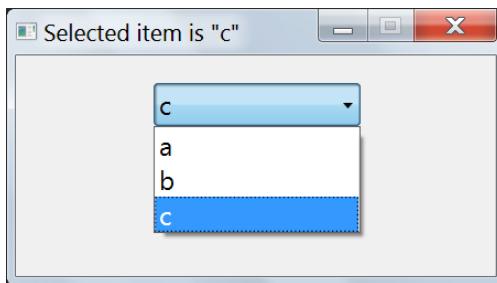
Example:

Add this code to the form global area.

```
ComboBox1 = Me.AddComboBox(  
    "ComboBox1", 100, 20, 150, -1)  
ComboBox1.AddItem({"a", "b", "c"})  
ComboBox1.OnSelection = ComboBox1_Select  
ComboBox1.SelectedIndex = 3  
  
Sub ComboBox1_Select()  
    Me.Text = Text.Format(  
        "Selected item is [1][2][1]",  
        {Chars.Quote, ComboBox1.SelectedItem})  
    )  
EndSub
```

Press F5 to run the program. The form will display the combo
box with the last item selected, and the form title bar will

show the selected item, and will be updated every time you change the selected item.



⚙️ **Form.AddDatePicker(datePickerName, left, top, width, selectedDate) As DatePicker**

Adds a new [DatePicker control](#) to the form.

It is easier to use the form designer to create controls and adjust their locations and sizes in design time, so, don't use this method unless you really need to.

Parameters:

- **datePickerName:**

A unique name for the new DatePicker.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **selectedDate:**

The date that will be selected in the control when it is displayed.

Returns:

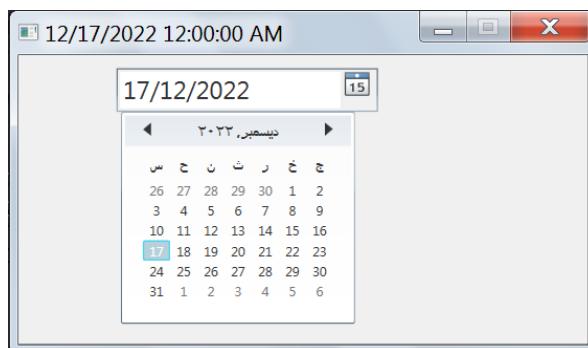
The key of the DatePicker in the formula
{formName.datePikkerName} like "form1.datepicker1".
sVB can deal with this key as an object of type [DatePicker](#), so,
you can access the DatePicker properties, methods and events
via it.

Example:

Add this code to the form global area.

```
DatePicker1 = Me.AddDatePicker(  
    "DatePicker1", 75, 10, 200, #12/18/2022#)  
DatePicker1.OnSelection = DatePicker1_OnSelection  
  
Sub DatePicker1_OnSelection()  
    Me.Text = DatePicker1.SelectedDate  
EndSub
```

Press F5 to run the program. The date 12/18/2022 will be selected in date picker and every time you change the selected date, it will be displayed on the form title bar.



Form.AddLabel(labelName, left, top, width, height) As Label

Adds a new [Label control](#) to the form.

It is easier to use the form designer to create controls and adjust their locations and sizes in design time, so, don't use this method unless you really need to.

Parameters:

- **labelName:**

A unique name for the new Label.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

Returns:

The key of the label in the formula {formName.labelName} like "form1.label1". sVB can deal with this key as an object of type [Label](#), so, you can access the Label properties, methods and events via it.

Example:

Add this code to the form global area.

```
Label1 = Me.AddLabel("Label1", 10, 20, -1, -1)  
Label1.Text = "Hi there!"
```

```

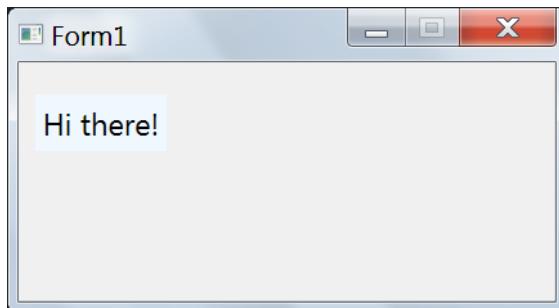
Label1.OnMouseEnter = Label1_OnMouseEnter
Label1.OnMouseLeave = Label1_OnMouseleave

Sub Label1_OnMouseEnter()
    Label1.BackColor = Colors.AliceBlue
EndSub

Sub Label1_OnMouseleave()
    Label1.BackColor = Colors.Transparent
EndSub

```

Press F5 to run the program. The form will display the "Hi there" label, and when the mouse pointer enters its area, its back color will change, where you can confirm that its width and height fits its content (because we used -1 for both). When the mouse pointer leaves the label area, its back color will be transparent again. You can repeat that as many times as you want!



Form.AddListBox(listBoxName, left, top, width, height) As ListBox

Adds a new [ListBox control](#) to the form.

It is easier to use the form designer to create controls and adjust their locations and sizes in design time, so, don't use this method unless you really need to.

Parameters:

- **listBoxName:**

A unique name for the new ListBox.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

Returns:

The key of the list box in the formula

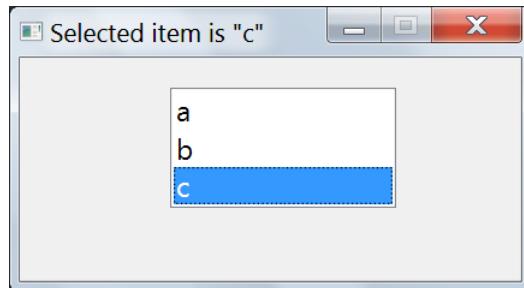
{formName.listBoxName} like "form1.listBox1". sVB can deal with this key as an object of type [ListBox](#), so, you can access the ListBox properties, methods and events via it.

Example:

Add this code to the form global area.

```
ListBox1 = Me.AddListBox(  
    "ListBox1", 100, 20, 150, -1)  
ListBox1.AddItem({"a", "b", "c"})  
ListBox1.OnSelection = ListBox1_Select  
ListBox1.SelectedIndex = 3  
  
Sub ListBox1_Select()  
    Me.Text = Text.Format(  
        "Selected item is [1][2][1]",  
        {Chars.Quote, ListBox1.SelectedItem}  
    )  
EndSub
```

Press F5 to run the program. The form will display the listbox with the last item selected, and the form title bar will show the selected item, and will be updated every time you change the selected item.



Form.AddMainMenu(menuName) As MainMenu

Adds a [MainMenu control](#) to the current form. If there is already a main menu, it will be replaced.

Parameter:

- **menuName:**

The name of the main menu

Returns:

The key of the menu in the formula

{formName.mainMenuName} like "form1.mainMenu1".

sVB can deal with this key as an object of type [MainMenu](#), so, you can access the MainMenu properties, methods and events via it.

Example:

You can see this method in action in the "sVB Notepad" project in the samples folder. It is used in the FrmMain form to create the main menu by code, because the form designer

doesn't support adding menus visually:

```
M = Me.AddMainMenu("mainmenu")
```

Then the main menu variable M is used to add the menu items as you can see in the rest of the code.

Form.AddProgressBar(

**progressBarName, left, top,
width, height, minimum, maximum**

) As ProgressBar

Adds a new [ProgressBar control](#) to the form.

It is easier to use the form designer to create controls and adjust their locations and sizes in design time, so, don't use this method unless you really need to.

Parameters:

- progressBarName:**

A unique name for the new ProgressBar.

- left:**

The X-pos of the control.

- top:**

The Y-pos of the control.

- width:**

The width of the control. Use -1 for auto-width.

- height:**

The height of the control. Use -1 for auto-height.

- minimum:**

The progress minimum value

- **maximum:**

The progress maximum value. Use 0 if the max value is indeterminate. This will display an endlessly moving marquee.

Returns:

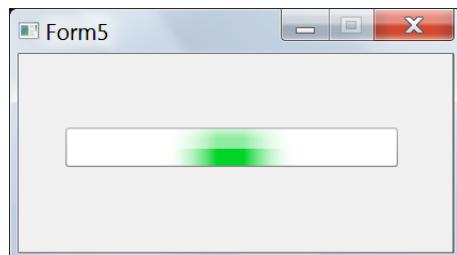
The key of the ProgressBar in the formula {formName.progressBar} like "form1.progressBar1". sVB can deal with this key as an object of type [ProgressBar](#), so, you can access the ProgressBar properties, methods and events via it.

Example:

Add this code to the form global area.

```
ProgressBar1 = Me.AddProgressBar(  
    "ProgressBar1",  
    35, 55, 250, 30,  
    0, 0  
)
```

Press F5 to run the program. The form will display the progress bar with an endlessly moving marquee.



Form.AddRadioButton(radioButtonName, left, top, text, groupName, checked) As RadioButton

Adds a new [RadioButton control](#) to the form.

It is easier to use the form designer to create controls and adjust their locations and sizes in design time, so, don't use this method unless you really need to.

Parameters:

- **radioButtonName:**

A unique name for the new RadioButton.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **text:**

The text to display on the RadioButton

- **groupName:**

The name of the group to add the button to.

- **checked:**

The value to set to the Checked property

Returns:

The key of the RadioButton in the formula

{formName.radioButtonName} like "form1.radioButton1".

sVB can deal with this key as an object of type [RadioButton](#), so, you can access the RadioButton properties, methods and events via it.

Example:

Add this code to the form global area. It creates 2 groups of radio buttons, each contains two buttons:

```
' Radio buttons group1
RdoTop = Me.AddRadioButton(
    "RdoTop", 35, 30, "Top", "group1", False)
RdoBottom = Me.AddRadioButton(
    "RdoBottom", 235, 30, "Bottom", "group1", True)

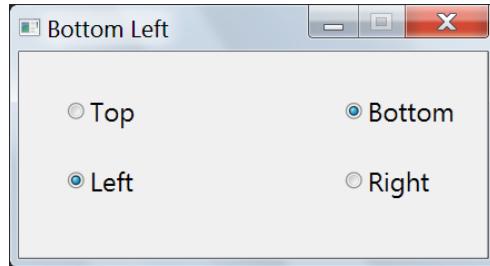
' Radio buttons group2
RdoLeft = Me.AddRadioButton(
    "RdoLeft", 35, 80, "Left", "group2", True)
RdoRight = Me.AddRadioButton(
    "RdoRight", 235, 80, "Right", "group2", False)

' Only one event handler for all radio buttons:
RdoTop.OnCheck = OnCheck
RdoBottom.OnCheck = OnCheck
RdoLeft.OnCheck = OnCheck
RdoRight.OnCheck = OnCheck

' Call the handler once to handle initial radio buttons
' states which are set by the AddRadioButton method
OnCheck()

Sub OnCheck()
    pos = ""
    If RdoTop.Checked Then
        pos = "Top"
    Else
        pos = "Bottom"
    EndIf
    If RdoLeft.Checked Then
        pos = pos + " Left"
    Else
        pos = pos + " Right"
    EndIf
    Me.Text = pos
EndSub
```

Press F5 to run the program. The form will display the 4 radio buttons with the "Bottom" and "Left" radio buttons selected. The form title bar will show the chosen pos, and will be updated every time you change the selected radio buttons.



⚙️ **Form.AddScrollBar(**

**scrollBarName, left, top, width, height,
minimum, maximum, value**

) As ScrollBar

Adds a new [ScrollBar control](#) to the form.

It is easier to use the form designer to create controls and adjust their locations and sizes in design time, so, don't use this method unless you really need to.

Parameters:

- scrollBarName:**

A unique name for the new ScrollBar.

- left:**

The X-pos of the control.

- top:**

The Y-pos of the control.

- width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

- **minimum:**

The scrollbar minimum value

- **maximum:**

The scrollbar maximum value.

- **value:**

The scrollbar current value.

Returns:

The key of the Scrollbar in the formula
{formName.scrollbarName} like "form1.scrollbar1".

sVB can deal with this key as an object of type [Scrollbar](#), so, you can access the Scrollbar properties, methods and events via it.

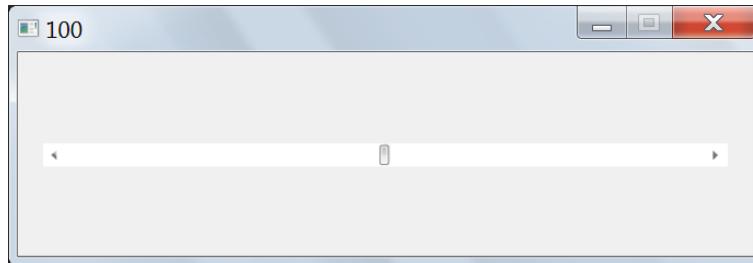
Example:

Add this code to the form global area:

```
ScrollBar1 = Me.AddScrollBar(
    "ScrollBar1",
    260, -175, -1, 500,
    0, 200, 50
)
' Rotate the scroll bar to be horizontal
ScrollBar1.Angle = -90
ScrollBar1.BackColor = Colors.White
ScrollBar1.OnScroll = ScrollBar1_OnScroll
Me.Text = 50

Sub ScrollBar1_OnScroll()
    Me.Text = Math.Round(ScrollBar1.Value)
EndSub
```

Press F5 to run the program. The form will display the scroll bar, and the form title bar will show its value (which is indicated by the thumb position). The form title will be updated every time you change the thumb position of the scroll bar.



⚙️ **Form.AddSlider(sliderName, left, top, width, height, minimum, maximum, value, tickFrequency) As Slider**

Adds a new [Slider control](#) to the form.

It is easier to use the form designer to create controls and adjust their locations and sizes in design time, so, don't use this method unless you really need to.

Parameters:

- **sliderName:**

A unique name for the new Slider.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

- **minimum:**

The slider minimum value

- **maximum:**

The slider maximum value.

- **value:**

The slider current value.

- **tickFrequency:**

The distance between slide ticks.

Returns:

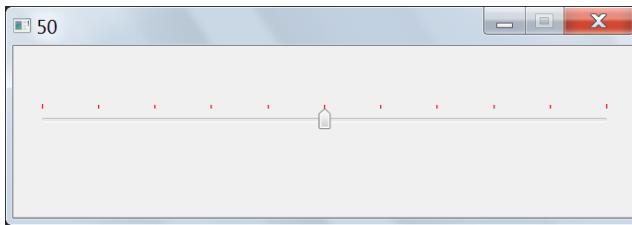
The key of the Slider in the formula {formName.sliderName} like "form1.slider1". sVB can deal with this key as an object of type [Slider](#), so, you can access the Slider properties, methods and events via it.

Example:

Add this code to the form global area:

```
Slider1 = Me.AddSlider()  
    "Slider1",  
    20, 50, 500, 30,  
    0, 100, 50, 10  
)  
  
Slider1.ForeColor = Colors.Red  
Slider1.OnSlide = Slider1_OnSlide  
Me.Text = 50  
  
Sub Slider1_OnSlide()  
    Me.Text = Math.Round(Slider1.Value)  
EndSub
```

Press F5 to run the program. The form will display the slider, and the form title bar will show its value (which is indicated by the thumb position). The form title will be updated every time you slide the thumb on the slider.



⚙️ **Form.AddTextBox(textBoxName, left, top, width, height) As TextBox**

Adds a new [TextBox control](#) to the form.

It is easier to use the form designer to create controls and adjust their locations and sizes in design time, so, don't use this method unless you really need to.

Parameters:

- **textBoxName:**

A unique name for the new TextBox.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

Returns:

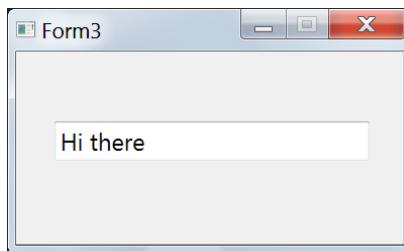
The key of the TextBox in the formula {formName.textBoxName} like "form1.textBox1". sVB can deal with this key as an object of type [TextBox](#), so, you can access the TextBox properties, methods and events via it.

Example:

Add this code to the form global area:

```
TextBox1 = Me.AddTextBox(  
    "TextBox1", 30, 55, 250, -1)  
TextBox1.Text = "Hi there"
```

Press F5 to run the program. The form will display the textbox as shown in the picture:



⌚ [Form.AddTimer\(timerName, interval\) As WinTimer](#)

Adds a new [WinTimer control](#) to the form in runtime.

You can't use the form designer to add timers to the form in design time.

Parameters:

- **timerName:**

A unique name for the new timer.

- **interval:**

The delay time in milliseconds between ticks.

Returns:

The key of the Timer in the formula {formName.timerName} like "form1.timer1". sVB can deal with this key as an object of type [WinTimer](#), so, you can access the WinTimer properties, methods and events via it.

Example:

You can see this method in action in the "Stop Watch" project in the samples folder. It is used to add two timers to the form, one to display the clock on the title bar, and the other to update the stop watch display:

```
ClockTimer = Me.AddTimer("clockTimer", 1000)  
ClockTimer.OnTick = ShowClock  
StopWatch = Me.AddTimer("StopWatch", 10)  
StopWatch.Pause()  
StopWatch.OnTick = ShowStopWatch
```

Form.AddToggleButton(toggleButtonName, left, top, width, height) As ToggleButton

Adds a new [ToggleButton control](#) to the form.

It is easier to use the form designer to create controls and adjust their locations and sizes in design time, so, don't use this method unless you really need to.

Parameters:

- **toggleButtonName:**

A unique name for the new ToggleButton.

- **left:**

The X-pos of the control.

- **top:**

The Y-pos of the control.

- **width:**

The width of the control. Use -1 for auto-width.

- **height:**

The height of the control. Use -1 for auto-height.

Returns:

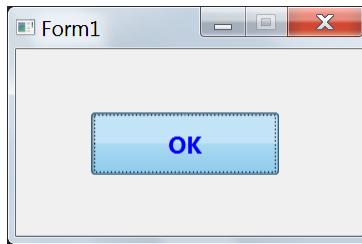
The key of the ToggleButton in the formula
{formName.toggleButtonName} like "form1.toggleButton1".
sVB can deal with this key as an object of type [ToggleButton](#),
so, you can access the ToggleButton properties, methods and
events via it.

Example:

Add this code to the form global area:

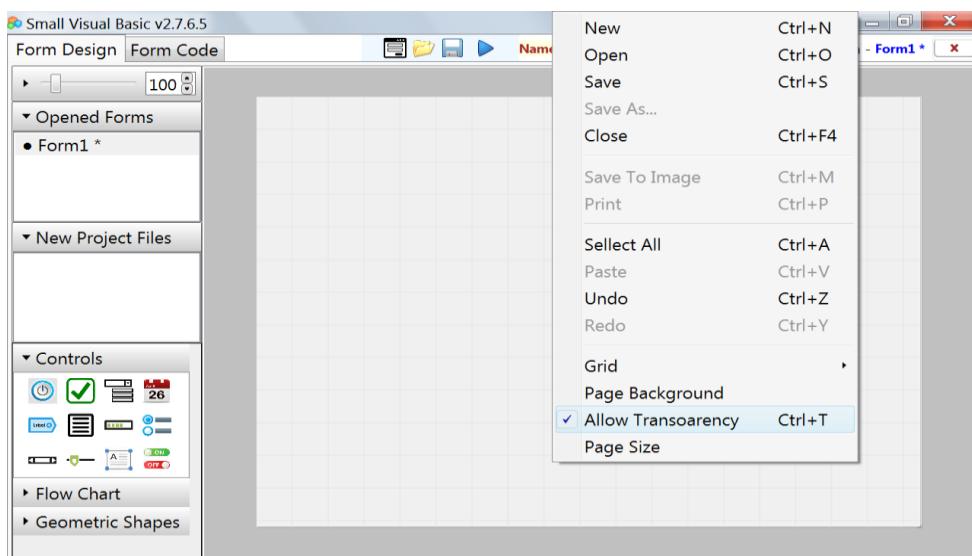
```
ToggleButton1 = Me.AddToggleButton(  
    "ToggleButton1", 60, 50, 150, 50)  
ToggleButton1.Text = "OK"  
ToggleButton1.ForeColor = Colors.Blue  
ToggleButton1.OnCheck = ToggleButton1_OnCheck  
ToggleButton1.Checked = True  
  
Sub ToggleButton1_OnCheck()  
    ToggleButton1.FontBold = ToggleButton1.Checked  
EndSub
```

Press F5 to run the program. The form will display the toggle button, and it will be checked. Click it to toggle its check state, and notice that its font weight changes from Bold to normal and vice versa as its checked state changes.



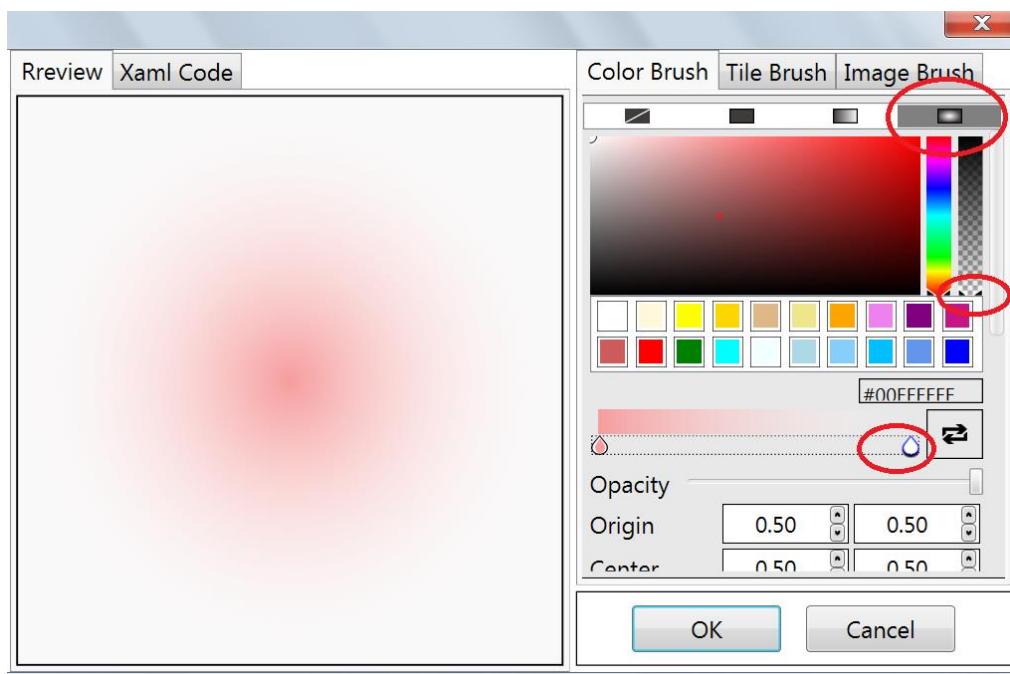
💡 **Form.AllowTransparency()**

Call this method to allow using transparent colors on the form background. You must do this before the form is displayed, which is valid only when you create the form by calling the [Forms.AddForm](#) or [Forms.LoadForm](#) methods, and before you call the Form.Show method. Forms created by the form designer will not be affected by calling the AllowTransparency method, because the form is displayed before executing any code you write in the global area of the form. In such case, you must use the form designer to allow form transparency, by right-clicking the form surface and checking the "Allow Transparency" context menu item. This will add a call to the Form.AllowTransparency method in the code behind generated in the form.sb.gen file, before calling the Form.Show method.

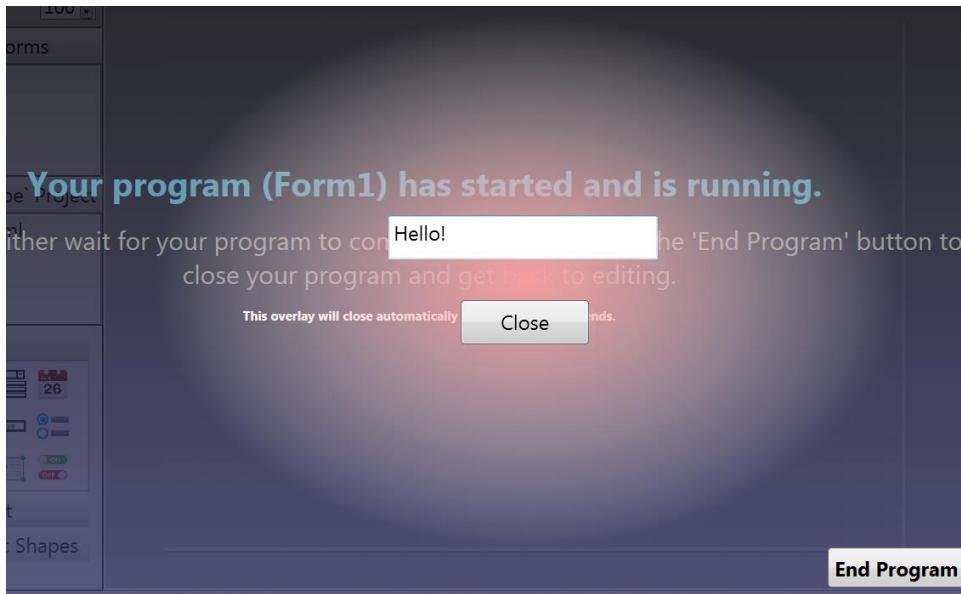


Example:

You can see this method in action in the FormShape project in the samples folder. We used the "Allow Transparency" in form designer to allow the form transparency, then we used the "Page Background" context menu item to change the form background to a radial gradient brush, giving its the outer stop a transparent color to erase the form background surrounding the ellipse:



Press F5 to run this project. You will have a gradient transparent elliptic form, and you can drag it from any point of the ellipse to move it on the screen, but points out of the ellipse don't exist, and clicking out of the ellipse will pass the click to whatever object behind the form, like the window desktop or another window!



Form.ArgsArr As Array

Returns the additional data sent to the form via the second argument of the [Forms.ShowForm](#), [Forms.ShowDialog](#) and [Form.ShowChildForm](#) methods.

Despite its name, you can send a string, a number or a control to this property, not just an array. Always remember that sVB is a dynamic language, and any variable can hold any data type.

Examples:

You can see this method in action in the "Show Dialogs" project in the samples folder. It is used in the FrmMsg.sb form to get the title and message text to be displayed by the message box:

```
Info = Me.ArgsArr  
Me.Text = Info!Title  
LblMsg.Text = Info!Message
```

To pass these two info, we sent a dynamic object to the second argument of the ShowDialog method. You can find this code in the MsgBox function in the Global.sb file:

```
info!Title = strTitle  
info!Message = strMessage  
Return Forms.ShowDialog("FrmMsg", info)
```

ArgsArr is also used in the FrmInput.sb form to get the message text to be displayed by the input box:

```
LblMsg.Text = Me.ArgsArr
```

This is only one info, so, it is just passed as a single string to the second argument of the ShowDialog method. You can find this code in the InputBox function in the Global.sb file:

```
result = Forms.ShowDialog("FrmInput", message)
```

Form.Close()

Closes the form. This will clear any data displayed by its controls, as in fact the form will be totally disposed. You can't call the Form.Show method to show the form after it is closed, unless you re-created a new instance of it first. If you want to keep the form alive but just hide it, you can just set its Form.Visible property to False or call the Form.Hide method to hide it, so you can still be able to show it again.

Note that closing the main form of the project will close the whole application. You can choose the main form by just open any form of the project in sVB and press F5 to run the project, so that form will be the first form displayed when the application starts up.

Form.ContainsControl(controlName) As Boolean

Returns True if the form displays a control with the given name.

Parameter:

- **controlName:**

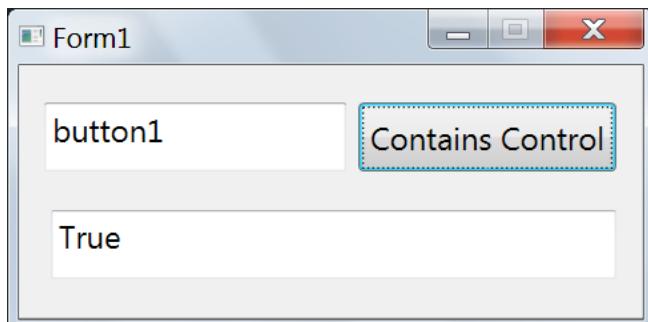
The name of the control to search for. It is case-insensitive.

Returns:

True or False

Example:

You can test this method, by designing a form like this:



Then add the following code in the Button1_Click event handler:

```
TextBox2.Text = Me.ContainsControl(TextBox1.Text)
```

Press F5 to run the project, and test some names, like an empty string, textbox1, button1, and btnTest.

Form.Controls As Array

Returns an array containing the names of all controls displayed on the form.

It is preferred to use this property in a ForEach loop, with a proper the loop iterator variable name that starts or ends with

the word Control, so that you can use it as a control object.

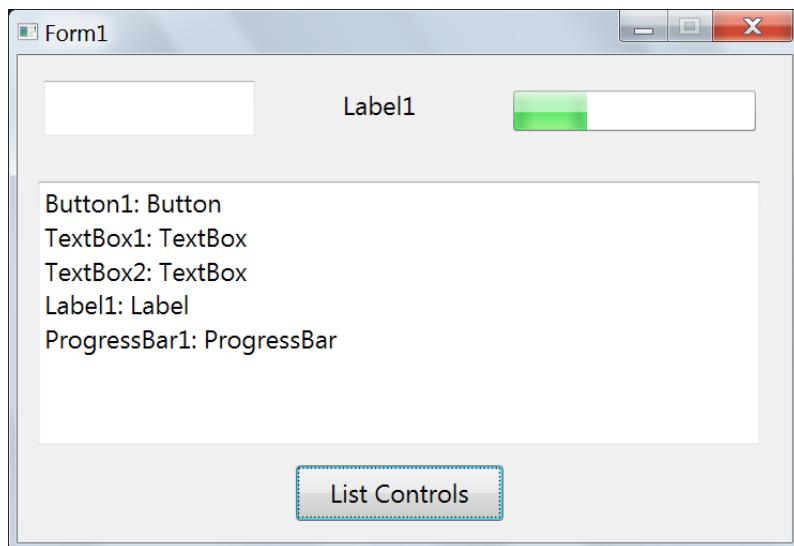
Example:

Add a textbox, a button and some other controls on the form, then add this code in the button click event handler, which will show all the control names and types in the textbox:

```
ForEach control1 In Me.Controls  
    TextBox1.Append(control1.Name)  
    TextBox1.Append(": ")  
    TextBox1.AppendLine(control1.TypeName)
```

[Next](#)

Press F5 to run the project, and click the button. The textbox will show info like this:



Form.DialogResult As DialogResult

Sets the name of the button that the user clicked when he closes the dialog form.

The default value of this property is DialogResult.Cancel, which actually the string "Cancel".

Note that you can't directly read this property after closing the dialog form, but its value is returned by the [Forms.ShowDialog](#) method. See the [DialogResults](#) type example.

Example:

The [DialogResults](#) type contains the most famous button names, but you can also use any custom name you want, or even use an array or a dynamic object that carries any values you want as a result from the dialog form. You can see this in the InputBox.sb file in the "Dialogs" project in the samples folder, where we want to pass the input text and the button name together as the dialog result:

```
Sub BtnOK_OnClick()
    result!InputText = TxtInput.Text
    result!Button = DialogResults.OK
    Me.DialogResult = result
    Me.Close()
EndSub
```

We receive this result in the InputBox function in the Global.sb file like this:

```
result = Forms.ShowDialog("FrmInput", message)
If result = DialogResults.Cancel Then
    Return ""
Else
    Return result!InputText
EndIf
```

Form.Hide()

Hides the form. It actually sets the Form.Visible property to False.

You can show the form again by calling the Form.Show Method.

Form.Icon

Gets or sets the image file path to be displayed as an icon on the title bar of the form.

Form.IsLoaded As Boolean

Returns True if the current form is loaded, whether or not it is shown at this moment.

Form.OnClosed

This event is fired after the form is closed. You can use this event to dispose any resources or save any settings.

For more info, read about [handling control events](#).

Form.OnClosing

This event is fired just before the form is closed.

You can set the Event.Handled property to True to cancel closing the form.

For more info, read about [handling control events](#).

Example:

You can see this event in action in the "sVB Notepad" project in the samples folder. It is used in the frmMain.sb form to call the AskToSave function to ask the user to save changes before closing the form. If the user chooses the cancel button, the AskToSave function returns false, which sets the Event.Handled property to True, so it cancels closing the form.

```

Sub FrmMain_OnClosing()
    Event.Handled = (AskToSave() = False)
EndSub

Function AskToSave()
    If IsModified Then
        result = Dialogs.MessageBox(
            "Save Changes",
            "Do you want to save changes?"
        )

        If result = DialogResult.Yes Then
            If Save() = False Then
                Return False
            EndIf
        ElseIf result = DialogResult.Cancel Then
            Return False
        EndIf
    EndIf

    Return True
EndFunction

```

Form.OnPreviewKeyDown

This event is fired when the user presses a keyboard key down on the form or any of its child controls. A typical usage is to handle this event to respond to all pressed keys even when a control on the form has the focus, like when the user is typing in a textbox. This allows you to handle keyboard shortcuts for menus and other functionality.

Use the [Keyboard](#) properties to get info about the key that fired the event and the state of the Ctrl, Shift, Alt and other special keys.

For more info, read about [handling control events](#).

Example:

You can see this property in action in the `sVB Notepad` project in the samples folder. It is used in the main form (named FrmMain) to execute the shortcut keys for menus. These shortcuts involve pressing the control key like Ctrl+N for opening a new document, so we first check that the control key is pressed, otherwise we exit the subroutine. Then we can check the last pressed key to execute the required action. This is a portion of this event handler, and you can see the full code in the project:

```
Sub FrmMain_OnPreviewKeyDown()
    If Keyboard.CtrlPressed = False Then
        Return
    EndIf

    key = Event.LastKey

    If key = Keys.N Then
        MenuNew_OnClick()
        Event.Handled = True
    ElseIf key = Keys.O Then
        MenuOpen_OnClick()
        Event.Handled = True
    ElseIf key = Keys.S Then
        MenuSave_OnClick()
        Event.Handled = True
    EndIf
EndSub
```

Form.OnPreviewKeyUp

This event is fired when the user releases a keyboard key on the form or any of its child controls. A typical usage is to handle this event to respond to all pressed keys even when a control on the form has the focus, like when the user is typing in a textbox. This allows you to handle keyboard shortcuts for menus and other functionality (See the sample of the OnPreviewKeyDown event).

Use the [Keyboard](#) properties to get info about the key that fired the event and the state of the Ctrl, Shift, Alt and other special keys. For more info, read about [handling control events](#).

Form.OnPreviewMouseWheel

This event is fired repeatedly while the user moves the mouse wheel over the form or any of its controls. The form can also handle the [Control.OnMouseWheel](#) event, but it will not be fired when the mouse is over any control.

Use the [Event.LastMouseWheelDirection](#) property to know if the wheel is moving up or down.

For more info, read about [handling control events](#).

Example:

Add a textbox on the form, then add this handler sub in the code window to allow the user to zoom in or out when he presses the Ctrl key while moving the mouse wheel up and down. Note that this will work even when the mouse is over the textbox:

```
Sub Form1_OnPreviewMouseWheel()
    If Keyboard.CtrlPressed Then
        d = Event.LastMouseWheelDirection
        Me.Width = Me.Width + 6 * d
        Me.Height = Me.Height + 6 * d
        TextBox1.Top = TextBox1.Top + 3 * d
        TextBox1.Left = TextBox1.Left + 3 * d
        TextBox1.Width = TextBox1.Width + 3 * d
        TextBox1.Height = TextBox1.Height + 3 * d
    EndIf
EndSub
```

Form.OnShown

This event is fired after the form is shown and all controls are rendered and are ready to use their properties. So, if you run into any troubles with executing code in the form global area because some controls are still not rendered, you can use this event instead. But don't use the global area and the OnShown event together, because the order of executing them is unknown, so, when you use the OnShown event, put all the initialization code into it, and leave the global area empty.

For more info, read about [handling control events](#).

Form.RemoveControl(controlName)

Removes the given control from the form.

Examples:

Add a textbox, a button, a label and ListBox on the form, then add this code in the button click event handler, which will Remove the textbox, the label and the listbox from the form:

```
Me.RemoveControl(TextBox1)
```

```
Me.RemoveControl(Label1)  
Me.RemoveControl(ListBox1)
```

You can also use a loop to remove all controls (including the button):

```
ForEach _control In Me.Controls  
    Me.RemoveControl(_control)  
Next
```

Form.RunGlobalTests() As Double

Runs the test functions written in the Global.sb file of the current project, and shows the test results in the TxtTest textbox on the current form. If the form doesn't contain a textbox with this name, it will be added at run time to show the results, with a size that makes it cover the whole form.

Note that calling the [UnitTest.RunGlobalTests](#) method will call the Form.RunGlobalTests method of the form passed as an argument.

Note also that calling the [UnitTest.RunAllTests](#) method will run all the tests in the prohect including the global tests, so you don't need to call this method again.

For more info about how to write and debug test functions, see the notes on the [Form.RunTests](#) method in the next section.

Returns:

The number of tests that have been run. If there is no global file, or if it doesn't contains any test functions, this method witll return 0.

Form.RunTests() As Double

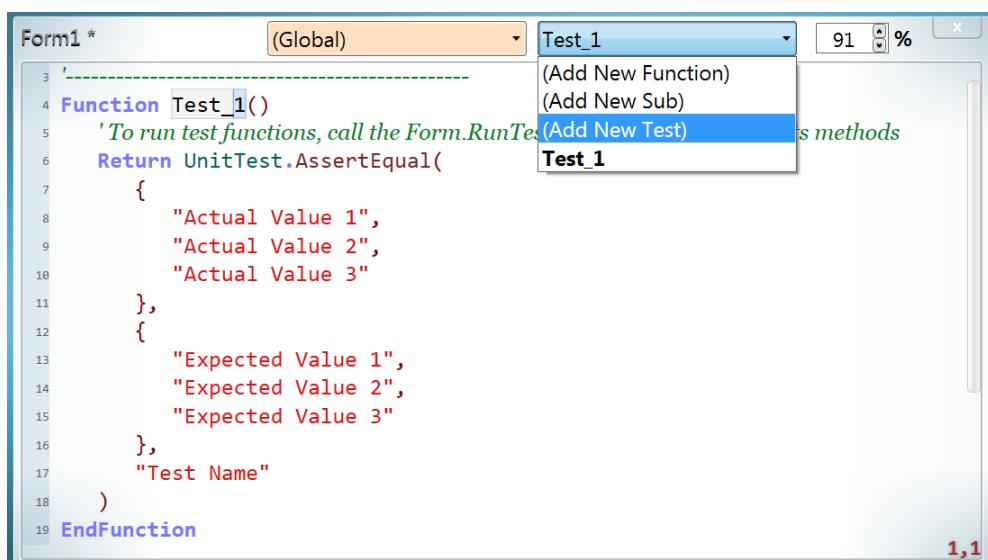
Runs the test functions written in the current form, and shows the test results in the TxtTest textbox. If the form doesn't contain a textbox with this name, it will be added at run time to show the results, with a size that makes it cover the whole form.

Returns:

The number of tests that have been run, or 0 if the form doesn't have any test functions.

Notes:

- To make it easy for you to write test functions, the upper-right dropdown-list of the code editor offers you the "Add New Test" command, and when you click it, it will add a test function template, so you can easily modify its name and code:



- The test function must follow these rules:
 - It must be a function not a sub.
 - Its name must start with 'Test_', like 'Test_FindNames'.

3. It can't have any parameters.
4. Its return value should be a string containing the test result like "passed" or "failed". If you return an empty string, this result will be ignored.

For practical examples, see the test functions written in the "UnitTest Sample" project in the samples folder.

- The auto-completion list of the code editor will not show the tests function names, but you can still write them manually and call them in code if this is what you need.
- When you write test functions in the global.sb file, they will be compiled as friend/internal methods, which means that you can't call them from any other project when you use your project as a code library. So, be careful when you write a code library not to start the name of any non-test function with the "Test_" prefix.
- Use the Form.RunGlobalTests method to run tests written in the Global.sb file.
- Calling the [UnitTest.RunFormTests](#) method with the form passed as an argument will call the Form.RunTests method.
- Calling the [UnitTest.RunAllTests](#) method will call the Form.RunTests method for all forms of the project, and shows the results summary in the textbox that is passed as an argument, in addition to calling the Form.RunGlobalTests method for the parent form of that textbox.
- The sVB debugger can step into test functions one after another, whether they are called by the Form.RunTests, Form.RunGlobalTests, UnitTest.RunFormTests or UnitTest.RunAllTests methods. You can also add breakpoints inside any test function.

Examples:

Add the following two functions to the form:

```
Function Add(a, b)
    Return a + b
EndFunction

Function Divide(a, b)
    If b = 0 Then
        Return ""
    EndIf
    Return a / b
EndFunction
```

Now add the following two test functions to test them:

```
Function Test_Add()
    If Add(1, 2) = 12 Then
        Return "Add(1, 2) failed!"
    ElseIf Add(1, 2) = 3 Then
        Return "Add(1, 2) passed."
    EndIf
EndFunction

Function Test_Divide()
    Return UnitTest.AssertTrue(
    {
        Divide("4", "2") = 2,
        Divide(4, 2) = 2,
        Math.Round2(Divide(5, 3), 2) = 1.67,
        Divide(2, 0) = "",
        Divide(0, 0) = ""
    },
    "Divide"
)
EndFunction
```

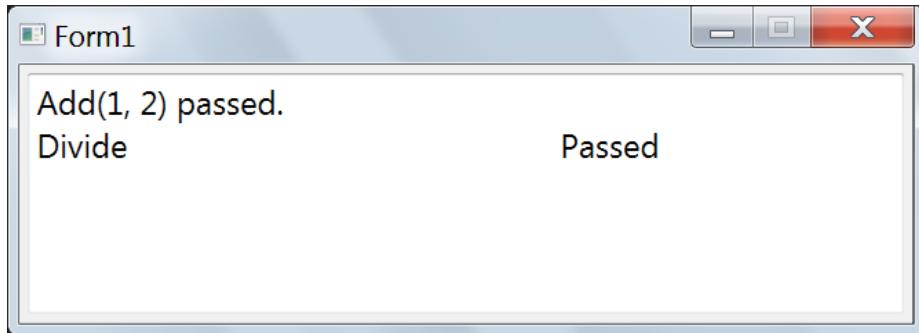
Note that the Test_Add function returns the test results manually, while the Test_Divide function uses one of the [UnitTest](#) methods to check the test values and return a

formatted string containing the results, which makes writing tests easier.

Now to run these two tests, add this statement at the beginning of the form code:

Me.RunTests()

Press F5 to run the project. The form will show these results:



⌚ **Form.Show()**

Displays the form on the screen, if it is loaded but not shown yet, or if it is hidden.

Example:

You can see this method in action in the "Random Buttons" project in the samples folder. It is used in the Button2_OnClick event handler to show the new form that we create in runtime:

```
Sub Button2_OnClick()
    form2 = Forms.AddForm("form2")
    form2.Text = "Form2"
    newButton = CreateRndButton(form2)
    newButton.OnClick = Button2_OnClick
    form2.Show()
EndSub
```

Form.ShowChildForm(childFormName, argsArr) As Form

Shows the form that has the given name as a child form of the current form. This method will do the following:

- Load the design of the given form from its xaml file.
- Pass the given argsArr data to the [Form.ArgsArr](#) property of the target form.
- Execute the code written in the global area of the code file of the target form. You can use [Form.ArgsArr](#) in this global area to initialize the form. Note that global code is executed only when the form is opened for the first time, or after it is closed then re-opened. It will not be executed when you hide or minimize the form then show it again.
- Show the target form as a child dialog form. The child form is a form that always appears on top of its parent form, even when the parent form is the active form. This means that the child form doesn't block the parent form as a modal dialog form does. This allows the user to work with the two forms at the same time. A find and replace window is a good example of a child form, where you can interact with the text you search in while the find and replace window is still displayed. The user can minimize or close the child form alone, but minimizing the parent form will minimize the child form too, and closing the parent form will close the child form too.
- Fire the OnShown Event of the form if it has a handler.

Parameters:

- **childFormName:**

The name of the child form.

● **argsArr:**

Any additional data, an array, or a dynamic object you want to pass to the form. It will be stored in the [Form.ArgsArr](#) property of the child form, so you can use it as you want.

Returns:

The child form name.

Example:

You can see this method in action in the "sVB Notepad" project in the samples folder. It is used in the MenuFind_OnClick event handler to show the search form, named "FrmFind", as a child of the main form. We pass the textbox, named "TxtEditor", to the ArgsArr, so that frmFind can do the search operations in the contents of that textbox.

```
Sub MenuFind_OnClick()
    Me.ShowChildForm("FrmFind", TxtEditor)
EndSub
```

⚙️ **Form.ShowDialog() As DialogResult**

Displays the current form on the screen as a modal dialog, so the user must close it first to be able to access other forms of your app.

Returns:

The dialog result, which can be a string that represents the type of the button that the user clicked, like OK, Yes, No, ... etc. It is easier to use the [DialogResults](#) members instead of string button names to avoid typos.

It is also possible to return an array or a dynamic object to get more info from the dialog. You return this info by setting the

[Form.DialogResult](#) property in the dialog form.

Example:

Add a button on the form, and add write in the form code file:

```
N = 1

Sub Button1_OnClick()
    N = N + 1
    f = Forms.AddForm("form" + N)
    f.Text = "Form " + N
    oKButton = f.AddButton(
        "OKButton", 130, 300, 100, 40)
    oKButton.Text = "OK"
    oKButton.OnClick = DialogButton_OnClick

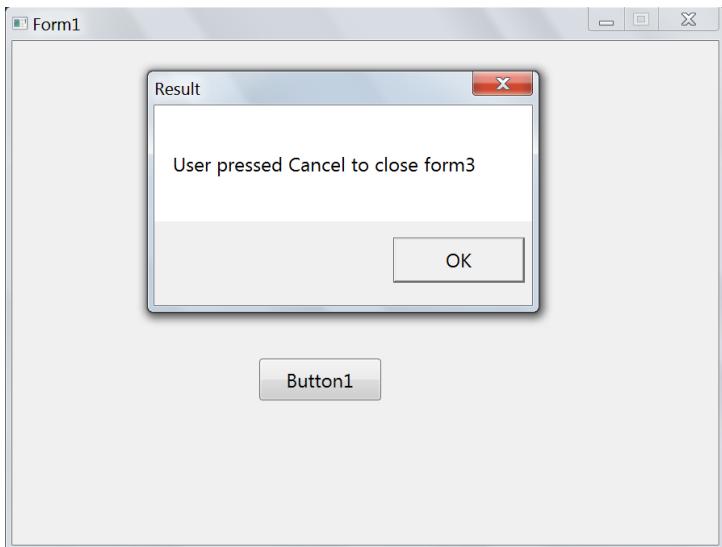
    cancelButton = f.AddButton(
        "CancelButton", 250, 300, 100, 40)
    cancelButton.Text = "Cancel"
    cancelButton.OnClick = DialogButton_OnClick

    Me.ShowMessage(
        "User pressed " + f.ShowDialog()
        + " to close " + f,
        "Result"
    )
EndSub

Sub DialogButton_OnClick()
    _button = Event.SenderControl
    _form = _button.ParentForm
    If _button.Name = "OKButton" Then
        _form.DialogResult = DialogResults.OK
    EndIf
    _form.Close()
EndSub
```

Press F5 to run the project. Each time you click the button, a new dialog form will be shown with OK and Cancel buttons. When you close the form by clicking any of them or using the

form x-button, a message will appear to tell you what button you clicked to close the form.



💡 **Form.ShowMessage(message, title)**

Shows a message box dialog on the current form, which becomes its owner. The message box will show the supplied message and an OK button to close the window.

Note that you can show a Yes/No/Cancel message box and interact with the user choice by using the [Dialogs.MsgBox](#) method.

Parameters:

- **message:**

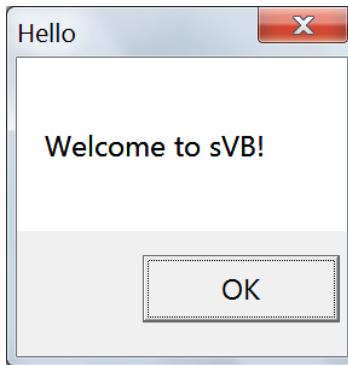
The text to display on the message box.

- **title:**

The title to display of the dialog box.

Example:

```
Me.ShowMessage("Welcome to sVB!", "Hello")
```



Form.Text As String

Gets or sets the title of the form.

Form.Topmost As Boolean

Gets or sets whether or not the form is the topmost window that always appears on top of all other desktop windows even when it is not the active window.

Form.Validate() As Boolean

Calls then [Control.Validate](#) method for every control on the current form. The process stops at first control with errors and returns false.

Note that you need to call this method when the user clicks the button that processes the input data, because he may leave some controls empty and never set the focus on them, so, they never fired the [OnLostFocus event](#), hence they are not validated yet, while your logic requires some of them to have data and can't be empty. Checking the [Error property](#) of these

controls will not solve this issue, because they have not checked for errors yet so they will seem OK. Calling Form.validate will make fix this issue, because it forces each control to fire its OnLostFocus event, so, controls that have handlers for this event will be forced to validate its input.

Returns:

True if the all controls are valid, or False otherwise.

Example:

This method is used in the validate2 button in the Validation project in the samples folder, to make sure that all controls are valid:

```
If Me.Validate() Then  
    Me.ShowMessage("OK", "Validation")  
EndIf
```

The Forms Type

This type provides methods to create forms, show them, and get a list of all opened forms in the program.

The Forms type has the members listed below:

Forms.AddForm(formName) As Form

Creates a new form with the given name, and adds it to the forms collection. The form will not be displayed until you call the Show or ShowDialog method.

Parameter:

- **formName:**

The name of the form.

Returns:

The name of the form. You need to assign it to a variable, to use it as a form object, so you can use it to access the form methods.

Example:

You can see this method in action in the "Random Buttons" project in the samples folder. It is used in the Button2_OnClick event handler to show the new form that we create in runtime:

```
Sub Button2_OnClick()
    form2 = Forms.AddForm("form2")
    form2.Text = "Form2"
    newButton = CreateRndButton(form2)
    newButton.OnClick = Button2_OnClick
    form2.Show()
EndSub
```

Forms.GetForms(*loadedFormsOnly*) As Array

Gets the forms of the current application.

Parameter:

- **loadedFormsOnly:**

Use True, to get the loaded forms only. This will include loaded and hidden forms, but will not include closed forms.

Use False to get all forms defined in this app even if you didn't load them yet.

Returns:

An array containing the names of the required forms you created.. You

Example:

You can see this method in action in the UnitTest project in the samples folder. It is used in the RunAllTests functions in the Global.sb file to get all the forms of the project, and run their tests. We send False to the GetForms method to get the predefined forms in the project even if they are not loaded, because we need to test all of them.

Note that the UnitTest is used as a library for sVB, but GetForms will not return its own forms, and instead it will return the forms of the project that uses the UnitTest library. This is because the GetForms methods search for the forms in the entry assembly.

This is a part of the RunAllTests function:

```
ForEach frm In Forms.GetForms(False)  
    testsCount = testsCount + _TestForm(  
        frm, resultsForm, resultsTextBox)
```

[Next](#)

Where `_TestForm` is a private function that runs the tests for the given form. You can see its code their.

Forms.LoadForm(formName, xamlPath) As Form

Loads a form from its xaml file.

Parameters:

- **formName:**

The name of the form.

- **xamlPath:**

The path pf the xaml file that contains the form design.

Returns:

The name of the form.

Example:

Open sVB, and save the default form (form1) to a folder in your file system, then go to this folder. You will see the 3 files of the form. If you open the "Form1.sb.gen" file with notepad, you will see that it contains this code, which is generated by the form designer:

```
Me = "form1"
_Path = Program.Directory + "\Form1.xaml"
Form1 = Forms.LoadForm("Form1", _Path)
Form.SetArgsArr(Form1,
    Stack.PopValue("_form1_argsArr"))
Form.Show(Form1)
```

Warning: don't ever make any manual changes to this code, because the designer will overwrite them whenever you use it to modify any control.

The above code is executed before the code you write in the form global area, to initialize the form and its controls, by doing the following:

1. It sets the Me variable to the form name.
 2. It loads the form design from the "Form1.xaml", and sets the form name to the Form1 variable.
 3. It sets the [Form.ArgsArr](#) property, by getting its value from the stack. This value is pushed into the stack when you pass it via the second parameter of the [Forms.ShowForm](#), [Forms.ShowDialog](#) and [Form.ShowChildForm](#) methods.
 4. It calls the Form.Show method to display the form.
 5. It sets the event handlers for controls if you add them by the designer or by the editor's upper lists.
-

Forms.ShowDialog(formName, argsArr) As DialogResult

Loads the form that has the given name if exists in the project, and shows it as a modal dialog.

This method will do the following:

- Load the design of the given form from its xaml file.
- Pass the given argsArr data to the [Form.ArgsArr](#) property of the target form.
- Execute the code written in the global area of the code file of the target form. You can use [Form.ArgsArr](#) in this global area to initialize the form. Note that global code is executed only when the form is opened for the first time, or after it is closed then re-opened. It will not be executed when you

hide or minimize the form then show it again.

- Show the target form as a modal dialog form.
- Fire the OnShown Event of the form if it has a handler.

Parameters:

- **formName:**

The name of the form.

- **argsArr:**

Any additional data, array, or a dynamic object you want to pass to the form. It will be stored in the [Form.ArgsArr](#) property, so you can use it as you want.

Returns:

The dialog result, which can be a string that represents the type of the button that the user clicked, like OK, Yes, No, ... etc. It is easier to use the [DialogResults](#) members instead of string button names to avoid typos.

It is also possible to return an array or a dynamic object to get more info from the dialog. You return this info by setting the [Form.DialogResult](#) property in the dialog form.

Examples:

You can see this method in action in the "Show Dialogs" project in the samples folder. It is used in the MsgBox function in the Global.sb file to show the frmMsg:

```
info!Title = strTitle  
info!Message = strMessage  
Return Forms.ShowDialog("FrmMsg", info)
```

It is also used in the InputBox function in the Global.sb file to show the FrmInput:

```
result = Forms.ShowDialog("FrmInput", message)
If result = DialogResult.OK Then
    Return InputText
Else
    Return ""
EndIf
```

Forms.ShowDialog(formName, argsArr) As Form

Shows the form that has the given name if exists in the project.

This method will do the following:

- Load the design of the given form from its xaml file.
- Pass the given argsArr data to the [Form.ArgsArr](#) property of the target form.
- Execute the code written in the global area of the code file of the target form. You can use [Form.ArgsArr](#) in this global area to initialize the form. Note that global code is executed only when the form is opened for the first time, or after it is closed then re-opened. It will not be executed when you hide or minimize the form then show it again.
- Show the target form as a normal form.
- Fire the OnShown Event of the form if it has a handler.

Parameters:

- **formName:**

The name of the form.

- **argsArr:**

Any additional data, array, or a dynamic object you want to pass to the form. It will be stored in the [Form.ArgsArr](#) property, so you can use it as you want.

Returns:

The form name.

Example:

You can see this method in action in the "Random Buttons 3" project in the samples folder. It is used in the Button2_OnClick event handler in the Form1.sb file to show the form2 form:

```
Forms .ShowForm("form2", "")
```

⚙️ Forms.ShowMessage(message, title)

Shows a message box dialog. It is similar to the Form.ShowMessage method, but it doesn't set any owner to the message box form, which has no actual effect. But the Forms.ShowMessage method is still helpful when you write code in a normal sb file (that has no form), like the global.sb file, or any other plain sb file you create or open in the code editor.

Parameters:

- **message:**

The text to display on the message box.

- **title:**

The title to display of the dialog box.

Note:

You can use MsgBox as a shortcut name to show the message box with the default title "Message". Ex:

```
MsgBox "Hello!"
```

When you leave the line or save the file, the above code will be converted to:

```
Forms .ShowMessage("Hello!", "Message")
```

You can also provide the title for the MsgBox shortcut syntax like this:

```
MsgBox "How are you?", "Hello"
```

Which will be converted to:

```
Forms.ShowMessage("How are you?", "Hello")
```

Note also that you can show a Yes/No message and interact with the user choice by using the [Dialogs.MsgBox](#) method.

Example:

Open sVB, select to the code editor tab, press Ctrl+N (or click the "New File" button on the toolbar) to open a new sb file, and add this code to it:

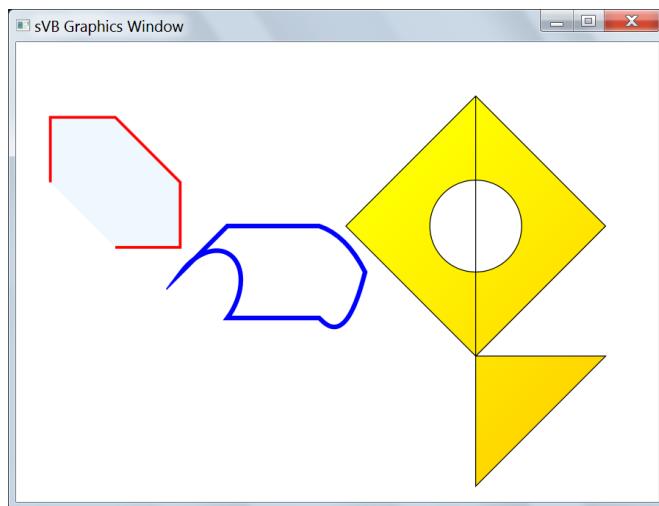
```
Forms.ShowMessage("Welcome to sVB!", "Hello")
```

Press F5 to run the code. You will get this message:



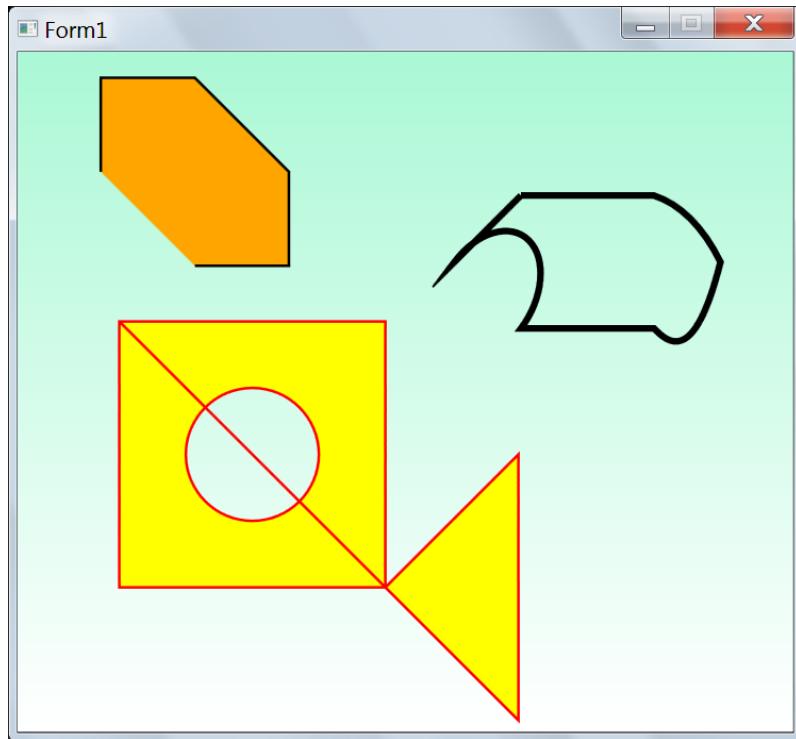
The GeometricPath Type

Allows you to combine shapes into one path, and create new geometric figures from line and arc segments, so you can compose a new complex custom shapes. You can add the geometric path to the shapes collection by calling the [Shapes.AddGeometricPath](#) method which draws it on the graphics window, then you can apply any rotation or animation on it as you do with any other normal shape. The "Geometric Path" project in the samples folder, shows you an example on that.



You can also add the geometric path to any label on any form by calling the [Label.AddGeometricPath](#) method, so you can move, rotate, or animate it using the label methods. You can even respond with the user interactions with the shape by handling the label events like `OnClick`.

The "Geometric Path 2" project in the samples folder, shows you an example on that, where you can drag any of the tree shapes to change its position on the form.



The GeometricPath type has these methods:

GeometricPath.AddArcSegment(x, y, xRadius, yRadius, angle, isLargeArc, isClockwise, usePen)

Adds an arc segment to the current figure in the geometric path, starting from the last point in the figure, and ending at the given point.

Parameters:

- **x:**

The x co-ordinate of the end point of the arc segment.

- **y:**

The y co-ordinate of the end point of the arc segment.

- **xRadius:**

The horizontal radius of the arc.

- **yRadius:**

The vertical radius of the arc.

- **angle:**

The x-axis rotation of the ellipse.

- **isLargArc:**

Use True if the arc should be greater than 180 degrees, or False otherwise

- **isClockwise:**

Use True to draw the arc in a positive angle direction, or False otherwise

- **usePen:**

Use True to draw the segment with the pen color, or False to hide the segment outline.

Example:

You can see this method in action in the "Geometric Path" and "Geometric Path 2" projects in the samples folder. It used in the CreateFigure2 function like this:

```
GeometricPath.AddArcSegment(  
    50, 100,  
    20, 30,  
    30,  
    False,  
    True,  
    True  
)
```

You can also see the examples provided for the [GraphicsWindow.DrawArc](#) method, which is similar to this method but draws directly on the Graphics window.

GeometricPath.AddBezierSegment(x1, y1, x2, y2, x3, y3, usePen)

Adds a cubic Bezier curve segment to the current figure in the geometric path, starting from the last point in the figure, passing through the two given control points, and ending at the given end point.

Parameters:

- **x1:**

The x co-ordinate of the first control point.

- **y1:**

The y co-ordinate of the first control point.

- **x2:**

The x co-ordinate of the second control point.

- **y2:**

The y co-ordinate of the second control point.

- **x3:**

The x co-ordinate of the end point.

- **y3:**

The y co-ordinate of the end point.

- **usePen:**

Use True to draw the segment with the pen color, or False to hide the segment outline.

Example:

You can see this method in action in the "Geometric Path" and "Geometric Path 2" projects in the samples folder. It used in the CreateFigure2 function like this:

```
GeometricPath.AddBezierSegment(  
    160, 110, 'First control point  
    180, 130, 'Second control point  
    200, 50,   'End point  
    True  
)
```

GeometricPath.AddEllipse(x, y, width, height)

Adds an ellipse to the geometric path.

Parameters:

- **x:**

The x co-ordinate of the ellipse.

- **y:**

The y co-ordinate of the ellipse.

- **width:**

The width of the ellipse.

- **height:**

The height of the ellipse.

Example:

You can see this method in action in the "Geometric Path" and "Geometric Path 2" projects in the samples folder.

It used in the CreateFigure3 function like this:

```
GeometricPath.AddEllipse(50, 50, 100, 100)
```

GeometricPath.AddLine(x1, y1, x2, y2)

Adds a line to the geometric path.

Parameters:

- **x1:**

The x co-ordinate of the first point.

- **y1:**

The y co-ordinate of the first point.

- **x2:**

The x co-ordinate of the second point.

- **y2:**

The y co-ordinate of the second point.

Example:

You can see this method in action in the "Geometric Path" and "Geometric Path 2" projects in the samples folder.

It used in the CreateFigure3 function like this:

```
GeometricPath.AddLine(0, 0, 300, 300)
```

GeometricPath.AddLineSegment(x, y, usePen)

Adds a line segment to the current figure in the geometric path, starting from the last point in the figure, and ending at the given point.

Parameters:

- **x:**

The x co-ordinate of the end point of the line segment.

- **y:**

The y co-ordinate of the end point of the line segment.

- **usePen:**

Use True to draw the segment with the pen color, or False to hide the segment outline.

Example:

You can see this method in action in the "Geometric Path" and "Geometric Path 2" projects in the samples folder.

It used in the CreateFigure1 function to create a hexagon shape:

```
Sub CreateFigure1()
    GeometricPath.CreateFigure(50, 0, True)
    GeometricPath.AddLineSegment(0, 50, True)
    GeometricPath.AddLineSegment(50, 100, True)
    GeometricPath.AddLineSegment(150, 100, False)
    GeometricPath.AddLineSegment(200, 50, True)
    GeometricPath.AddLineSegment(150, 0, True)
    Label1.AddGeometricPath(
        Colors.Black, 2, Colors.Orange)
    Label1.Rotate(45)
EndSub
```

GeometricPath.AddQuadraticBezierSegment(x1, y1, x2, y2, usePen)

Adds a cubic quadratic Bezier curve segment to the current figure in the geometric path, starting from the last point in the figure, passing through the given control point and ending at the given end point.

Parameters:

- **x1:**

The x co-ordinate of the control point.

- **y1:**

The y co-ordinate of the control point.

- **x2:**

The x co-ordinate of the end point.

- **y2:**

The y co-ordinate of the end point.

- **usePen:**

Use True to draw the segment with the pen color, or False to hide the segment outline.

Example:

You can see this method in action in the "Geometric Path" and "Geometric Path 2" projects in the samples folder.

It used in the CreateFigure2 function like this:

```
GeometricPath.AddQuadraticBezierSegment(  
    180, 10, 'Contol point  
    150, 0, 'End point  
    True  
)
```

GeometricPath.AddRectangle(x, y, width, height)

Adds a rectangle to the geometric path.

Parameters:

- **x:**

The x co-ordinate of the rectangle.

- **y:**

The y co-ordinate of the rectangle.

- **width:**

The width of the rectangle.

- **height:**

The height of the rectangle.

Example:

You can see this method in action in the "Geometric Path" and "Geometric Path 2" projects in the samples folder.

It used in the CreateFigure3 function like this:

```
GeometricPath.AddRectangle(0, 0, 200, 200)
```

⚙️ GeometricPath.AddTriangle(x1, y1, x2, y2, x3, y3)

Adds a triangle to the geometric path.

Parameters:

- **x1:**

The x co-ordinate of the first point.

- **y1:**

The y co-ordinate of the first point.

- **x2:**

The x co-ordinate of the second point.

- **y2:**

The y co-ordinate of the second point.

- **x3:**

The x co-ordinate of the third point.

- **y3:**

The y co-ordinate of the third point.

Example:

You can see this method in action in the "Geometric Path" and "Geometric Path 2" projects in the samples folder.

It used in the CreateFigure3 function like this:

```
GeometricPath.AddTriangle(  
    200, 200,  
    300, 100,  
    300, 300  
)
```

GeometricPath.CreateFigure(x, y, isClosed)

Adds a new figure to the geometric path that starts at the given point. You can create as many figures as you want in the geometric path. The AddxxxSegment methods will add the segments to the last figure you created.

Parameters:

- **x:**

The x co-ordinate of the start point of the figure

- **y:**

The y co-ordinate of the start point of the figure

- **isClosed:**

When True, a line segment is automatically drawn between the last point and the start point of the figure to make it a

closed shape. Use False if you want to draw an open figure like a curve.

Example:

You can see this method in action in the "Geometric Path" and "Geometric Path 2" projects in the samples folder.

It used in the CreateFigure2 function like this:

```
GeometricPath.CreateFigure(50, 0, False)
```

⚙️ GeometricPath.CreatePath()

Creates a new geometric path to add geometric shapes to it, so you can compose complex shapes. Call this method after adding the last path to the Shapes or the label, so you can start with a new empty path.

Example:

You can see this method in action in the "Geometric Path" and "Geometric Path 2" projects in the samples folder.

It used at the start of the CreateFigure1, CreateFigure2 and CreateFigure3 functions like this:

```
GeometricPath.CreatePath()
```

The Geometrics Type

Contains methods to create custom geometric figures. This is a sample sVB library, and its source code exists in the Geometrics project in the samples folder.

The Geometrics type has these members:

Geometrics.AllowDrag(targetControl)

Allows the user to drag the given target control by mouse, to change its position on the screen.

Parameter:

- **targetControl:**

The control you want to allow the user to drag. You can drag labels or any other controls. You can even drag the form itself!

Examples:

Add a list box on the form, then write this code in the form global area:

Geometrics.AllowDrag(ListBox1)

Press F5 to run the project. You can now drag the listbox by mouse to change its location on the form! Just left-click the listbox and hold the mouse button down while you move the mouse to a new location, then release the its button.

Geometrics.CreateCompositShape(targetLabel, penColor, penWidth, brushColor)

Creates a shape composed of basic geometric shapes.

Parameters:

- **targetLabel:**

The label to draw the figure on.

- **penColor:**

The color used to draw the figure outline.

- **penWidth:**

The thickness of the outline.

- **brushColor:**

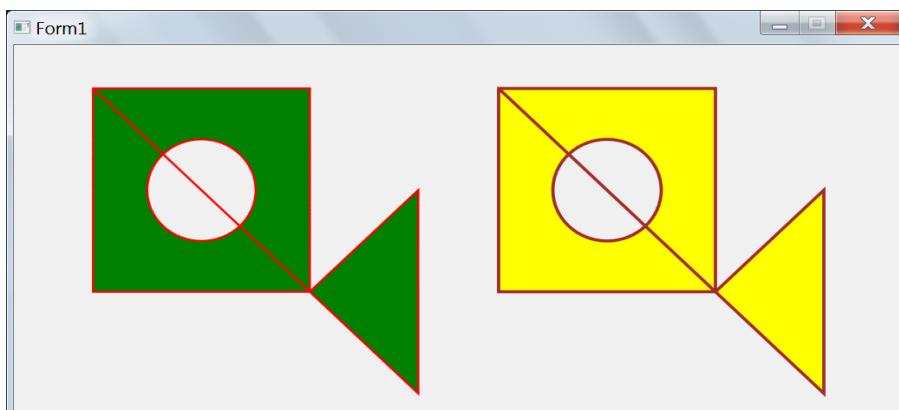
The color used to fill the internal area of the figure.

Examples:

Add a list box on the form, then write this code in the form global area:

```
Geometrics.CreateCompositShape(  
    Label1, Colors.Red, 2, Colors.Green)  
Geometrics.CreateCompositShape(  
    Label2, Colors.Brown, 3, Colors.Yellow)
```

Press F5 to run the project. The form will appear as this:



Geometrics.CreateCurvyHexagon(targetLabel, penColor, penWidth, brushColor)

Creates a hexagonal figure with curves.

Parameters:

- **targetLabel:**

The label to draw the figure on.

- **penColor:**

The color used to draw the figure outline.

- **penWidth:**

The thickness of the outline.

- **brushColor:**

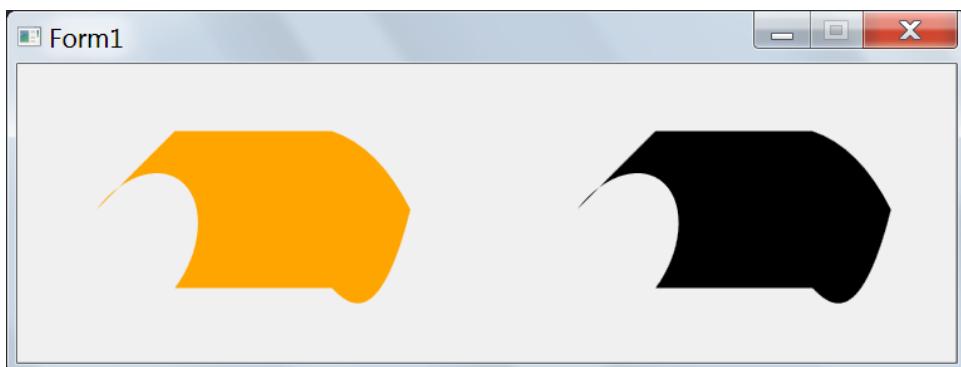
The color used to fill the internal area of the figure.

Examples:

Add a list box on the form, then write this code in the form global area:

```
Geometrics.CreateCurvyHexagon(  
    Label1, Colors.Black, 0, Colors.Orange)  
Geometrics.CreateCurvyHexagon(  
    Label2, Colors.None, 1, Colors.Black)
```

Press F5 to run the project. The form will appear as this:



Geometrics.CreateHexagon(**targetLabel**, **penColor**, **penWidth**, **brushColor**)

Creates a hexagonal figure with strait lines.

Parameters:

- **targetLabel**:

The label to draw the figure on.

- **penColor**:

The color used to draw the figure outline.

- **penWidth**:

The thickness of the outline.

- **brushColor**:

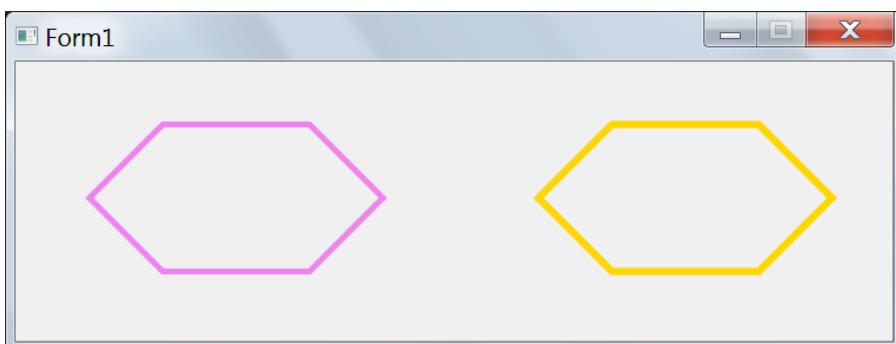
The color used to fill the internal area of the figure.

Examples:

Add a list box on the form, then write this code in the form global area:

```
Geometrics.CreateHexagon(  
    Label1, Colors.Violet, 4, Colors.None)  
Geometrics.CreateHexagon(  
    Label2, Colors.Gold, 5, Colors.Transparent)
```

Press F5 to run the project. The form will appear as this:



Geometrics.PreventDrag(targetControl)

Prevents the user from dragdng the given target control, if it was enabled before by calling the Geometrics.AllowDrag method.

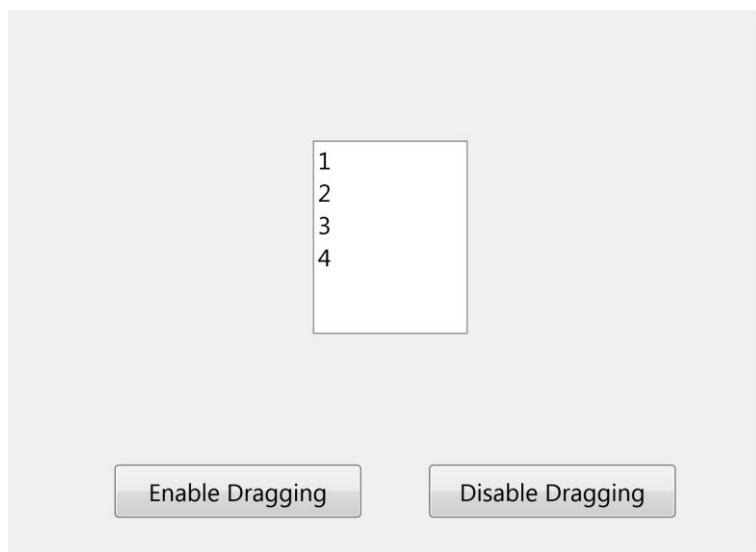
Parameter:

- **targetControl:**

The control you want to prevent the user from dragdng.

Examples:

Add a listbox and tow buttons on the form, and design them as you see in this picture:



Clicking the first button will allow the user to drag the listbox, and clicking the second will prevent that. The user can enable and disable dragging as many times as he wishes. This is the code that does that.

```
Sub Button1_OnClick()
    Geometrics.AllowDrag(ListBox1)
EndSub
```

```
Sub Button2_OnClick()
    Geometrics.PreventDrag(ListBox1)
EndSub
```

The GraphicsWindow Type

The GraphicsWindow provides graphics related input and output functionality. For example, using this type, it is possible to draw and fill circles and rectangles.

Note that sVB (starting from v2.8.6.1) allows you to use **GW** as a shortcut for the **GraphicsWindow** type. For example, the following statement:

```
GW.BackColor = Colors.AliceBlue
```

Is a valid shortcut for this statement:

```
GraphicsWindow.BackColor = Colors.AliceBlue
```

The GraphicsWindow type has these members:

GraphicsWindow.AsForm() As Form

Show the graphics window and gets the form object that represents it.

Returns:

The graphics window name which is "graphicswindow".

Note that sVB can deal with this name as a form object, so, you can use it to access the properties and methods of the form that represents the graphics window.

Examples:

Add a button of the form, double-click it to go to its OnClick event handler, and write this code into it:

```
gw = GraphicsWindow.AsForm()  
gw.ChooseBackColor()
```

This will show the graphics window and show the color dialog to allow you to choose the background color.

GraphicsWindow.AutoScale As Boolean

Set this property to true, to prevent the closed Graphics Window from being auto-shown again when you use one of its properties or methods. You can allow display the GW by calling the Show method, regardless of the value of this property.

The default value is True.

Example:

You may need this when you use a loop to draw some graphics, to allow the user to close the GW before the loop is finished. Don't forget to set this property back to True after the loop ends.

Note that this approach will not be needed if the GW is the only displayed window in your program, because closing it will end the program, but this will not be the case when you display the GW from another form in your project.

You may also need to check the GW.IsClosed property inside the loop to be able to exit the loop when the GW is closed.

To see this in action, look at the code written in the buttons of the Spirals project in the samples folder. For example, this is the code of the first button:

```
GW.AutoScale = False  
For i = 1 To 100 Step 3  
    If GW.IsClosed Then  
        ExitLoop  
    EndIf  
  
    Turtle.Move(2 * i)  
    Turtle.Turn(60)  
  
Next  
GW.AutoScale = True
```

GraphicsWindow.BackgroundColor As Color

Gets or sets the Background color of the Graphics Window.

Note that you can use this property to show a transparent Graphics Window. To do that you must set this property before showing the Graphics Window (i.e before using any other property or method of the Graphics Window), where the value that you use for the window background should be one of these value:

1. Colors.None
2. Colors.transparent
3. Any color that has a transparency ration other than zero.

Examples:

You can find two samples on transparent Graphics Window in the sVB samples folder:

2. The "Half transparent GW.sb" file shows you how to use the turtle to draw a circle on a half transparent window.



This is the line of code that makes the GW semi transparent:

```
GW.BackgroundColor = Color.FromARGB(  
    128, 0, 255, 255)
```

3. The "Two Balls.sb" file in the Balls folder shows you how to move two balls on your desktop by setting the background of the Graphics Window to Colors.None:

```
GW.BackgroundColor = Colors.None
```

GraphicsWindow.BrushColor As Color

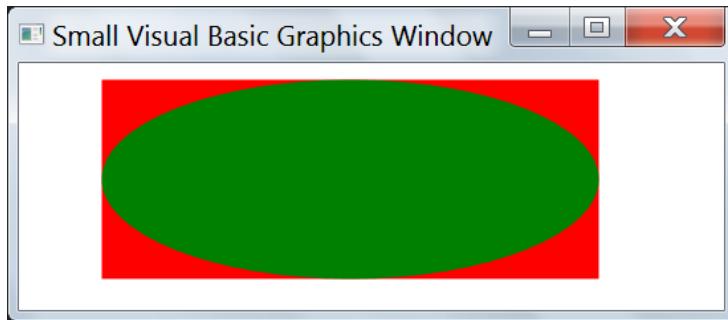
Gets or sets the brush color to be used to fill shapes drawn on the Graphics Window using the FillTriangle, FillEllipse, and FillRectangle methods. The shapes that already exist will not be affected when you change the brush color.

Examples:

In sVB code editor, open a new sb file, and write this code:

```
GraphicsWindow.BrushColor = Colors.Red  
GraphicsWindow.FillRectangle(50, 10, 300, 120)  
GraphicsWindow.BrushColor = Colors.Green  
GraphicsWindow.FillEllipse(50, 10, 300, 120)
```

Press F5 to run the code. This will display the graphics window, which will show a green ellipse inside a red rectangle:



➊ **GraphicsWindow.CanResize As Boolean**

Specifies whether or not the Graphics Window can be resized by the user. This will not affect your ability to resize the window from code by changing the Width and Height properties.

Example:

To prevent the user from resizing the Graphics Window, use this code:

```
GraphicsWindow.CanResize = False
```

➋ **GraphicsWindow.Clear()**

Clears the window, which means erasing all shapes and controls from the window.

➌ **GraphicsWindow.DrawArc(1x, y1, x2, y2, xRadius, yRadius, angle, isLargeArc, isClockwise)**

Draws an arc between the given two points. This arc is a part of the ellipse that has the given radius and passes through these two points.

Parameters:

- **x1:**

The x co-ordinate of the start point of the arc.

- **y1:**

The y co-ordinate of the start point of the arc.

- **x2:**

The x co-ordinate of the end point of the arc.

- **y2:**

The y co-ordinate of the end point of the arc.

- **xRadius:**

The horizontal radius of the arc.

- **yRadius:**

The vertical radius of the arc.

- **angle:**

The x-axis rotation of the ellipse.

- **isLargArc:**

Use True if the arc should be greater than 180 degrees, or False otherwise

- **isClockwise:**

Use True to draw the arc in a positive angle direction, or False otherwise

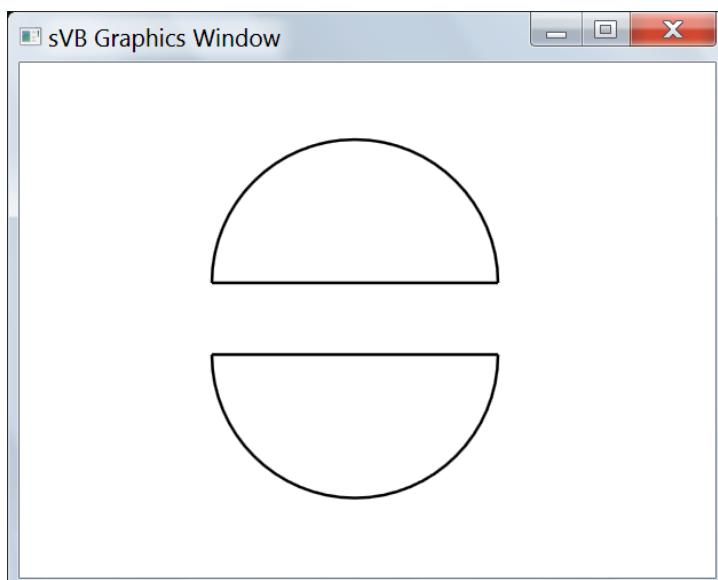
- **usePen:**

Use True to draw the segment with the pen color, or False to hide the segment outline.

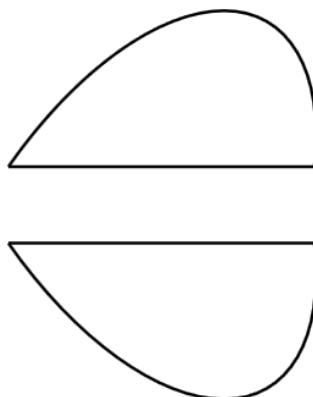
Examples:

You can use this method to draw a part of a circle by using the same value for xRadius and yRadius and choose two points on the circle to draw the arc between. For example, you can choose two points on the horizontal diameter (i.e. Y is the same for the two points) to draw a half circle. The following code shows you how to do it and also how to use the isClockwise parameter to show the upper and lower halves of the circle:

```
GW.DrawArc(  
    100, 200,  
    300, 200,  
    100, 100,  
    0, False, True)  
GW.DrawLine(100, 200, 300, 200)  
GW.DrawArc(  
    100, 250,  
    300, 250,  
    100, 100,  
    0, False, False)  
GW.DrawLine(100, 250, 300, 250)
```



To see the effect of the angle parameter, use different values for the xRadius and yRadius. For example, use 100, 150 for xRadius and yRadius parameters in the above code, and use 30 and -30 for the angle parameter of the two halves respectively and see the deference. The result will look like this:



It is a best practice to keep playing with parameter values and studying the effects.

GraphicsWindow.DrawBezierCurve(**x1, y1, x2, y2, x3, y3, x4, y4**)

Draws a cubic Bezier curve between the given start and end points, and passing through the two given control points.

Parameters:

- **x1:**

The x co-ordinate of the start point.

- **y1:**

The y co-ordinate of the end point.

- **x2:**

The x co-ordinate of the first control point.

- **y2:**

The y co-ordinate of the first control point.

- **x3:**

The x co-ordinate of the second control point.

- **y3:**

The y co-ordinate of the second control point.

- **x4:**

The x co-ordinate of the end point.

- **y4:**

The y co-ordinate of the end point.

Example:

```
GW.DrawBezierCurve(  
    100, 150,      ' Start point  
    160, 110,      ' First control point  
    180, 130,      ' Second control point  
    200, 50        ' End point  
)
```



GraphicsWindow.DrawBoundText(x, y, width, text)

Draws a line of text on the screen at the specified location.

Parameters:

- **x:**

The x co-ordinate of the text start point.

- **y:**

The y co-ordinate of the text start point.

- **width:**

The maximum available width. This parameter helps define when the text should wrap to a new line.

- **text:**

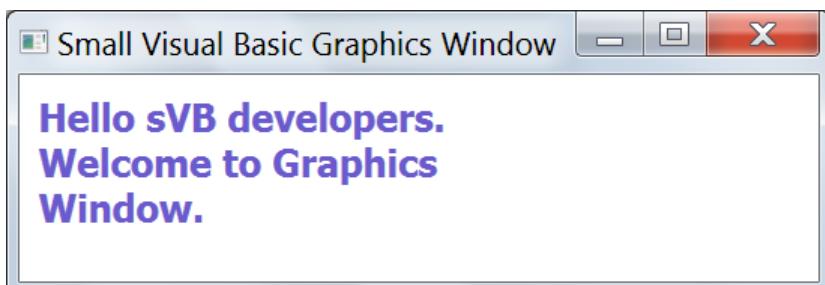
The text to draw.

Example:

In sVB code editor, open a new sb file, and write this code:

```
GraphicsWindow.FontSize = 20
GraphicsWindow.DrawBoundText(
    10, 10, 300,
    "Hello sVB developers. Welcome to Graphics Window."
)
```

Press F5 to run the code. This will display the graphics window, which will show this text:



GraphicsWindow.DrawEllipse(x, y, width, height)

Draws an ellipse using the graphics window pen.

Parameters:

- **x:**

The x co-ordinate of the ellipse.

- **y:**

The y co-ordinate of the ellipse.

- **width:**

The width of the ellipse.

- **height:**

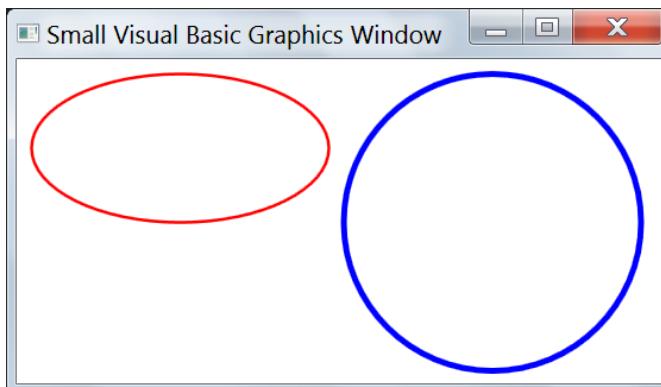
The height of the ellipse.

Examples:

In sVB code editor, open a new sb file, and write this code:

```
GraphicsWindow.PenColor = Colors.Red  
GraphicsWindow.PenWidth = 2  
GraphicsWindow.DrawEllipse(10, 10, 200, 100)  
  
GraphicsWindow.PenColor = Colors.Blue  
GraphicsWindow.PenWidth = 4  
GraphicsWindow.DrawEllipse(220, 10, 200, 200)
```

Press F5 to run the code. This will display the graphics window, which will show a red ellipse and a blue circle:



① **GraphicsWindow.DrawImage(imageName, x, y)**

Draws the specified image from memory.

Parameters:

- **imageName:**

The name of the image to draw. You must add an image with this name to the [ImageList](#), so you can draw it with this method.

- **x:**

The x co-ordinate of the point to draw the image at.

- **y:**

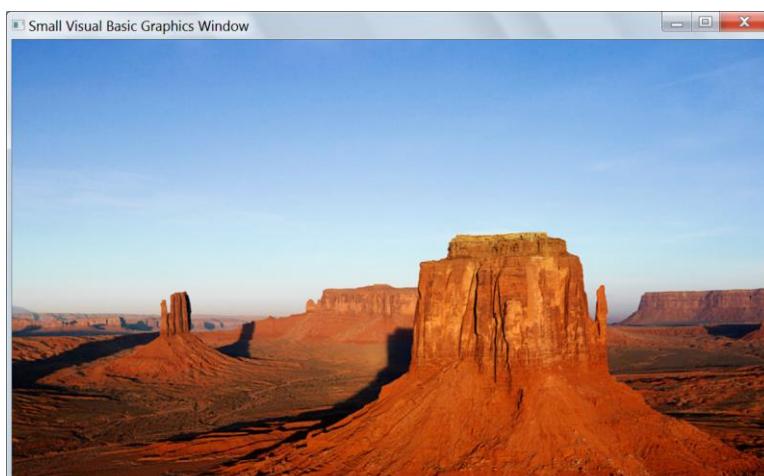
The y co-ordinate of the point to draw the image at.

Example:

In sVB code editor, open a new sb file, and write this code:

```
path = "%PUBLIC%\Pictures\Sample Pictures\Desert.jpg"
Image1 = ImageList.LoadImage(path)
GraphicsWindow.DrawImage(Image1, 0, 0)
```

Note you may change the value of the path variable to any valid image path on your PC. When you run the project, the graphics window will show the picture.



① **GraphicsWindow.DrawLine(x1, y1, x2, y2)**

Draws a line from one point to another using the graphics window pen.

Parameters:

- **x1:**

The x co-ordinate of the first point.

- **y1:**

The y co-ordinate of the first point.

- **x2:**

The x co-ordinate of the second point.

- **y2:**

The y co-ordinate of the second point.

Examples:

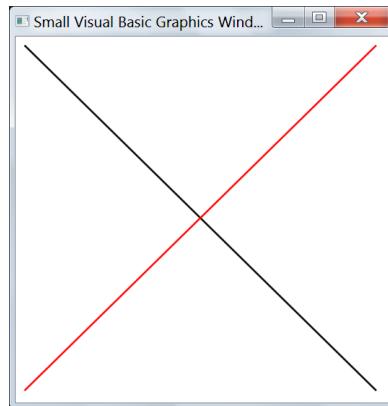
In sVB code editor, open a new sb file, and write this code:

```
GraphicsWindow.DrawLine(10, 10, 400, 400)
```

```
GraphicsWindow.PenColor = Colors.Red
```

```
GraphicsWindow.DrawLine(10, 400, 400, 10)
```

Press F5 to run the code. This will display the graphics window, which will show a black and a red intersected lines:



GraphicsWindow.DrawPolygon(

xOffset, yOffset,
xScale, yScale,
pointsArr)

Draws the polygon that is represented by the given points array, with the pen of the graphics window.

You don't need to represent the exact position and size in the points array, as you can use the offset and scale parameters to change the polygon position and size. So, you should just focus on choosing the points that form the polygon outline.

Parameters:

- **xOffset As Primitive:**

The horizontal offset to add to the x-coordinates of each point of the polygon.

- **yOffset As Primitive:**

The vertical offset to add to the y-coordinate of each point of the polygon.

- **xScale As Primitive:**

The factor to multiply the polygon width by.

- **yScale As Primitive:**

The factor to multiply the polygon height by.

- **pointsArr:**

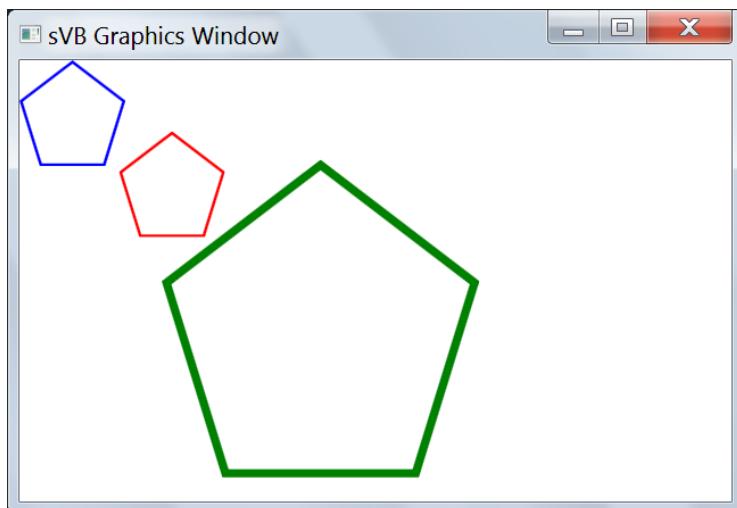
An array of points representing the heads of the polygon. Each item in this array is an array containing the x and y positions of the point.

Example:

You can see this method in action in the "Draw Pentagon" project in the samples folder.

The following code draws a pentagon, and shows you the effect of applying different offset and scaling on it:

```
PentaPoints = {  
    {37.50, 0.50},  
    {74.50, 28.76},  
    {60.35, 74.50},  
    {14.65, 74.50},  
    {0.50, 28.76}  
}  
GW.PenColor = Colors.Blue  
GW.PenWidth = 2  
GW.DrawPolygon(0, 0, 1, 1, PentaPoints)  
GW.PenColor = Colors.Red  
GW.DrawPolygon(70, 50, 1, 1, PentaPoints)  
GW.PenColor = Colors.Green  
GW.DrawPolygon(100, 70, 3, 3, PentaPoints)
```



GraphicsWindow.DrawQuadraticBezierCurve()

x1, y1, x2, y2, x3, y3)

Draws a cubic quadratic Bezier curve between the given start and end points, and passing through the given control point.

Parameters:

- **x1:**

The x co-ordinate of the start point.

- **y1:**

The y co-ordinate of the start point.

- **x2:**

The x co-ordinate of the control point.

- **y2:**

The y co-ordinate of the control point.

- **x3:**

The x co-ordinate of the end point.

- **y3:**

The y co-ordinate of the end point.

Example:

```
GW.DrawQuadraticBezierCurve(  
    50, 0, ' Start point  
    180, 10, ' Contol point  
    150, 200 ' End point  
)
```



GraphicsWindow.DrawRectangle(

x, y, width, height)

Draws a rectangle using the graphics window pen.

Parameters:

- **x:**

The x co-ordinate of the rectangle.

- **y:**

The y co-ordinate of the rectangle.

- **width:**

The width of the rectangle.

- **height:**

The height of the rectangle.

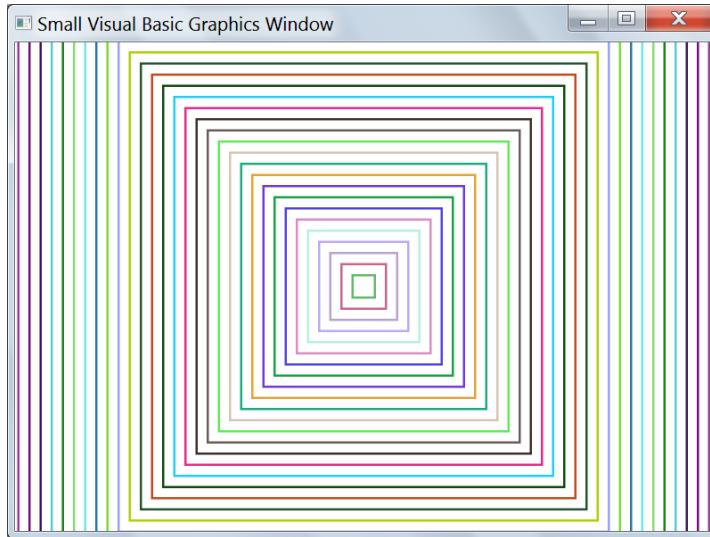
Example:

In sVB code editor, open a new sb file, and write this code:

```
W = GraphicsWindow.Width  
H = GraphicsWindow.Height  
X = W / 2  
Y = H / 2  
  
For I = 10 To H Step 10  
    GraphicsWindow.PenColor = Colors.Random  
    GraphicsWindow.DrawRectangle(  
        X - I,  
        Y - I,  
        I * 2,  
        I * 2  
    )
```

Next

Press F5 to run the code. This will display the graphics window, which will show nested squares with random colors:



⌚ **GraphicsWindow.DrawResizedImage(imageName, x, y, width, height)**

Draws the specified image from memory on to the screen, in the specified size.

Parameters:

- **imageName:**

The name of the image to draw.

- **x:**

The x co-ordinate of the point to draw the image at.

- **y:**

The y co-ordinate of the point to draw the image at.

- **width:**

The width to draw the image.

- **height:**

The height to draw the image.

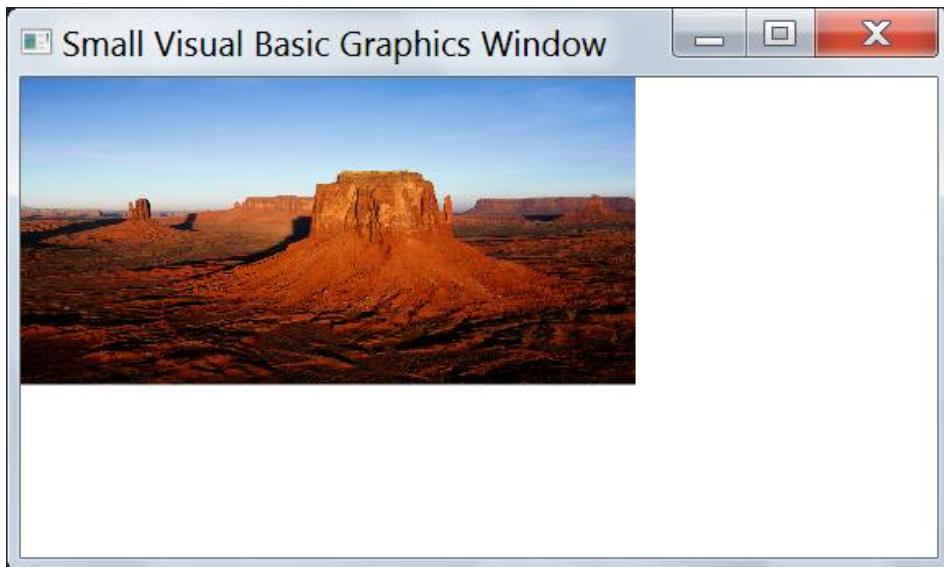
Example:

In sVB code editor, open a new sb file, and write this code:

```
path = "%PUBLIC%\Pictures\Sample Pictures\Desktop.jpg"  
Image1 = ImageList.LoadImage(path)  
GraphicsWindow.DrawResizedImage(  
    Image1, 0, 0, 300, 150)
```

Note you may change the value of the path variable to any valid image path on your PC.

When you run the project, the graphics window will show the picture in the given rectangle.



GraphicsWindow.DrawText(x, y, text)

Draws a line of text on the screen at the specified location. Long text can disappear out of the window right border. If you want to allow wrapping long text over new lines, use the [DrawBoundText](#) method.

Parameters:

- **x:**

The x co-ordinate of the text start point.

- **y:**

The y co-ordinate of the text start point.

- **text:**

The text to draw.

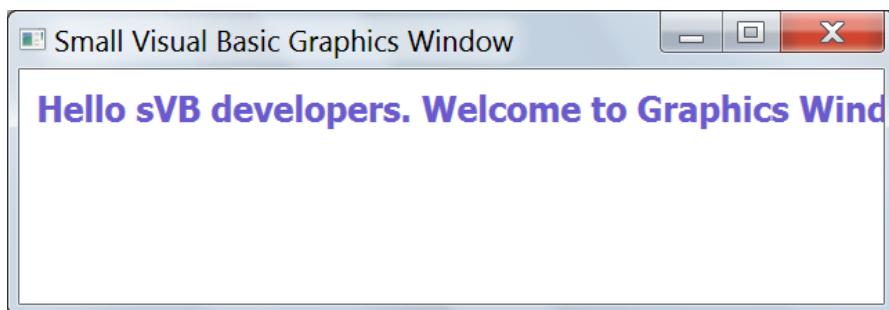
Example:

In sVB code editor, open a new sb file, and write this code:

```
GraphicsWindow.Width = 500  
GraphicsWindow.FontSize = 20
```

```
GraphicsWindow.DrawText(  
    10, 10,  
    "Hello sVB developers. Welcome to Graphics Window."  
)
```

Press F5 to run the code. This will display the graphics window, which will show this text:



You can expand the window width to see the hidden part of the text.

GraphicsWindow.DrawTriangle(

x1, y1, x2, y2, x3, y3)

Draws a triangle using the graphics window pen.

Parameters:

- **x1:**

The x co-ordinate of the first point.

- **y1:**

The y co-ordinate of the first point.

- **x2:**

The x co-ordinate of the second point.

- **y2:**

The y co-ordinate of the second point.

- **x3:**

The x co-ordinate of the third point.

- **y3:**

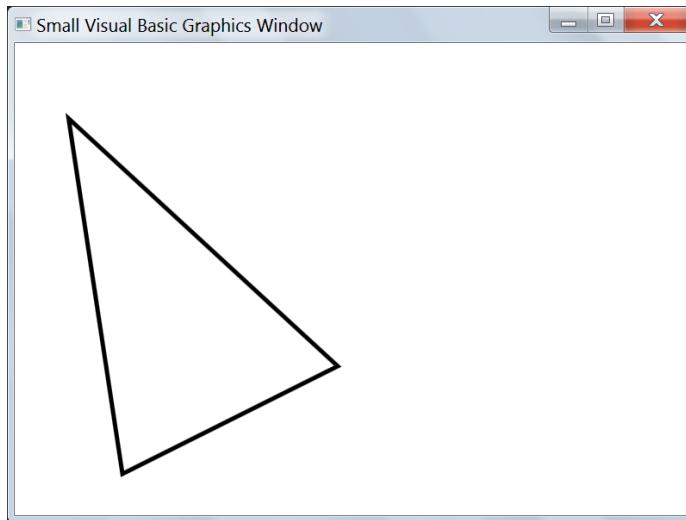
The y co-ordinate of the third point.

Example:

In sVB code editor, open a new sb file, and write this code:

```
GraphicsWindow.PenWidth = 4  
GraphicsWindow.DrawTriangle(  
    50, 70, 300, 300, 100, 400)
```

Press F5 to run the code. This will display the graphics window, which will show a black triangle:



GraphicsWindow.FillEllipse(x, y, width, height)

Draws an ellipse and fills it with the graphics window brush color. This method will not draw the outline of the ellipse, and if you want to do that, use both the FillEllipse and DrawEllipse methods (in this exact order) to fill the ellipse then draw its outline over the filled area.

Parameters:

- **x:**

The x co-ordinate of the ellipse.

- **y:**

The y co-ordinate of the ellipse.

- **width:**

The width of the ellipse.

- **height:**

The height of the ellipse.

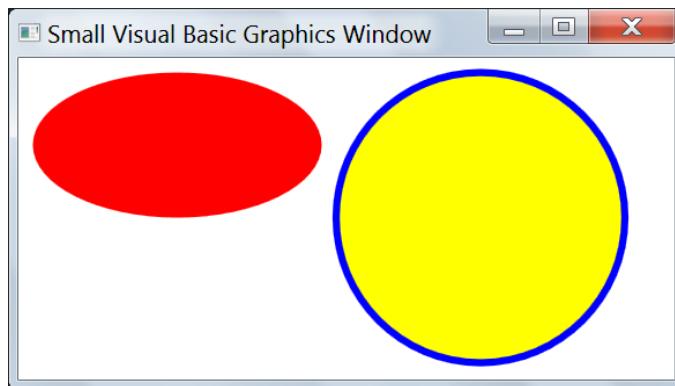
Examples:

In sVB code editor, open a new sb file, and write this code:

```
GraphicsWindow.BrushColor = Colors.Red
```

```
GraphicsWindow.FillEllipse(10, 10, 200, 100)
GraphicsWindow.BrushColor = Colors.Yellow
GraphicsWindow.PenWidth = 4
GraphicsWindow.FillEllipse(220, 10, 200, 200)
    ' Draw the circle outline
GraphicsWindow.PenColor = Colors.Blue
GraphicsWindow.PenWidth = 5
GraphicsWindow.DrawEllipse(220, 10, 200, 200)
```

Press F5 to run the code. This will display the graphics window, which will show a red ellipse without a border and a yellow circle with a blue border:



GraphicsWindow.FillGradientBackground(endColor)

Fills the Graphics Window background with a gradient brush that starts with the BackgroundColor and ends with the given color.

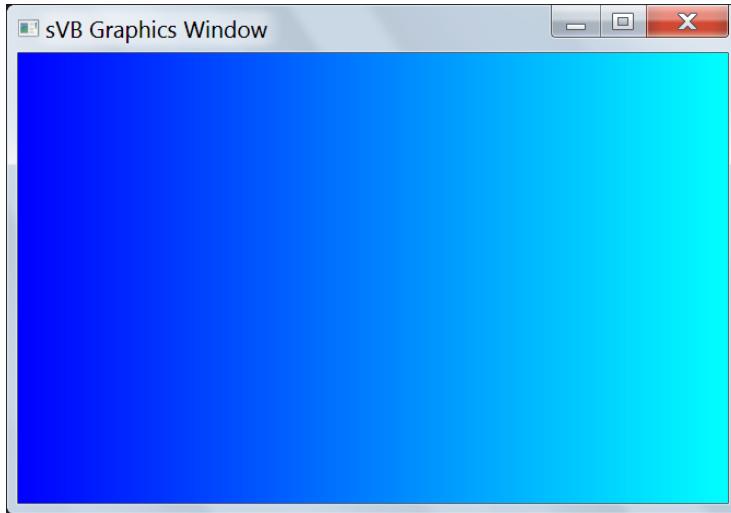
Parameter:

- **endColor:**

The end color of the gradient brush.

Example:

```
GW.BackgroundColor = Colors.Blue
GW.FillGradientBackground(Colors.Cyan)
```



⚙️ **GraphicsWindow.FillPolygon(**

xOffset, yOffset,
xScale, yScale,
pointsArr)

Fills the polygon that is represented by the given points array, with the brush of the graphics window.

You don't need to represent the exact position and size in the points array, as you can use the offset and scale parameters to change the polygon position and size. So, you should just focus on choosing the points that form the polygon outline.

Note that the area of polygon will be filled but it will not have an outlining border, so you if you need to outline it, you shod call the DrawPolygon method after calling the FillPolygon method, and pass the same arguments for them.

Parameters:

- **xOffset As Primitive:**

The horizontal offest to add to the x-coordinates of each point of the polygon.

- **yOffset As Primitive:**

The vertical offset to add to the y-coordinate of each point of the polygon.

- **xScale As Primitive:**

The factor to multiply the polygon width by.

- **yScale As Primitive:**

The factor to multiply the polygon height by.

- **pointsArr:**

An array of points representing the heads of the polygon.
Each item in this array is an array containing the x and y positions of the point.

Example:

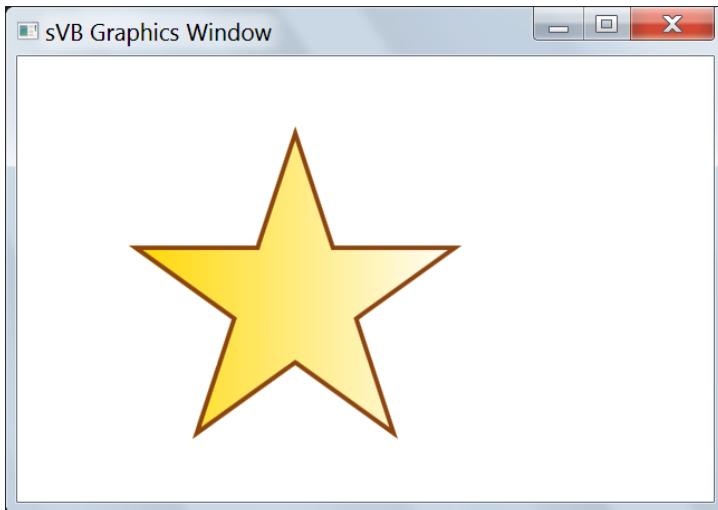
You can see this method in action in the "Draw Pentagon" project in the samples folder.

The following code fills a star with a gradient color and outlines it:

```
StarPoints = {  
    {40.00, 0.50},  
    {49.34, 28.76},  
    {79.50, 28.76},  
    {55.10, 46.24},  
    {64.35, 74.50},  
    {40.00, 57.02},  
    {15.60, 74.50},  
    {24.90, 46.24},  
    {0.50, 28.76},  
    {30.66, 28.76}  
}
```

```
GraphicsWindow.BrushColor = Colors.Gold  
GraphicsWindow.GradientEndColor = Colors.White
```

```
GW.FillPolygon(80, 50, 3, 3, StarPoints)  
GraphicsWindow.PenColor = Colors.SaddleBrown  
GraphicsWindow.PenWidth = 1.1  
GW.DrawPolygon(80, 50, 3, 3, StarPoints)
```



⚙️ **GraphicsWindow.FillRectangle(x, y, width, height)**

Draws a rectangle and fills it with the graphics window brush color.

Note that this method will not draw the outline of the rectangle, and if you want to do that, use both the FillRectangle and DrawRectangle methods (in this exact order) to fill the rectangle then draw its outline over the filled area.

Parameters:

- **x:**

The x co-ordinate of the rectangle.

- **y:**

The y co-ordinate of the rectangle.

- **width:**

The width of the rectangle.

- **height:**

The height of the rectangle.

Examples:

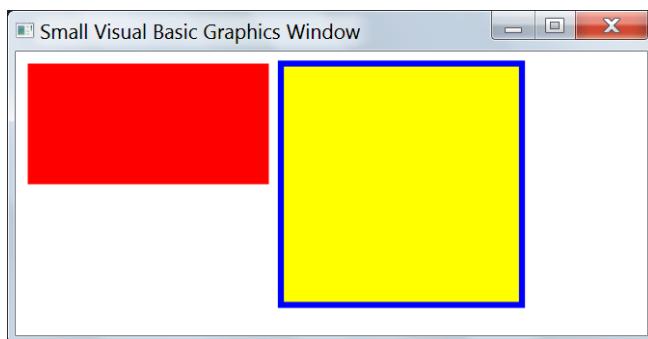
In sVB code editor, open a new sb file, and write this code:

```
GraphicsWindow.BrushColor = Colors.Red
GraphicsWindow.FillRectangle(10, 10, 200, 100)

GraphicsWindow.BrushColor = Colors.Yellow
GraphicsWindow.PenWidth = 4
GraphicsWindow.FillRectangle(220, 10, 200, 200)

' Draw the Rectangle outline
GraphicsWindow.PenColor = Colors.Blue
GraphicsWindow.PenWidth = 5
GraphicsWindow.DrawRectangle(220, 10, 200, 200)
```

Press F5 to run the code. This will display the graphics window, which will show a red rectangle without a border and a yellow square with a blue border:



💡 **GraphicsWindow.FillTriangle(x1, y1, x2, y2, x3, y3)**

Draws a triangle and fills it with the graphics window brush color.

Note that this method will not draw the outline of the triangle, and if you want to do that, use both the FillTriangle and DrawTriangle methods (in this exact order) to fill the triangle

then draw its outline over the filled area.

Parameters:

- **x1:**

The x co-ordinate of the first point.

- **y1:**

The y co-ordinate of the first point.

- **x2:**

The x co-ordinate of the second point.

- **y2:**

The y co-ordinate of the second point.

- **x3:**

The x co-ordinate of the third point.

- **y3:**

The y co-ordinate of the third point.

Examples:

In sVB code editor, open a new sb file, and write this code:

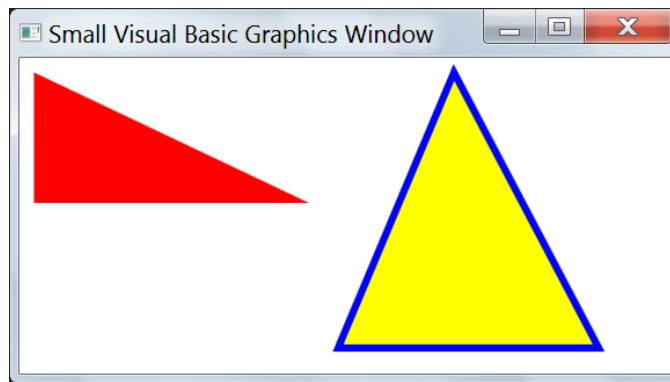
```
GraphicsWindow.BrushColor = Colors.Red
GraphicsWindow.FillTriangle(
    10, 10,
    200, 100,
    10, 100)

GraphicsWindow.BrushColor = Colors.Yellow
GraphicsWindow.PenWidth = 4
GraphicsWindow.FillTriangle(
    300, 10,
    400, 200,
    220, 200)

' Draw the triangle outline
GraphicsWindow.PenColor = Colors.Blue
```

```
GraphicsWindow.PenWidth = 5  
GraphicsWindow.DrawTriangle(  
    300, 10,  
    400, 200,  
    220, 200)
```

Press F5 to run the code. This will display the graphics window, which will show a red triangle without a border and a yellow triangle with a blue border:

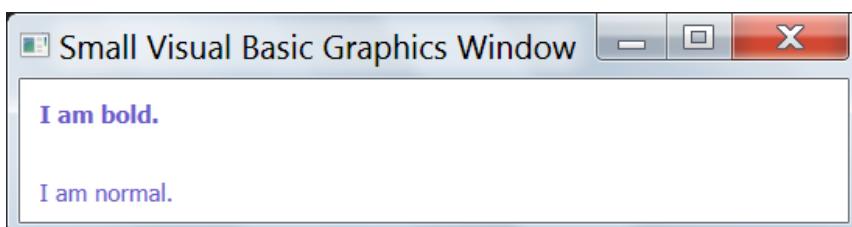


GraphicsWindow.FontBold As Boolean

Gets or sets whether or not the font to be used when drawing text on the Graphics Window, is bold.

Examples:

```
GraphicsWindow.FontBold = True  
GraphicsWindow.DrawText(10, 10, "I am bold.")  
GraphicsWindow.FontBold = False  
GraphicsWindow.DrawText(10, 50, "I am normal.")
```

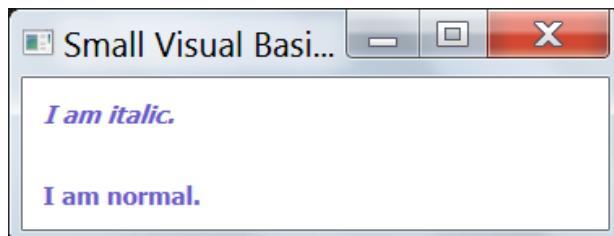


GraphicsWindow.FontItalic As Boolean

Gets or sets whether or not the font to be used when drawing text on the Graphics Window, is italic.

Examples:

```
GraphicsWindow.FontItalic = True  
GraphicsWindow.DrawText(10, 10, "I am italic.")  
GraphicsWindow.FontItalic = False  
GraphicsWindow.DrawText(10, 50, "I am normal.")
```

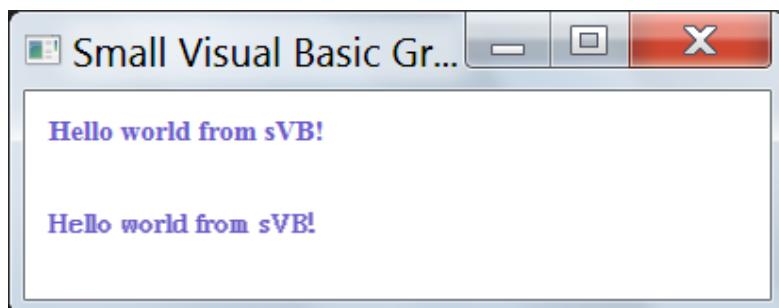


GraphicsWindow.FontName As String

Gets or sets the font name to be used when drawing text on the Graphics Window.

Examples:

```
GraphicsWindow.FontName = "Times New Roman"  
GraphicsWindow.DrawText(  
    10, 10, "Hello world from sVB!")  
GraphicsWindow.FontName = "MS PMincho"  
GraphicsWindow.DrawText(  
    10, 50, "Hello world from sVB!")
```



GraphicsWindow.FontSize As Double

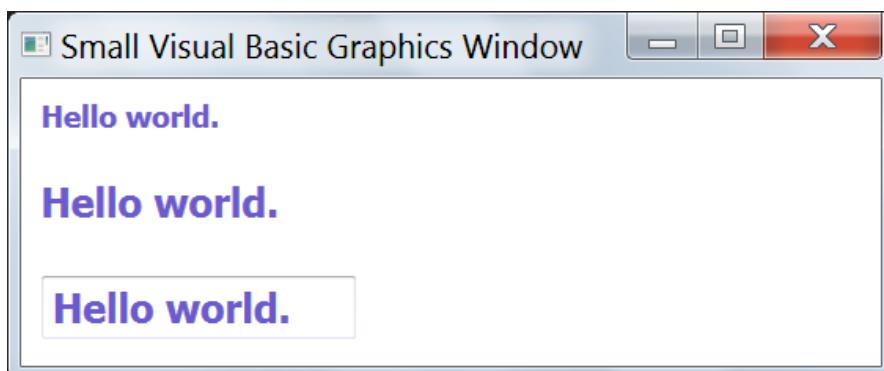
Gets or sets the font size to be used when drawing text on the Graphics Window.

Note that `GraphicsWindow.FontSize` is only 75% of the `Control.FontSize`. This means that using a 20 font size in `GraphicsWindow` will have the same effect of using a 15 font size in a `TextBox` or a label!

Small Visual Basic kept this behavior of `GraphicsWindow.FontSize` for backward computability with MS Small Basic, which uses WPF units, while sVB controls use windows forms units.

Examples:

```
GraphicsWindow.FontSize = 15
GraphicsWindow.DrawText(10, 10, "Hello world.")
GraphicsWindow.FontSize = 20
GraphicsWindow.DrawText(10, 50, "Hello world.")
' Note the result of using a 15 font in a textbox
TxtTest = Controls.AddTextBox(10, 100)
TxtTest.Text = "Hello world."
TxtTest.FontSize = 15
```



GraphicsWindow.FullScreen As Boolean

This is a Boolean property that you can use to toggle the full screen mode of the Graphics Window:

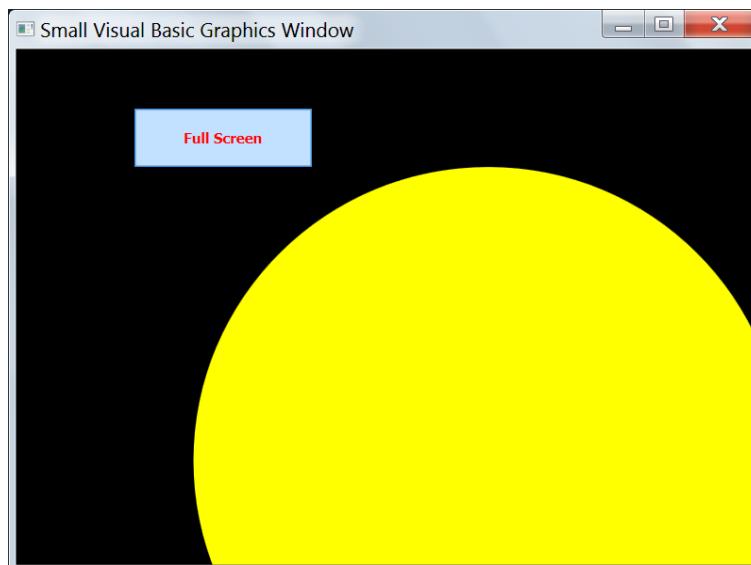
- When you set this property to True, the Graphics Window will lose its title bar and borders, and it will fill the entire area of the screen.
- When you set this property to False, the Graphics Window will exit the full screen mode restore its normal position and size, and display the title bar and its borders.

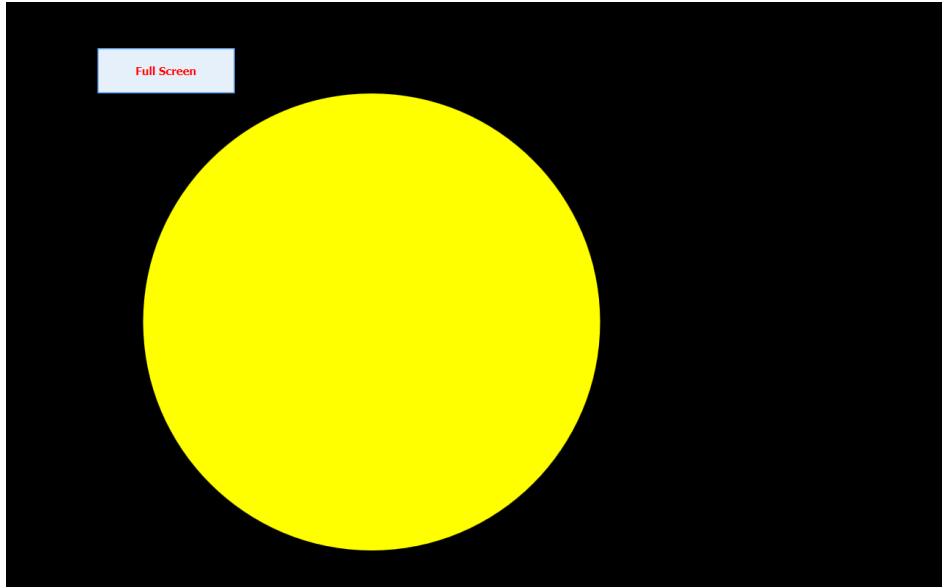
Note that the user can press Alt+F4 to close the window while it is in the full screen mode (or even in its normal mode).

The user also can press F11 to toggle the full screen mode, so he can manually display the graphics window in the full mode or restore its normal state by just pressing F11.

Example:

This code will display a toggle button and a circle on the graphics window. clicking the toggle button will toggle the full screen mode of the window:





```
GraphicsWindow.BackgroundColor = Colors.Black
GraphicsWindow.BrushColor = Colors.Yellow
GraphicsWindow.FillEllipse(150, 100, 500, 500)
Tb = Controls.AddToggleButton("Full Screen", 100,
50, 150, 50)
Tb.ForeColor = Colors.Red
Tb.IsFlat = True
Tb.OnClick = ToggleFS

Sub ToggleFS()
    If GraphicsWindow.FullScreen Then
        GraphicsWindow.FullScreen = False
    Else
        GraphicsWindow.FullScreen = True
    EndIf
EndSub
```

GraphicsWindow.GradientEndColor As Color

Gets or sets the end color for the gradient brush. Use the BrushColor property to set the start color of this gradient brush. The default value is Colors.None, which means that the BrushColor will be used alone to create a solid brush. Som you should remember to set this property to Colors.None after filling a shape with a gradient brush and you want to fill the next shape with a soild brush.

Example:

This code fills a rectangle and a triangle shape with gradient brushes, then fills a circle with solid brush, and finally fills a composite geometric shape with a gradient brush:

```
' Fill a rectangle with a gradient brush
GW.BrushColor = Colors.Blue      'start color
GW.GradientEndColor = Colors.AliceBlue 'end color
GW.FillRectangle(30, 20, 100, 100)

' Fill a triangle shape with a gradient brush
GW.BrushColor = Colors.Orange
GW.GradientEndColor = Colors.Yellow
Shapes.AddTriangle(170, 120, 170, 20, 300, 120)

' Fill a circle shape with a solid brush
GW.BrushColor = Colors.Red
GW.GradientEndColor = Colors.None ' disable it
GW.FillEllipse(30, 130, 100, 100)

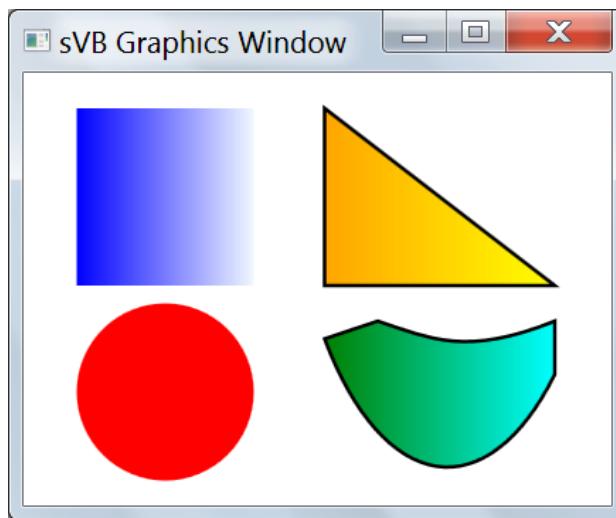
' Create a geometric path
GeometricPath.CreatePath()
GeometricPath.CreateFigure(170, 150, True)
GeometricPath.AddArcSegment(300, 170, 90, 200,
    0, False, False, True)
GeometricPath.AddLineSegment(300, 140, True)
```

```

GeometricPath.AddBezierSegment(250, 160, 230, 150,
200, 140, True)

' Add the geometric path to the shapes
' and fill it with a gradient brush
GW.BrushColor = Colors.Green
GW.GradientEndColor = Colors.Cyan
Shapes.AddGeometricPath()

```



GraphicsWindow.GetColorFromRGB(red, green, blue) As Color

Constructs a color given the Red, Green and Blue values.

It will be shorter to use the [Color.FromRGB](#) method instead.

Parameters:

- **red:**

The red component of the Color (0-255).

- **green:**

The green component of the color (0-255).

- **blue:**

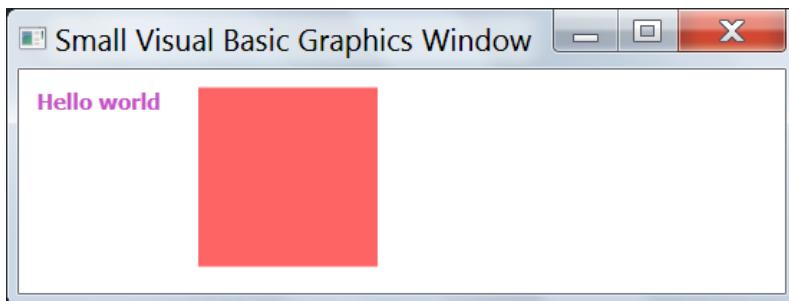
The blue component of the color (0-255).

Returns:

A custom color that can be used to set the brush or pen color.

Examples:

```
c1 = GraphicsWindow.GetColorFromRGB(200, 80, 200)
c2 = Color.FromRGB(255, 100, 100)
GraphicsWindow.BrushColor = c1
GraphicsWindow.DrawText(10, 10, "Hello world")
GraphicsWindow.BrushColor = c2
GraphicsWindow.FillRectangle(100, 10, 100, 100)
```



⚙️ **GraphicsWindow.GetPixel(x, y) As Color**

Gets the color of the pixel at the specified x and y co-ordinates.

Parameters:

- **x:**

The x co-ordinate of the pixel.

- **y:**

The y co-ordinate of the pixel.

Returns:

The color of the pixel.

GraphicsWindow.GetRandomColor() As Color

Gets a valid random color, by composing a custom color from random RGB components.

Note: You may find the [Color.GetRandomColor](#) method or the Colors.Rand property useful in some situations, because they select a random color from a list of 139 basic colors, so you will have a better chance to get certain colors like a pure white or a pure yellow, which can't probably happen when you use the GW.GetRandomColor, because it selects a color from more than 16 million possible color shades!

Returns:

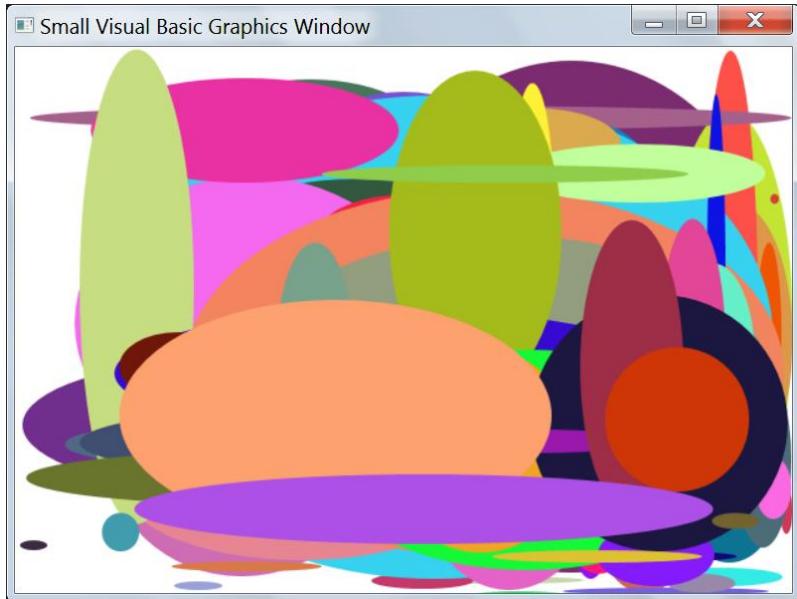
A valid random color.

Examples:

The following code will draw infinite number of random colors ellipses on random locations with random sizes (ensuring that the minimum width and height are 5):

```
W = GraphicsWindow.Width
H = GraphicsWindow.Height

While True
    GraphicsWindow.BrushColor =
        GraphicsWindow.GetRandomColor()
    X = Math.GetRandomNumber(W)
    Y = Math.GetRandomNumber(H)
    GraphicsWindow.FillEllipse(
        X,
        Y,
        5 + Math.GetRandomNumber(W - X - 5),
        5 + Math.GetRandomNumber(H - Y - 5)
    )
    Program.Delay(150)
Wend
```



🎨 **GraphicsWindow.Height As Double**

Gets or sets the Height of the graphics window.

Note that the Graphics window by default will be shown in its maximized state, but when you change its width or height it will be shown in its normal state to have the new width or height.

⚙️ **GraphicsWindow.Hide()**

Hides the graphics window. You can use the Show method to display it again.

🎨 **GraphicsWindow.IsClosed As Boolean**

Returns True if the graphics window is closed, otherwise False.

See the example provided by the [AutoShow](#) method.

GraphicsWindow.KeyDown

This event is raised when a key is pressed down on the keyboard.

Examples:

This code will show a message box that displays the name of each key you press on the keyboard:

```
GraphicsWindow.KeyDown = KeyDown  
  
Sub KeyDown()  
    GraphicsWindow.ShowMessage(  
        GraphicsWindow.LastKey, "Key")  
EndSub
```

Note that the graphics window is actually a [form](#) with the name "GraphicsWindow", so you can use this simple trick to deal with it as a normal form:

```
GraphicsWindow.Show()  
GWForm = "GraphicsWindow"  
GWForm.OnKeyDown = KeyDown  
  
Sub KeyDown()  
    GraphicsWindow.ShowMessage(  
        Event.LastKeyName, "Key")  
EndSub
```

GraphicsWindow.KeyUp

This event is raised when a key is released on the keyboard.

GraphicsWindow.LastKey As String

Gets the string name of the last key that was pressed or released like "Escape", "Left" and "A".

You should avoid using this property because you can easily mistype the key name, and instead you should use the

[Event.LastKey](#), which allows you to use the values of the Keys enumerator to represent the key, which allows the sVB code editor to show you the names of the keys in the auto completion list, so you can easily choose one.

GraphicsWindow.LastText As String

Gets the last text that was entered on the Graphics Window.

GraphicsWindow.Left As Double

Gets or sets the Left Position of the graphics window.

GraphicsWindow.MouseDown

This event is raised when the mouse button is clicked down.

GraphicsWindow.MouseMove

This event is raised when the mouse is moved around.

GraphicsWindow.MouseUp

This event is raised when the mouse button is released.

GraphicsWindow.MouseX As Double

Gets the x-position of the mouse relative to the Graphics Window.

GraphicsWindow.MouseY As Double

Gets the y-position of the mouse relative to the Graphics Window.

GraphicsWindow.PenColor As Color

Gets or sets the color of the pen used to draw shapes on the Graphics Window using the DrawLine, DrawTriangle, DrawEllipse, and DrawRectangle methods.

GraphicsWindow.PenWidth As Double

Gets or sets the width of the pen used to draw shapes on the Graphics Window using the DrawLine, DrawTriangle, DrawEllipse, DrawRectangle, DrawText and DrawBoundText methods.

GraphicsWindow.SetPixel(x, y, color)

Draws the pixel specified by the x and y co-ordinates using the specified color.

Parameters:

- **x:**

The x co-ordinate of the pixel.

- **y:**

The y co-ordinate of the pixel.

- **color:**

The color of the pixel to set.

Examples:

The following code draws the two curves of the Sin and Cos functions. Note that:

1. We scale the y value by 50 so we can see the curve on the screen.
2. We also negate the y value, because the 0 value on the screen is on the top, and value increases while we go down, which is the reverse direction of the y axis in math.

3. We also add 150 to the value of y to move the curve down to the middle of the screen.
4. We draw the x-axis in the middle of the screen, so, we consider that the first half of the screen is the negative x values, hence we add half of the width to the x value to translate it to the screen co-ordinates.
5. We draw the Sin function with red, and the Cos function with blue.

This is the code:

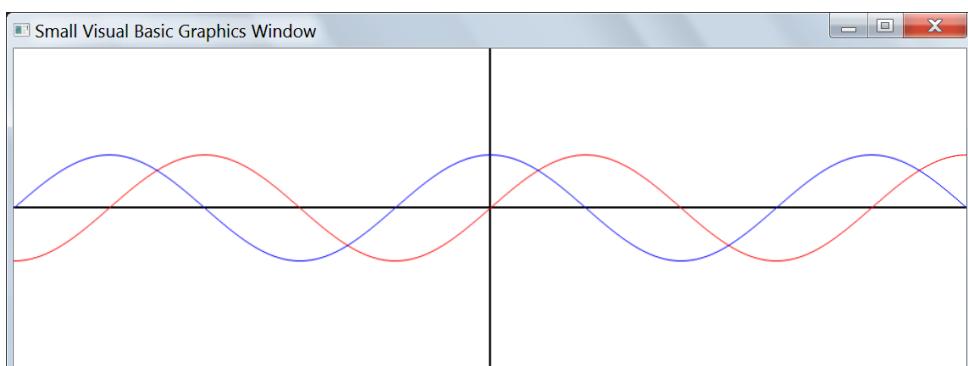
```

GraphicsWindow.Width = 900
GraphicsWindow.Height = 300
GraphicsWindow.DrawLine(0, 150, 900, 150)
w2 = Math.Round(GraphicsWindow.Width / 2)
GraphicsWindow.DrawLine(w2, 0, w2, 300)
R = Math.Pi / 180
For X = -w2 To w2
    GraphicsWindow.SetPixel(
        X + w2,
        150 - 50 * Math.Sin(X * R),
        Colors.Red)
    GraphicsWindow.SetPixel(
        X + w2,
        150 - 50 * Math.Cos(X * R),
        Colors.Blue)

```

[Next](#)

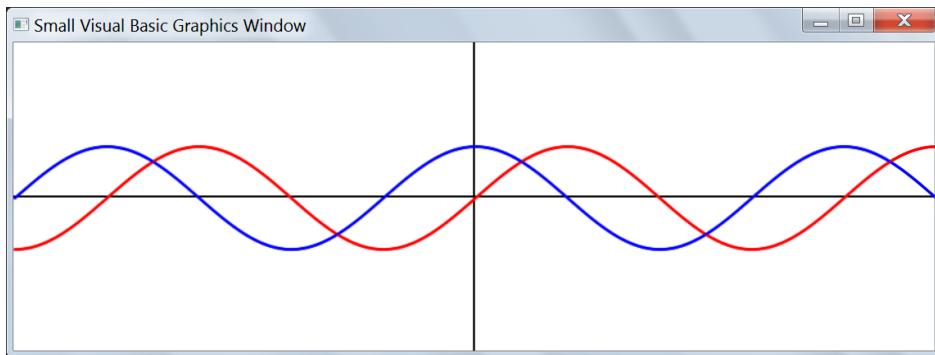
And when you run it, you will get this:



Note that you can make the curves thicker by drawing very small circles with width And height of 2 Or 3 points instead of dots. Change the loop in the above code to:

```
For X = -H To H
    GraphicsWindow.BrushColor = Colors.Red
    GraphicsWindow.FillEllipse(
        X + H,
        150 - 50 * Math.Sin(X * R),
        3, 3)
    GraphicsWindow.BrushColor = Colors.Blue
    GraphicsWindow.FillEllipse(
        X + H,
        150 - 50 * Math.Cos(X * R),
        3, 3)
```

[Next](#)



GraphicsWindow.Show()

Shows the graphics window to enable interactions with it.

Note that using any method or property of the graphics window will show it, so you don't actually need to call the Show method unless you used the Hide method to hide it, so you will need to show it again.

GraphicsWindow.ShowMessage(message, title)

Displays a message box to the user that displayed the supplied text and an OK button to close the window.

Note that you can show a Yes/No/Cancel message box and interact with the user choice by using the [Dialogs.MsgBox](#) method.

Parameters:

- **message:**

The text to be displayed on the message box.

- **title:**

The title for the message box.

Example:

```
GraphicsWindow.ShowMessage(  
    "Welcome to sVB!", "Hello")
```

GraphicsWindow.TextInput

This event is raised when text is entered on the graphics window.

GraphicsWindow.Title As String

Gets or sets the title for the graphics window.

GraphicsWindow.Top As Double

Gets or sets the top position of the graphics window.

GraphicsWindow.Topmost As Boolean

Gets or sets whether or not the Graphics Window is the topmost window that always appears on top of all other desktop windows even when it is not the active window.

GraphicsWindow.Width As Double

Gets or sets the Width of the graphics window.

Note that the Graphics window by default will be shown in its maximized state, but when you change its width or height it will be show in its normal state to have the new width or height.

The ImageList Type

This class helps to load and store images in memory.
The ImageList type has these members:

ImageList.GetHeightOfImage(imageName) As Double

Gets the height of the stored image.

Parameter:

- **imageName:**

The name of the image in memory. You get this name from the ImageList.LoadImage method.

Returns:

The height of the specified image.

ImageList.GetWidthOfImage(imageName) As Double

Gets the width of the stored image.

Parameter:

- **imageName:**

The name of the image in memory. You get this name from the ImageList.LoadImage method.

Returns:

The width of the specified image.

ImageList.LoadImage(fileNameOrUrl) As String

Loads an image from a file or the internet into memory.

Parameter:

- **fileNameOrUrl:**

The file name to load the image from. This could be a local file or a URL to an internet location.

If the file exists in the same folder of your application, you can use a relative path like "image1.jpg" with no need to mention the application directory path.

This also works if the image exists in a subfolder inside your application's folder, hence you can use a relative path starting with the subfolder like "images\image1.jpg".

Returns:

Returns the name of the image that was loaded. You can use this name to draw the image on the graphics window.

Examples:

In sVB code editor, open a new sb file, and write this code:

```
path = "%PUBLIC%\Pictures\Sample Pictures\Desktop.jpg"
Image1 = ImageList.LoadImage(path)
GraphicsWindow.DrawResizedImage(
    Image1, 0, 0, 300, 150)
GraphicsWindow.DrawImage(Image1, 0, 150)
```

Press F5 to run the project. The GraphicsWindow will look like this:



The Keyboard Type

Contains info about the state of the keyboard keys.

To see this type in action, see the 'Pressed keys` project in the samples folder.

The Keyboard type has these members:

Keyboard.AltPressed As Boolean

Returns True if the Alt key is pressed, otherwise False.

Keyboard.CapsLockOn As Boolean

Returns True if the Caps Lock key is on, otherwise False.

Keyboard.CtrlPressed As Boolean

Returns True if the Control key is pressed, otherwise False.

Keyboard.InsertOn As Boolean

Returns True if the Insert key is on, otherwise False.

Keyboard.LastKey

Returns the last Key pressed on the keyboard.

Use the [Keys](#) enum members to check they key.

Example:

```
If Keyboard.LastKey = Keys.A Then  
    ' Do something  
EndIf
```

Keyboard.LastKeyName As String

Returns the name of the last key pressed on the keyboard.

Keyboard.LastTextInput As String

Returns the last text that was about to be written to the TextBox

Keyboard.NumLockOn As Boolean

Returns True if the Num Lock key is on, otherwise False.

Keyboard.ScrollOn As Boolean

Returns True if the Scroll Lock key is on, otherwise False.

Keyboard.ShiftPressed As Boolean

Returns True if the Shift key is pressed, otherwise False.

Keyboard.WinPressed As Boolean

Returns True if the Win key is pressed, otherwise False.

The Keys Type

Specifies the possible key values on a keyboard.

The Keys type has these members:

Property	Retuned Key
 A	A
 AbntC1	ABNT_C1 (Brazilian)
 AbntC2	ABNT_C2 (Brazilian)
 Add	Add
 Apps	Application (Microsoft Natural Keyboard)
 Attn	ATTN
 B	B
 Back	Backspace
 BrowserBack	Browser Back
 BrowserFavorites	Browser Favorites
 BrowserForward	Browser Forward
 BrowserHome	Browser Home
 BrowserRefresh	Browser Refresh
 BrowserSearch	Browser Search
 BrowserStop	Browser Stop
 C	C
 Cancel	Cancel
 Capital	Caps Lock
 CapsLock	Caps Lock
 Clear	Clear
 CrSel	CRSEL
 D	D
 D0	0 (zero)
 D1	1 (one)
 D2	2

 D3	3
 D4	4
 D5	5
 D6	6
 D7	7
 D8	8
 D9	9
 DbeAlphanumeric	DBE_ALPHANUMERIC
 DbeCodeInput	DBE_CODEINPUT
 DbeDbcsChar	DBE_DBCSCHAR
 DbeDetermineString	DBE_DETERMINESTRING
 DbeEnterDialogConversionMode	DBE_ENTERDLGCONVERSIONMODE
 DbeEnterIMEConfigureMode	DBE_ENTERIMECONFIGMODE
 DbeEnterWordRegisterMode	DBE_ENTERWORDREGISTERMODE
 DbeFlushString	DBE_FLUSHSTRING
 DbeHiragana	DBE_HIRAGANA
 DbeKatakana	DBE_KATAKANA
 DbeNoCodeInput	DBE_NOCODEINPUT
 DbeNoRoman	DBE_NOROMAN
 DbeRoman	DBE_ROMAN
 DbeSbcsChar	DBE_SBCSCHAR
 DeadCharProcessed	The key is used with another key to create a single combined character.
 Decimal	Decimal
 Delete	Delete
 Divide	Divide
 Down	Down Arrow
 E	E
 End	End
 Enter	Enter
 EraseEof	ERASE EOF
 Escape	ESC
 Execute	Execute

 ExSel	EXSEL
 F	F
 F1	F1
 F10	F10
 F11	F11
 F12	F12
 F13	F13
 F14	F14
 F15	F15
 F16	F16
 F17	F17
 F18	F18
 F19	F19
 F2	F2
 F20	F20
 F21	F21
 F22	F22
 F23	F23
 F24	F24
 F3	F3
 F4	F4
 F5	F5
 F6	F6
 F7	F7
 F8	F8
 F9	F9
 FinalMode	IME Final mode
 G	G
 H	H
 HangulMode	IME Hangul mode
 HanjaMode	IME Hanja mode
 Help	Help
 Home	Home

 I	I
 ImeAccept	IME Accept
 ImeConvert	IME Convert
 ImeModeChange	IME Mode change request
 ImeNonConvert	IME NonConvert
 ImeProcessed	special key masking the real key being processed by an IME
 Insert	Insert
 J	J
 JunjaMode	IME Junja mode
 K	K
 KanaMode	IME Kana mode
 KanjiMode	IME Kanji mode
 L	L
 LaunchApplication1	Launch Application1
 LaunchApplication2	Launch Application2
 LaunchMail	Launch Mail
 Left	Left Arrow
 LeftAlt	left ALT
 LeftCtrl	left CTRL
 LeftShift	left Shift
 LineFeed	Line feed
 LWin	left Windows logo key (Microsoft Natural Keyboard)
 M	M
 MediaNextTrack	Media Next Track
 MediaPlayPause	Media Play Pause
 MediaPreviousTrack	Media Previous Track
 MediaStop	Media Stop
 Multiply	Multiply
 N	N
 Next	Page Down
 NoName	Reserved for future use

 None	No key pressed
 NumLock	Num Lock
 NumPad0	0 on the numeric keypad
 NumPad1	1 on the numeric keypad
 NumPad2	2 on the numeric keypad
 NumPad3	3 on the numeric keypad
 NumPad4	4 on the numeric keypad
 NumPad5	5 on the numeric keypad
 NumPad6	6 on the numeric keypad
 NumPad7	7 on the numeric keypad
 NumPad8	8 on the numeric keypad
 NumPad9	9 on the numeric keypad
 O	O
 Oem1	OEM 1
 Oem102	OEM 102
 Oem2	OEM 2
 Oem3	OEM 3
 Oem4	OEM 4
 Oem5	OEM 5
 Oem6	OEM 6
 Oem7	OEM 7
 Oem8	OEM 8
 OemAttn	OEM ATTN
 OemAuto	OEM AUTO
 OemBackslash	OEM Backslash
 OemBackTab	OEM BACKTAB
 OemClear	OEM Clear
 OemCloseBrackets	OEM Close Brackets
 OemComma	OEM Comma
 OemCopy	OEM COPY
 OemEnlw	OEM ENLW
 OemFinish	OEM FINISH
OemMinus	OEM Minus

 OemOpenBrackets	OEM Open Brackets
 OemPeriod	OEM Period
 OemPipe	OEM Pipe
 OemPlus	OEM Addition
 OemQuestion	OEM Question
 OemQuotes	OEM Quotes
 OemSemicolon	OEM Semicolon
 OemTilde	OEM Tilde
 P	P
 Pa1	PA1
 PageDown	Page Down
 PageUp	Page Up
 Pause	Pause
 Play	PLAY
 Print	Print
 PrintScreen	Print Screen
 Prior	Page Up
 Q	Q
 R	R
 Return	Return
 Right	Right Arrow
 RightAlt	Right ALT
 RightCtrl	Right CTRL
 RightShift	Right Shift
 RWin	Right Windows logo key (Microsoft Natural Keyboard)
 S	S
 Scroll	Scroll Lock
 Select	Select
 SelectMedia	Select Media
 Separator	Separator
 Sleep	Computer Sleep
 Snapshot	Print Screen

 Space	Spacebar
 Subtract	Subtract
 System	A special key masking the real key being processed as a system key.
 T	T
 Tab	Tab
 U	U
 Up	Up Arrow
 V	V
 VolumeDown	Volume Down
 VolumeMute	Volume Mute
 VolumeUp	Volume Up
 W	W
 X	X
 Y	Y
 Z	Z
 Zoom	ZOOM

The Label Type

Represents a Label control, that can show a text, a hyper link, an image or a graphic to the user.

- Use the Text property to set the text displayed by the label.
- Use the Append... methods to add a formatted text to the label.
- Use the AppendLink method to add a hyper link to the label.
- Use the Image property to load an image from a file and display it in the label.
- Use the Add... methods to add geometric shapes to the label.

You can use the form designer to add a label to the form by dragging it from the toolbox.

It is also possible to use the [Form.AddLabel](#) or the [Controls.AddLabel](#) methods to create a new label and add it to the form or the graphics window at runtime.

Note that the Label control inherits all the properties, methods and events of the [Control](#) type.

Besides, the Label control has some new members that are listed below:

Label.AddEllipse(width, height, penColor, penWidth, brushColor)

Draws an ellipse shape with the specified width and height on the current label.

Note that the label can contain only one shape, but you can add many shapes to the [GeometricPath](#) then use the

[Label.AddGeometricPath](#) method to add them as a combined shape to the label.

Parameters:

- **width:**

The width of the ellipse shape.

- **height:**

The height of the ellipse shape.

- **penColor:**

The color used to draw the shape outline.

Use Colors.None to erase the outline.

Use Colors.transparent to hide the outline.

- **penWidth:**

The width of the shape outline.

Use 0 to erase the outline.

- **brushColor:**

The color used to fill the shape background.

Use Colors.None to erase the shape background.

Use Colors.transparent to hide the shape background.

Example:

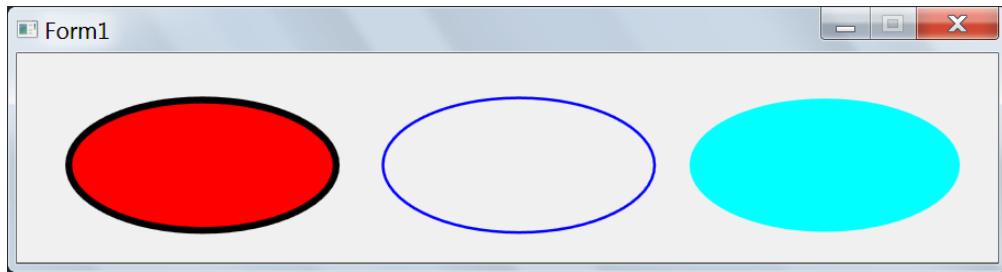
Add three labels on the form, and write this code in the global area:

```
Label1.AddEllipse(  
    200, 100,  
    Colors.Black, 5,  
    Colors.Red  
)
```

```

Label2.AddEllipse(
    200, 100,
    Colors.Blue, 2,
    Colors.None
)
Label3.AddEllipse(
    200, 100,
    Colors.Transparent, 3,
    Colors.Cyan
)

```



⌚ **Label.AddGeometricPath(penColor, penWidth, brushColor)**

Draws the shape defined by GeometricPath on the label.

Parameters:

- **penColor:**

The color used to draw the shape outline.

Use Colors.None to erase the outline.

Use Colors.transparent to hide the outline.

- **penWidth:**

The width of the shape outline.

Use 0 to erase the outline.

• brushColor:

The color used to fill the shape background.

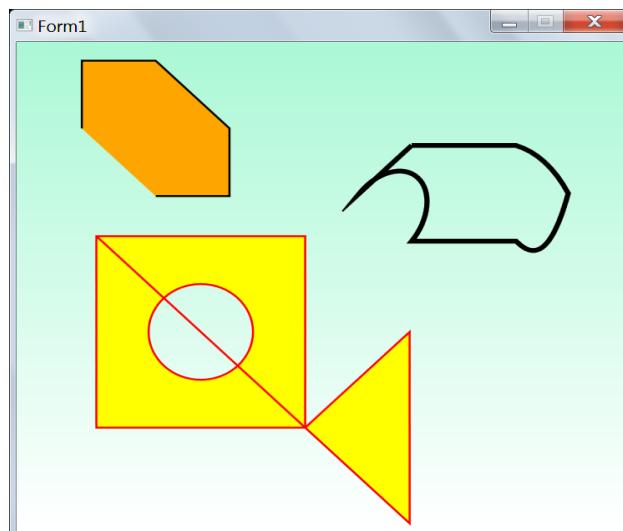
Use Colors.None to erase the shape background.

Use Colors.transparent to hide the shape background.

Example:

You can see this method in action in the "Geometric Path 2" project in the samples folder. It is used in the CreateFigure1, CreateFigure2 and CreateFigure3 sub routines to draw the geometric path on each label. For example, this is the code of the CreateFigure3, which draws the yellow figure you see in the following picture:

```
Sub CreateFigure3()
    GeometricPath.CreatePath()
    GeometricPath.AddLine(0, 0, 300, 300)
    GeometricPath.AddRectangle(0, 0, 200, 200)
    GeometricPath.AddEllipse(50, 50, 100, 100)
    GeometricPath.AddTriangle(
        200, 200,
        300, 100,
        300, 300)
    Label13.AddGeometricPath(
        Colors.Red, 2, Colors.Yellow)
EndSub
```



Label.AddLine(x1, y1, x2, y2, penColor, penWidth)

Draws a line between the specified two points on the current label.

Note that the label can contain only one shape, but you can add many shapes to the [GeometricPath](#) then use the [Label.AddGeometricPath](#) to add them as a combined shape to the label.

Parameters:

- **x1:**

The x co-ordinate of the first point.

- **y1:**

The y co-ordinate of the first point."

- **x2:**

The x co-ordinate of the second point.

- **y2:**

The y co-ordinate of the second point.

- **penColor:**

The color used to draw the shape outline.

Use Colors.None to erase the outline.

Use Colors.transparent to hide the outline.

- **penWidth:**

The width of the shape outline.

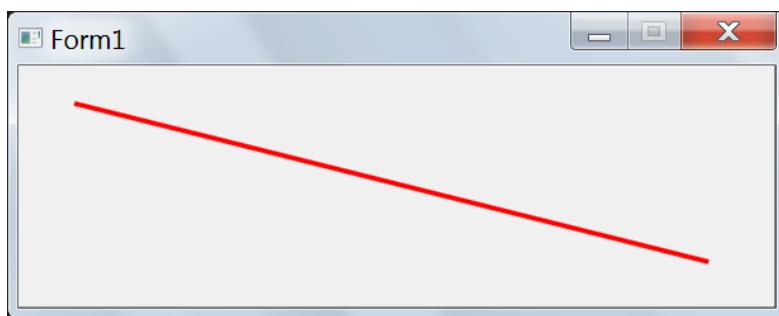
Use 0 to erase the outline.

Example:

Add a label on the form, and write this code in the global area:

```
Label1.Width = 400  
Label1.Height = 100  
Label1.AddLine(  
    0, 0,  
    Label1.Width, Label1.Height,  
    Colors.Red, 3  
)
```

Press F5 to run the project. This is what you will see:



⚙️ **Label.AddPolygon(pointsArr, penColor, penWidth, brushColor)**

Draws a polygon shape represented by the given points array on the current label.

Note that the label can contain only one shape, but you can add many shapes to the [GeometricPath](#) then use the [Label.AddGeometricPath](#) to add them as a combined shape to the label.

Parameters:

- **pointsArr:**

An array of points representing the heads of the polygon.
Each item in this array is an array containing the x and y positions of the point.

- **penColor:**

The color used to draw the shape outline.

Use Colors.None to erase the outline.

Use Colors.transparent to hide the outline.

- **penWidth:**

The width of the shape outline.

Use 0 to erase the outline.

- **brushColor:**

The color used to fill the shape background.

Use Colors.None to erase the shape background.

Use Colors.transparent to hide the shape background.

Example:

You can see this method in action in the "Draw Pentagon 2" project in the samples folder. This is the code that draws the star:

```
Label1.AddPolygon(  
    {  
        {0, 100},  
        {76.5, 100},  
        {100, 25},  
        {123.5, 100},  
        {200, 100},  
        {140, 142.5},  
        {160, 200},  
        {100, 145},  
        {40, 200},  
        {62, 142.5}  
    },  
    Colors.Black, 3, Colors.Red  
)
```



⚙️**Label.AddRectangle(width, height, penColor, penWidth, brushColor)**

Draws a rectangle shape with the specified width and height on the current label.

Note that the label can contain only one shape, but you can add many shapes to the [GeometricPath](#) then use the [Label.AddGeometricPath](#) to add them as a combined shape to the label.

Parameters:

- **width:**

The width of the rectangle shape.

- **height:**

The height of the rectangle shape.

- **penColor:**

The color used to draw the shape outline.

Use Colors.None to erase the outline.

Use Colors.transparent to hide the outline.

- **penWidth:**

The width of the shape outline.

Use 0 to erase the outline.

- **brushColor:**

The color used to fill the shape background.

Use Colors.None to erase the shape background.

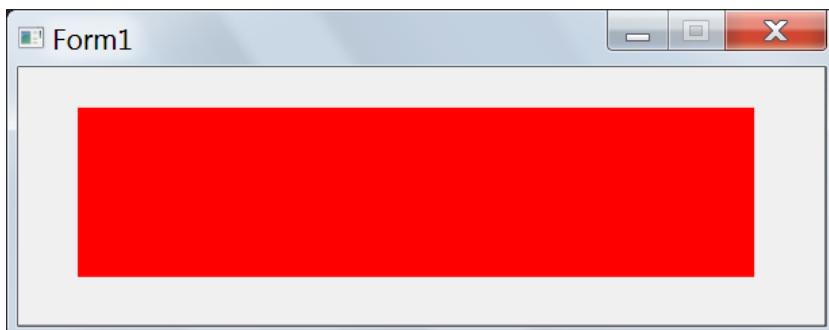
Use Colors.transparent to hide the shape background.

Example:

Add a label on the form, and write this code in the global area:

```
Label1.Width = 400  
Label1.Height = 100  
Label1.AddRectangle(  
    Label1.Width, Label1.Height,  
    Colors.None, 0,  
    Colors.Red  
)
```

Press F5 to run the project. This is what you will see:



Label.AddTriangle(x1, y1, x2, y2, x3, y3, penColor, pnWidth, brushColor)

Draws a triangle shape represented by the specified points on the current label.

Note that the label can contain only one shape, but you can add many shapes to the [GeometricPath](#) then use the [Label.AddGeometricPath](#) to add them as a combined shape to the label.

Parameters:

- **x1:**

The x co-ordinate of the first point.

- **y1:**

The y co-ordinate of the first point.

- **x2:**

The x co-ordinate of the second point.

- **y2:**

The y co-ordinate of the second point.

- **x3:**

The x co-ordinate of the third point.

- **y3:**

The y co-ordinate of the third point.

- **penColor:**

The color used to draw the shape outline.

Use Colors.None to erase the outline.

Use Colors.transparent to hide the outline.

- **penWidth:**

The width of the shape outline.

Use 0 to erase the outline.

- **brushColor:**

The color used to fill the shape background.

Use Colors.None to erase the shape background.

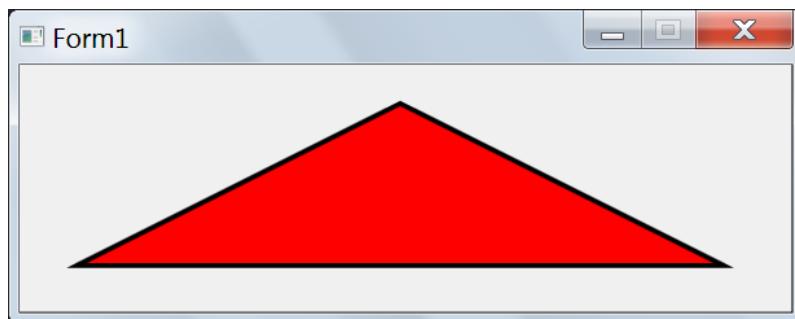
Use Colors.transparent to hide the shape background.

Example:

Add a label on the form, and write this code in the global area:

```
Label1.Width = 400  
Label1.Height = 100  
Label1.AddTriangle(  
    Label1.Width / 2, 0,  
    0, Label1.Height,  
    Label1.Width, Label1.Height,  
    Colors.Black, 3,  
    Colors.Red  
)
```

Press F5 to run the project. This is what you will see:



⌚Label.Append(text)

Adds the given text at the end of the current label.

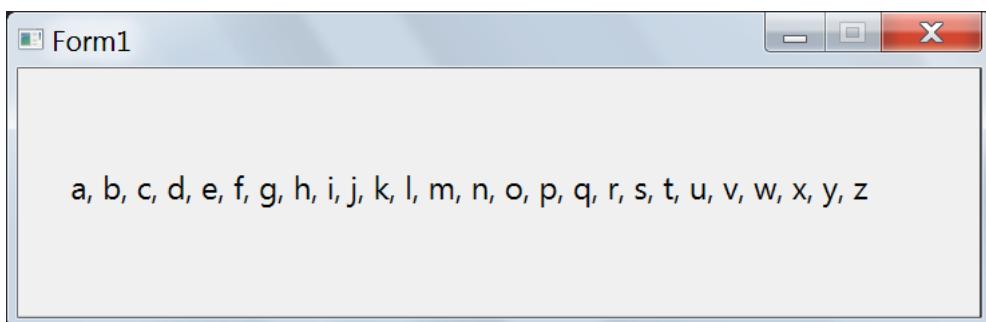
Parameter:

- **text:**

The text to add to the label.

Example:

```
Label1.Text = ""  
For C = "a" To "y"  
    Label1.Append(Chars.GetCharacter(C))  
    Label1.Append(", ")  
Next  
Label1.Append("z")  
Label1.FitContentWidth()
```



⌚Label.AppendBold(text)

Adds the given text at the end of the current label with a bold font.

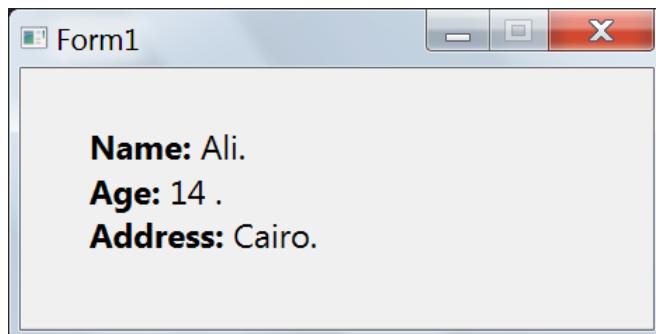
Parameter:

- **text:**

The text to add to the label.

Example:

```
Label1.Text = ""  
Label1.AppendBold("Name: ")  
Label1.AppendLine("Ali.")  
Label1.AppendBold("Age: ")  
Label1.AppendLine("14 .")  
Label1.AppendBold("Address: ")  
Label1.AppendLine("Cairo.")  
Label1.FitContentSize()
```



Label.AppendBoldItalic(text)

Adds the given text at the end of the current label with a bold and italic font.

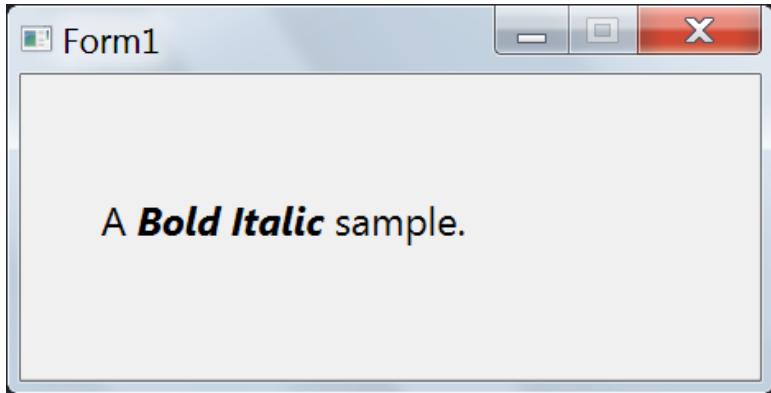
Parameter:

- **text:**

The text to add to the label.

Example:

```
Label1.Text = "A "  
Label1.AppendBoldItalic("Bold Italic")  
Label1.Append(" sample.")  
Label1.FitContentSize()
```



Label.AppendBoldItalicLink(text, url)

Adds the given text at the end of the current label with a bold and italic font, and formats it as a hyper link that opens the given url.

Parameters:

- **text:**

The text to add to the label.

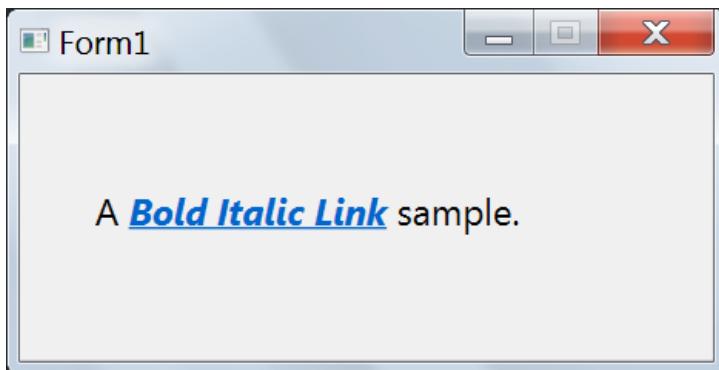
- **url:**

The address to navigate to. The value of this parameter can be:

- an empty string "" to view a normal text (not a link).
- a url to an internet or a network address.
- "about:blank", to open a new empty page in the browser.
- a local application path (an exe file) to open it.
- a file path (like a txt file), to make the windows open it with the suitable application (like notepad).
- a folder path to open it in the file explorer.

Example:

```
Label1.Text = "A "
Label1.AppendBoldItalicLink(
    "Bold Italic Link", "about:blank")
Label1.Append(" sample.")
Label1.FitContentSize()
```



Label.AppendBoldLink(text, url)

Adds the given text at the end of the current label with a bold font and formats it as a hyper link that opens the given url.

Parameters:

- **text:**

The text to add to the label.

- **url:**

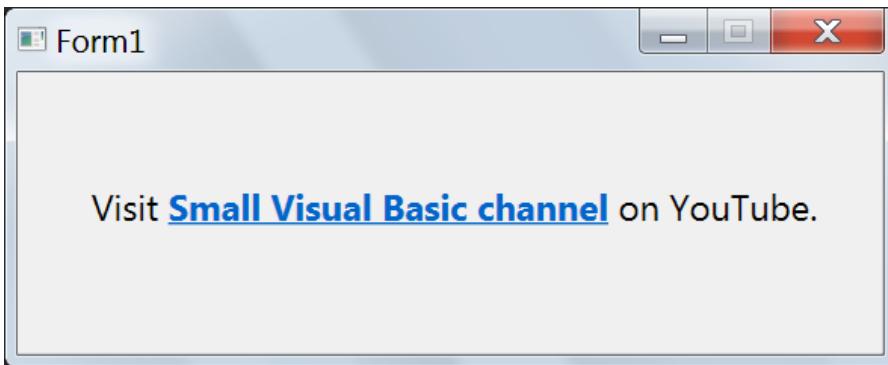
The address to navigate to. The value of this parameter can be:

- an empty string "" to view a normal text (not a link).
- a url to an internet or a network address.
- "about:blank", to open a new empty page in the browser.

- a local application path (an exe file) to open it.
- a file path (like a txt file), to make the windows open it with the suitable application (like notepad).
- a folder path to open it in the file explorer.

Example:

```
Label1.Text = "Visit "
Label1.AppendBoldLink(
    "Small Visual Basic channel",
    "https://youtube.com/@smallvisualbasic"
)
Label1.Append(" on YouTube.")
Label1.FitContentSize()
```



⌚ **Label.AppendFormatted(text, fontName, fontSize, isBold, isItalic, isUnderlined, foreColor, backColor, url)**

Adds the given text at the end of the current label with the given formats.

Note that all other Append methods like AppendBold and AppendItalicLink are just shortcuts for this method, and they actually call it to do the formatting.

Parameters:

- **text:**

The text to add to the label.

- **fontName:**

The name of the font to apply on the text. Send an empty string to use the current label font.

- **fontSize:**

The font size of the text. Send 0 to use the current label font size.

- **isBold:**

True to use a bold font, False otherwise. Send an empty string to use the current label FontBold value.

- **isItalic:**

True to use an italic font, False otherwise. Send an empty string to use the current label FontItalic value.

- **isUnderlined:**

True to draw a line under the text, False otherwise. Send an empty string to use the current label Underlined value.

- **foreColor:**

The color of the text. Send Colors.None to use the current label foreColor.

- **backColor:**

The background color of the text. Send Colors.None to use the current label backColor.

- **url:**

The address to navigate to. The value of this parameter can be:

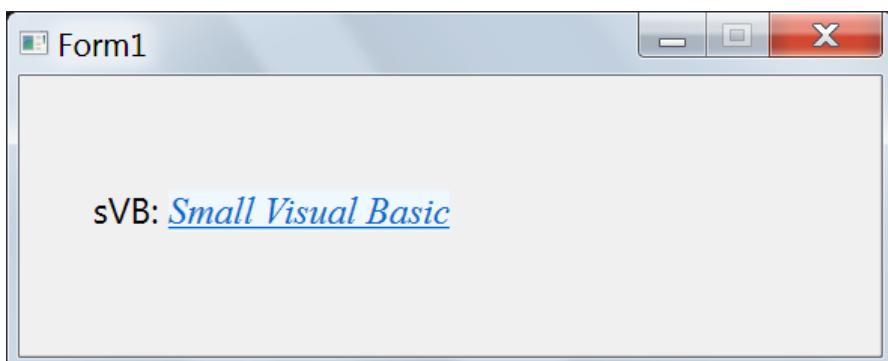
- an empty string "" to view a normal text (not a link).
- a url to an internet or a network address.
- "about:blank", to open a new empty page in the browser.
- a local application path (an exe file) to open it.
- a file path (like a txt file), to make the windows open it with the suitable application (like notepad).
- a folder path to open it in the file explorer.

Example:

```

Label1.Text = "sVB: "
Label1.AppendFormatted(
    "Small Visual Basic",
    "Times New Roman",
    20,
    False,
    True,
    "",
    Colors.None,
    Colors.AliceBlue,
    "https://github.com/VBAndCs/sVB-Small-Visual-Basic"
)
Label1.FitContentSize()

```



⌚Label.AppendItalic(text)

Adds the given text at the end of the current label with an italic font.

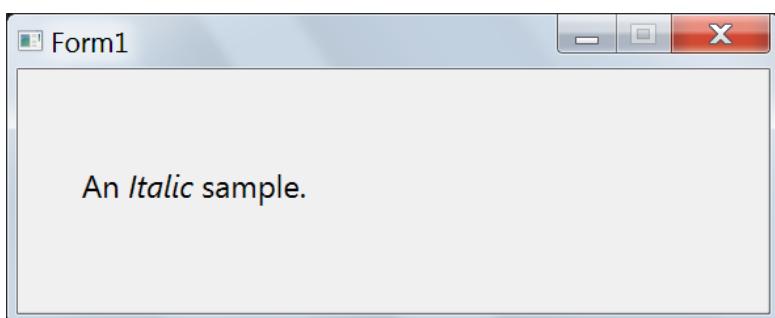
Parameter:

- **text:**

The text to add to the label.

Example:

```
Label1.Text = "An "
Label1.AppendItalic("Italic")
Label1.Append(" sample.")
Label1.FitContentSize()
```



⌚Label.AppendItalicLink(text, url)

Adds the given text at the end of the current label with an italic font, and formats it as a hyper link that opens the given url.

Parameters:

- **text:**

The text to add to the label.

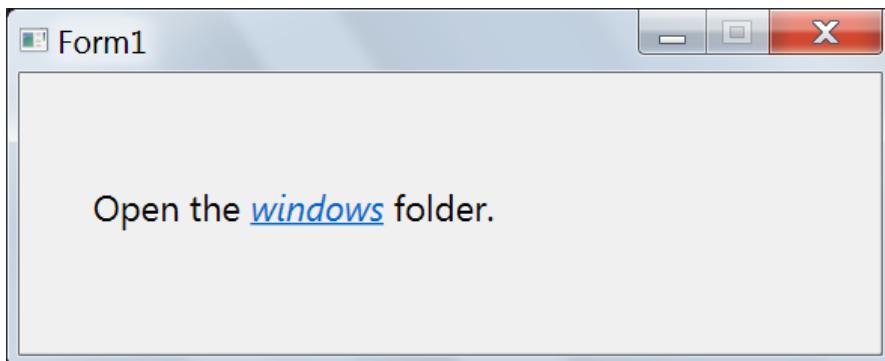
- **url:**

The address to navigate to, which can be:

- an empty string "" to view a normal text (not a link).
- a url to an internet or a network address.
- "about:blank", to open a new empty page in the browser.
- a local application path (an exe file) to open it.
- a file path (like a txt file), to make the windows open it with the suitable application (like notepad).
- a folder path to open it in the file explorer.

Example:

```
Label1.Text = "Open the "
Label1.AppendItalicLink("windows", "%windir%")
Label1.Append(" folder.")
Label1.FitContentSize()
```



⌚ **Label.AppendLine(text)**

Adds the given text at the end of the current label then inserts a new line.

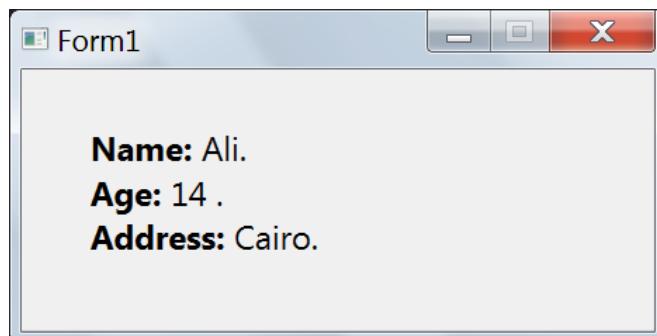
Parameter:

- **text:**

The text to add to the label.

Example:

```
Label1.Text = ""  
Label1.AppendBold("Name: ")  
Label1.AppendLine("Ali.")  
Label1.AppendBold("Age: ")  
Label1.AppendLine("14 .")  
Label1.AppendBold("Address: ")  
Label1.AppendLine("Cairo.")  
Label1.FitContentSize()
```



Label.AppendLink(text, url)

Adds the given text at the end of the current label, and formats it as a hyper link that opens the given url.

Parameters:

- **text:**

The text to add to the label.

- **url:**

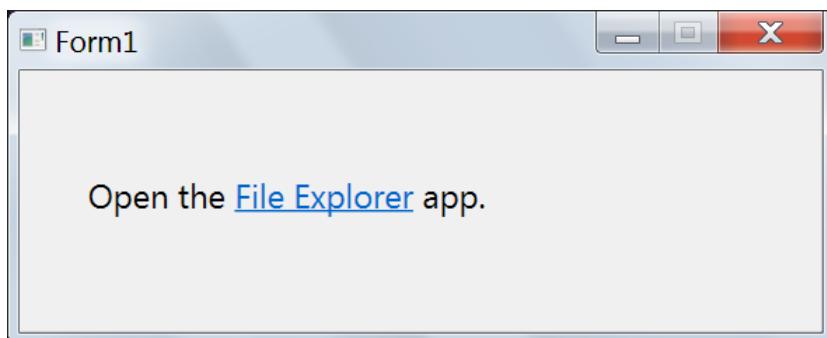
The address to navigate to. The value of this parameter can be:

- an empty string "" to view a normal text (not a link).
- a url to an internet or a network address.

- "about:blank", to open a new empty page in the browser.
- a local application path (an exe file) to open it.
- a file path (like a txt file), to make the windows open it with the suitable application (like notepad).
- a folder path to open it in the file explorer.

Example:

```
Label1.Text = "Open the "
Label1.AppendLink("File Explorer",
    "%windir%\explorer.exe")
Label1.Append(" app.")
Label1.FitContentSize()
```



Label.AppendUnderlined(text)

Adds the given text at the end of the current label with a line drawn under it.

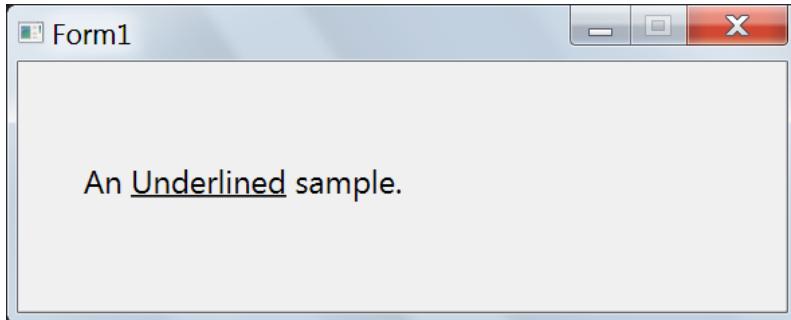
Parameter:

- **text:**

The text to add to the label.

Example:

```
Label1.Text = "An "
Label1.AppendUnderlined("Underlined")
Label1.Append(" sample.")
Label1.FitContentSize()
```



Label.AppendWithBackColor(text, backColor)

Adds the given text at the end of the current label with the given back color.

Parameters:

- **text:**

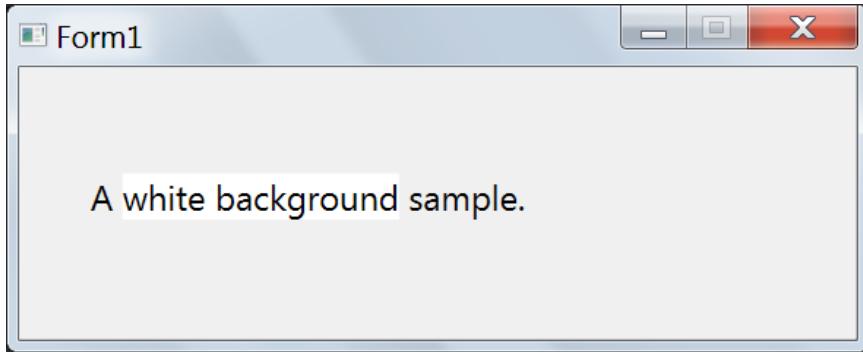
The text to add to the label.

- **backColor:**

The background color of the text. Send Colors.None to use the current label backColor.

Example:

```
Label1.Text = "A "
Label1.AppendWithBackColor(
    "white background", Colors.White)
Label1.Append(" sample.")
Label1.FitContentSize()
```



Label.AppendWithColors(text, foreColor, backColor)

Adds the given text at the end of the current label with the given fore and back colors.

Parameters:

- **text:**

The text to add to the label.

- **foreColor:**

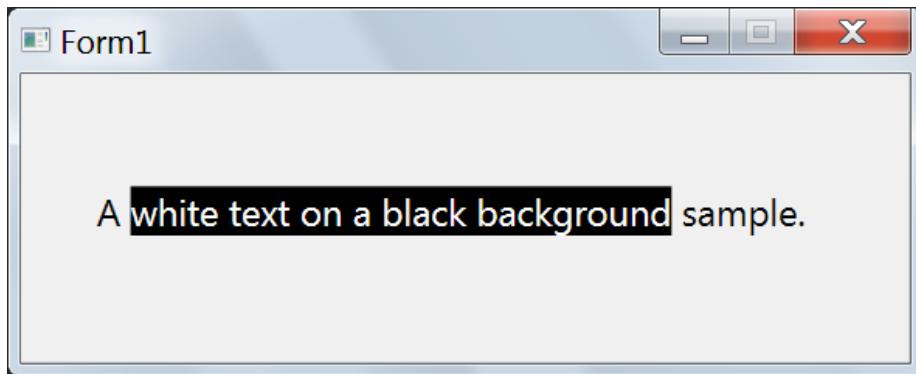
The color of the text. Send Colors.None to use the current label foreColor.

- **backColor:**

The background color of the text. Send Colors.None to use the current label backColor.

Example:

```
Label1.Text = "A "
Label1.AppendWithColors(
    "white text on a black background",
    Colors.White, Colors.Black)
Label1.Append(" sample.")
Label1.FitContentSize()
```



Label.AppendWithFont(text, fontName, fontSize, isBold, isItalic, isUnderlined)

Adds the given text at the end of the current label with the given font properties.

Parameters:

- **text:**

The text to add to the label.

- **fontName:**

The name of the font to apply on the text. Send an empty string to use the current label font.

- **fontSize:**

The font size of the text. Send 0 to use the current label font size.

- **isBold:**

True to use a bold font, False otherwise. Send an empty string "" to use the current label FontBold value.

- **isItalic:**

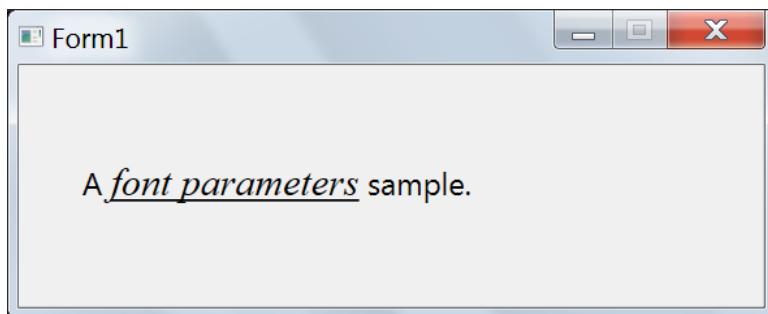
True to use an italic font, False otherwise. Send an empty string "" to use the current label FontItalic value.

- **isUnderlined:**

True to draw a line under the text, False otherwise. Send an empty string "" to use the current label Underlined value.

Example:

```
Label1.Text = "A "
Label1.AppendWithFont(
    "font parameters",
    "Times New Roman", 24,
    False, True, True)
Label1.Append(" sample.")
Label1.FitContentSize()
```



Label.AppendWithFontEffects(text, isBold, isItalic, isUnderlined)

Adds the given text at the end of the current label with the given font effects.

Parameters:

- **text:**

The text to add to the label.

- **isBold:**

True to use a bold font, False otherwise. Send an empty string "" to use the current label FontBold value.

- **isItalic:**

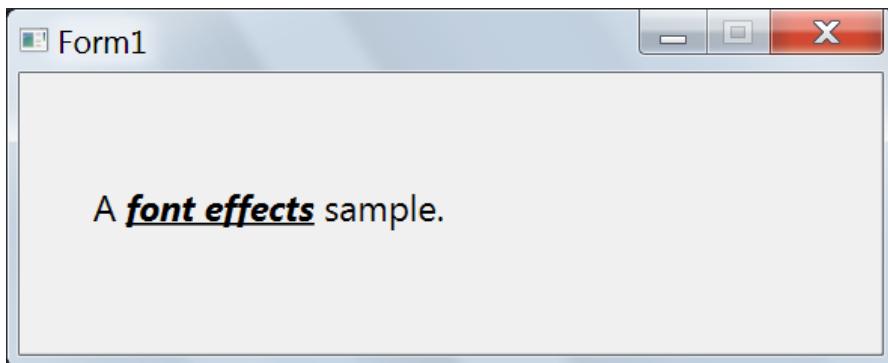
True to use an italic font, False otherwise. Send an empty string "" to use the current label FontItalic value.

- **isUnderlined:**

True to draw a line under the text, False otherwise. Send an empty string "" to use the current label Underlined value.

Example:

```
Label1.Text = "A "
Label1.AppendWithFontEffects(
    "font effects", True, True, True)
Label1.Append(" sample.")
Label1.FitContentSize()
```



⚙️ **Label.AppendWithFontName(text, fontName)**

Adds the given text at the end of the current label with the given font name.

Parameters:

- **text:**

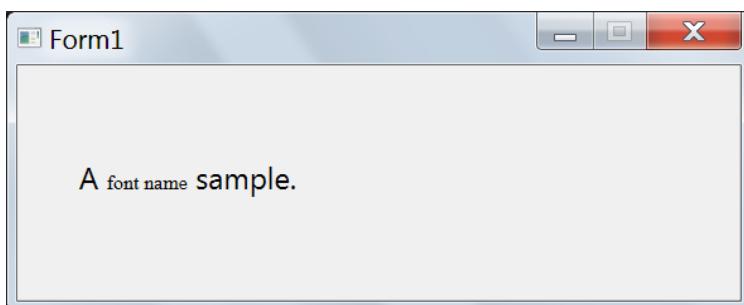
The text to add to the label.

- **fontName:**

The name of the font to apply on the text. Send an empty string to use the current label font.

Example:

```
Label1.Text = "A "
Label1.AppendWithFontName(
    "font name", "Microsoft Himalaya")
Label1.Append(" sample.")
Label1.FitContentSize()
```



Label.AppendWithFontNameAndSize(text, fontName, fontSize)

Adds the given text at the end of the current label with the given font name and size.

Parameters:

- **text:**

The text to add to the label.

- **fontName:**

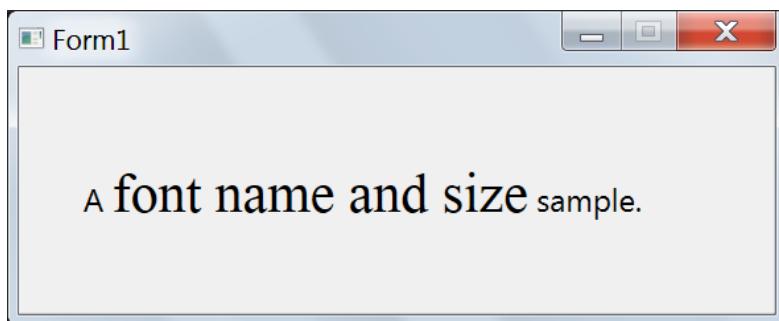
The name of the font to apply on the text. Send an empty string to use the current label font

- **fontSize:**

The font size of the text. Send 0 to use the current label font size.

Example:

```
Label1.Text = "A "
Label1.AppendWithFontNameAndSize(
    "font name", "Microsoft Himalaya", 50)
Label1.Append(" sample.")
Label1.FitContentSize()
```



⌚Label.AppendWithFontSize(text, fontSize)

Adds the given text at the end of the current label with the given font size.

Parameter:

- **text:**

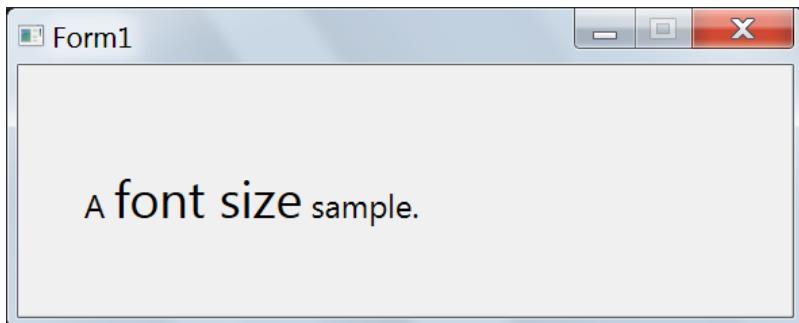
The text to add to the label.

- **fontSize:**

The font size of the text. Send 0 to use the current label font size.

Example:

```
Label1.Text = "A "
Label1.AppendWithFontSize(
    "font size", 30)
Label1.Append(" sample.")
Label1.FitContentSize()
```



Label.AppendWithForeColor(text, foreColor)

Adds the given text at the end of the current label with the given for color.

Parameters:

- **text:**

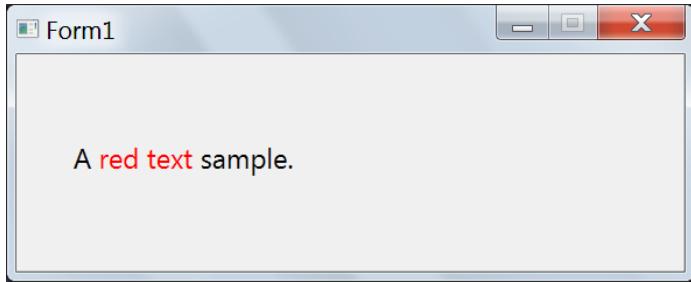
The text to add to the label.

- **foreColor:**

The color of the text. Send Colors.None to use the current label foreColor.

Example:

```
Label1.Text = "A "
Label1.AppendWithForeColor("red text", Colors.Red)
Label1.Append(" sample.")
Label1.FitContentSize()
```



Label.Image As String

Gets or sets the path of the image that is displayed on the label.

Notes:

1. You should save the image file in the project folder, and set its relative path to this property, so that the code continues to work fine regardless where it is saved on the PC. The relative path is the image name and extension without the folder path, like "myImage.jpg".
2. If you use a very high resolution image (with large dimensions and file size), it will take time to be loaded and resized down to the label size, so, you may suffer some delay before it is displayed, or it may not move smoothly on the screen when you animate it. You can make things better by editing the image (in Paint or any other photo editor) to make its dimensions smaller. This will make it faster when using it in your app.
3. You can remove the displayed image from the label by setting this property to an empty string "".
4. The image is displayed as the content of the label, so if you set the text property of the label, the label text will remove the displayed picture. This is different from using the form designer to set a picture as the background of the label, which allows the label to display both the text and image at the same time.

Examples:

You can see this method in action in the "Jerry" project in the samples folder. It is used in the form global area to display the jerry image in the label:

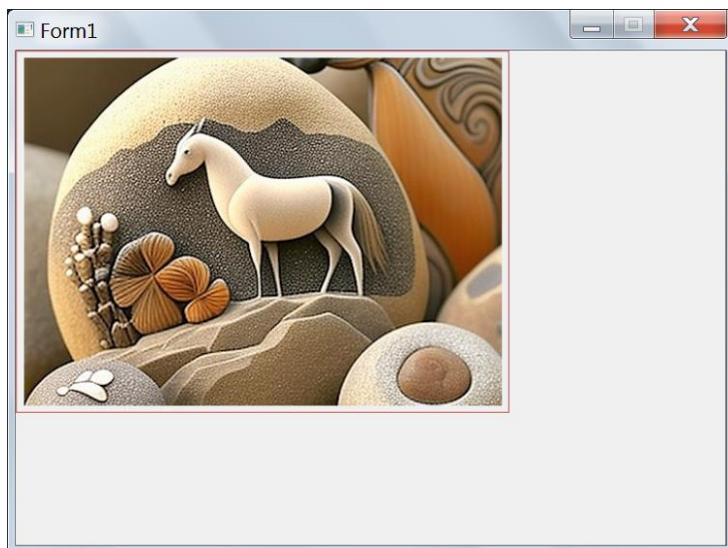
```
Jerry.Image = "1.png"
```

Note that the image will be resized to fit the label's width or height, to keep its aspect ratio. In other words: The image may not fully fill the label, but you can solve this by resizing the label itself to fully contain the displayed image. For example, if you set the label height to 300 and set its width to -1 (auto size), the image will be resized to have a height of 300, while its width will keep the original ratio to its height, and the label width will be resized to fit the width of the displayed image:

```
Label1.Image = "d:\1.png"
```

```
Label1.Width = -1
```

```
Label1.Height = 300
```

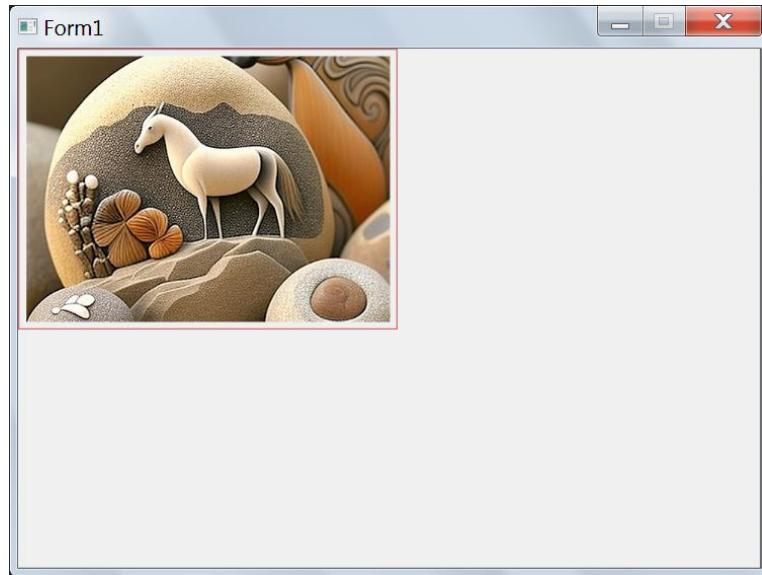


Or you can use a fixed width with an auto height:

```
Label1.Image = "d:\1.png"
```

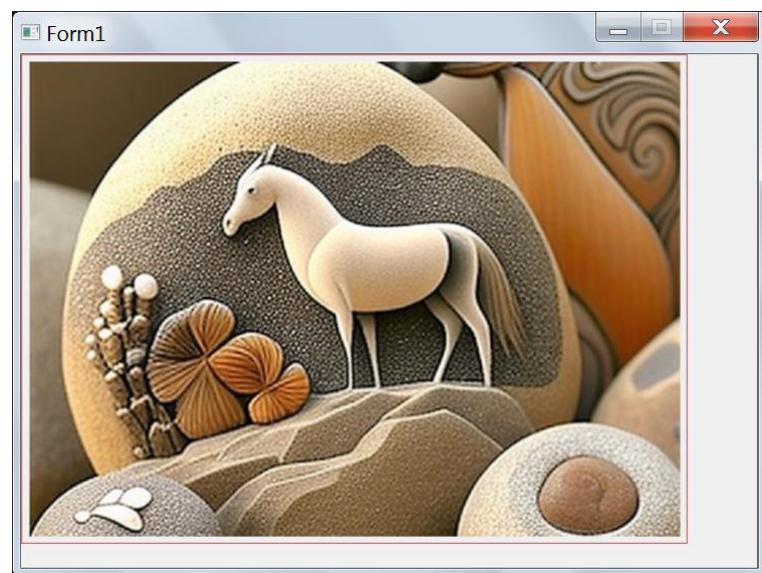
```
Label1.Width = 300
```

```
Label1.Height = -1
```



Or auto resize both the width and height of the label to take the size of the iamge, which may not fit the size of your form, so I don't recommend to do it unless you deal with small size images.

```
Label1.Image = "d:\1.png"  
Label1.Width = -1  
Label1.Height = -1
```



Label.Text As String

Gets or sets the text that is displayed on the label.

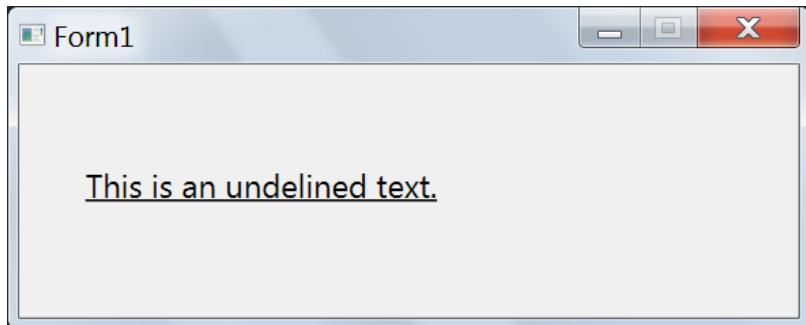
Label.Underlined As Boolean

Gets or sets whether or not to draw a line under the text displayed by the label.

Example:

```
Label1.Text = "This is an undelined text."
```

```
Label1.Underlined = True
```



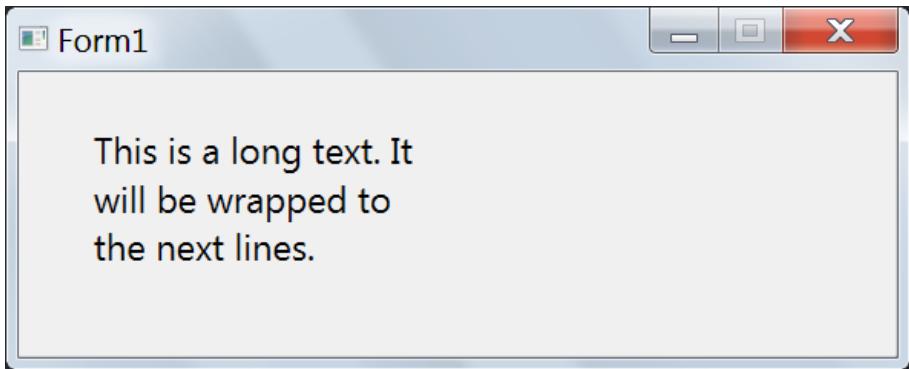
Label.WordWrap As Boolean

Gets or sets whether or not the text will be continue on the next line if it exceeds the width of the label.

The default value is True, so the label title will be wrapped over lines by default. You can use the [Label.FitContentHeight](#) method to ensure that the label height fits the wrapped lines.

Example:

```
Label1.Text = "This is a long text. "
+ "It will be wrapped To the Next lines."
Label1.WordWrap = True
Label1.FitContentHeight()
```



The ListBox Type

Represents a ListBox control, which shows a list of items to the user to select one of them.

You can use the form designer to add a list box to the form by dragging it from the toolbox.

It is also possible to use the Form.AddListBox method to create a new list box and add it to the form at runtime.

Note that the ListBox control inherits all the properties, methods and events of the [Control](#) type.

Besides, the ListBox control has some new members that are listed below:

ListBox.AddItem(value) As Double

Adds an item at the end of the list.

Parameter:

- **value:**

The item you want to add to the list. You can send an array to add all its items.

Returns:

The index of the newly added item, or 0 if the operation failed.

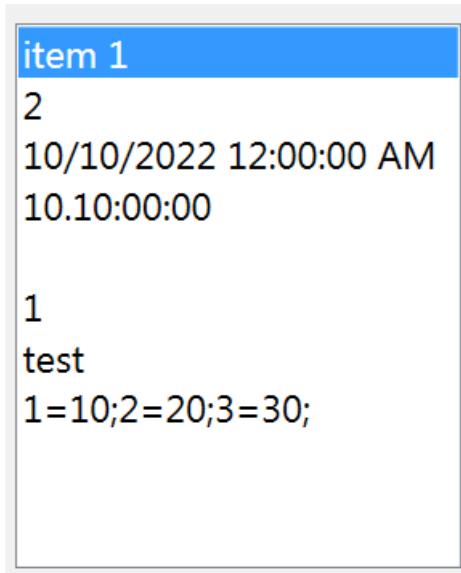
Examples:

Add a ListBox on the form, and add the following code to the form global area:

```
ListBox1.AddItem("item 1")
ListBox1.AddItem(2)
```

```
ListBox1.AddItem("#10/10/2022#")
ListBox1.AddItem("#+10.10:00#")
ListBox1.AddItem("")
ListBox1.AddItem({1, "test", {10, 20, 30}})
```

When you run the program, the list box will be populated with the items shown in the picture:



As you can see, the list box can show strings, dates and durations (time spans). You can even add an empty item, like the fifth item, which is the result of adding empty string "" to the list box.

It is also obvious that the array added with the last line of code is added to the list box as the three items:

```
1
test
1=10;2=20;3=30;
```

Where the last one is the sVB string representation of the array {10, 20, 30}. This means that only first level items are added to the list box as single items, but the AddItem method will not go deeper in the inner arrays. It just add the inner

array as a single item, which appears as you see, while you can still read this item as an array. This is just the how the list box displays it.

In fact, this is not a user friendly way, so, don't add an array as a single item.

● **ListBox.AddItemAt(value, index)**

Adds the given item to the list at the given index.

Parameters:

- **value:**

The item you want to add to the list. You can send an array to add all its items to the list box starting from the given index.

- **index:**

The index you want to add the item at. The value of this index must be greater than 0 and less than list items count + 1, otherwise the item will not be added.

Returns:

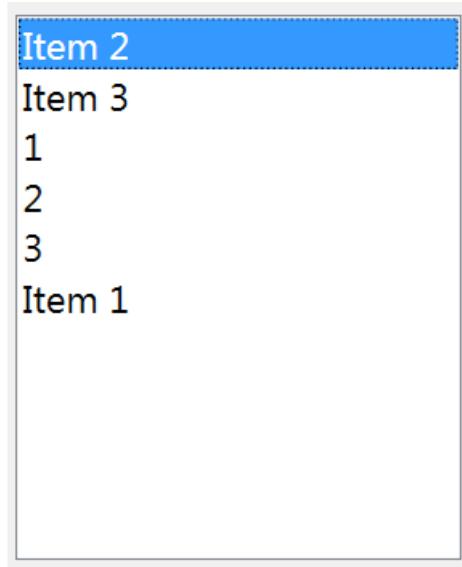
True if the item is successfully added at the given index, otherwise False.

Examples:

Add a ListBox on the form, and add the following code to the form global area:

```
ListBox1.AddItemAt("Item 1", 1)
ListBox1.AddItemAt("Item 2", 1)
ListBox1.AddItemAt("Item 3", 2)
ListBox1.AddItemAt("Item 4", 5)
ListBox1.AddItemAt({1, 2, 3}, 3)
```

When you run the program, the list box will be populated with the items shown in the picture:



note that "Item 4" was not added, because we asked to add it outside the possible list indexes.

💡 **ListBox.ContainsItem(value) As Boolean**

Checks whether or not the given item exists in the list.

Parameter:

- **value:**

The value of the item to search for in the listbox.

Returns:

True if the item exists, or False otherwise.

Examples:

Add a list box on the form, and write the following code in the form global area:

```
ListBox1.AddItem({1, 2, 3})  
TextWindow.WriteLine(ListBox1.ContainsItem(2))  
TextWindow.WriteLine(ListBox1.ContainsItem(5))
```

When you run the program, the text window will show the following results:

True
False

ListBox.FindItem(value) As Double

Returns the index of the given item if it exists in the list, otherwise returns 0.

Parameter:

- **value:**

The item you want to find.

Returns:

The index of the item if found, or 0 otherwise.

Examples:

Add a list box on the form, and write the following code in the form global area:

```
ListBox1.AddItem({1, 2, 3})  
TextWindow.WriteLine(ListBox1.FindItem(2))  
TextWindow.WriteLine(ListBox1.FindItem(5))
```

When you run the program, the text window will show the following results:

2
0

ListBox.FindItemAt(value, startIndex, endIndex) As Double

Returns the index of the given item if it exists in the list in the given index range, otherwise returns 0.

Parameters:

- **value:**

The item you want to find.

- **startIndex:**

The array index the search starts at.

- **endIndex:**

The array index the search ends at. If endIndex < startIndex, the search direction will be reversed to find the last index of the item.

Returns:

The index of the item if found, or 0 otherwise.

Examples:

Add a list box on the form, and write the following code in the form global area:

```
ListBox1.AddItem({1, 2, 3, 2, 4, 5})
TextWindow.WriteLine(ListBox1.FindItemAt(2, 1, 4))
TextWindow.WriteLine(ListBox1.FindItemAt(2, 3, 5))
TextWindow.WriteLine(ListBox1.FindItemAt(2, 5, 6))
TextWindow.WriteLine(ListBox1.FindItemAt(2, 6, 1))
```

When you run the program, the text window will show the following results:

2
4
0
4

ListBox.GetItemAt(index) As String

Returns the item that has the given index.

Parameter:

- **index:**

The index of the item. It should be greater than zero and not exceed the count of items, otherwise, this method will return an empty string.

Returns:

The value of the item if the index is valid, or "" otherwise.

Examples:

Add a list box on the form, and write the following code in the form global area:

```
ListBox1.AddItem({1, 2, 3})
TextWindow.WriteLine("[")
TextWindow.WriteLine(ListBox1.GetItemAt(0) + ", ")
TextWindow.WriteLine(ListBox1.GetItemAt(3) + ", ")
TextWindow.WriteLine(ListBox1.GetItemAt(5) + "]")
```

When you run the program, the text window will show:

```
[, 3, ]
```

ListBox.Items As Array

Gets an array containing the items of the ListBox.

Examples:

Add a list box on the form, and write the following code in the form global area:

```
ListBox1.AddItem({1, 2, 3})
items = ListBox1.Items
TextWindow.WriteLine(items.ToString())
```

When you run the program, the text window will show:

{1, 2, 3}

ListBox.ItemsCount As Double

Gets the count of the items in the ListBox.

Examples:

Add a list box on the form, and write the following code in the form global area:

```
TextWindow.WriteLine(ListBox1.ItemsCount)
ListBox1.AddItem({1, 2, 3})
TextWindow.WriteLine(ListBox1.ItemsCount)
```

When you run the program, the text window will show:

0
3

ListBox.OnSelection

This event is fired when the selected item in the list is changed.

This event will also be fired when the user deselects all the items by setting the SelectedIndex property to 0.

Note that this is the default event for the ListBox control, so, you can double-click any ListBox on the form designer (say ListBox1 for example), to get this event handler written for you in the code editor:

```
Sub ListBox1_OnSelection()
```

```
EndSub
```

For more info, read about [handling control events](#).

Example:

This event is used in the "ListBox" project in the samples folder, to display the selected item in the label, and update the state of the remove button. You will find this code in that project:

```
Sub ListBox1_OnSelection
    lblSelection.Text = "Selected item: " +
    ListBox1.SelectedItem
    btnRemove.Enabled = (ListBox1.SelectedIndex > 0)
EndSub
```

ListBox.RemoveAllItems()

Removes all the items from the listbox.

Example:

```
ListBox1.RemoveAllItems()
```

ListBox.RemoveItem(value)

Searches for the given value in the list, and removes the first found item.

Parameter:

- **value:**

The item you want to remove. You can send an array to remove all its items.

Returns:

True if the item is successfully removed, or False if the item is not found.

Example:

Add a list box on the form, and write the following code in the form global area:

```
ListBox1.AddItem({2, 1, 2, 2, 2, 3, 4, 2, 5, 2})  
While ListBox1.RemoveItem(2)  
Wend  
TextWindow.WriteLine(Array.ToString(ListBox1.Items))
```

The above code shows you how to use the RemoveItem in a while loop to remove all items that have the value 2 from the list box. After running this project, the text window will display:

{1, 3, 4, 5}

○ **ListBox.RemoveItemAt(index)**

Removes the list item that exists at the given index.

Parameter:

- **index:**

The index of the item you want to remove. You can send and array of indexes to remove them all, but be careful that removing an item changes the indexes of the following items, so it will be easier for you to arrange the indexes in a descending order, so that removing each item doesn't affect the indexes to be deleted next.

Returns:

True if the item is successfully removed, or False if the given index is < 1 or > list items count.

Examples:

Add a list box on the form, and write the following code in the form global area:

```
ListBox1.AddItem({1, 2, 2, 3, 4, 2, 5})  
ListBox1.RemoveItemAt(6)  
ListBox1.RemoveItemAt(2)  
TextWindow.WriteLine(Array.ToString(ListBox1.Items))
```

The above code shows you how to use the RemoveItem in a while loop to remove all items that have the value 2 from the list box.

Note the two RemoveAt calls can be done with only this single call:

```
ListBox1.RemoveItemAt({6, 2})
```

After running this project, the text window will display:

```
{1, 2, 3, 4, 5}
```

ListBox.SelectedIndex As Double

Gets or sets the index of the selected item in the ListBox.
Zero indicates that no item is selected.

Examples:

You can see the SelectedIndex property in action in the "ListBox" project in the samples folder. It is used in many places, such as the ListBox1_OnSelection event handler to disable the Remove button if there is no item selected in the list:

```
btnRemove.Enabled = (ListBox1.SelectedIndex > 0)
```

And it is also used in the BtnRemove_OnClick event handler to remove the selected item and select the next or previous item:

```
Sub BtnRemove_OnClick
    index = ListBox1.SelectedIndex
    ListBox1.RemoveItemAt(index)

    If index < ListBox1.ItemsCount Then
        ListBox1.SelectedIndex = index
    ElseIf ListBox1.ItemsCount > 0 Then
        ListBox1.SelectedIndex = ListBox1.ItemsCount
    EndIf
EndSub
```

ListBox.SelectedItem As String

Gets or sets the item that is currently selected in the ListBox.

If you set this property to a value that exists in the list, the first item that has this value will be selected (comparison is case-sensitive), otherwise the value is ignored, and the selected item doesn't change.

Examples:

You can see the SelectedIndex property in action in the "ListBox" project in the samples folder. It is used in many places, such as the ListBox1_OnSelection event handler to display the selected item in the lblSelection:

```
lblSelection.Text = "Selected item: "
    + ListBox1.SelectedItem
```

ListBox.SetItemAt(index, value)

Sets the value of the item that exists at the given index in the list.

Parameters:

- **index:**

The index of the item. It should be greater than zero and not exceed the count of the items, otherwise, this method will return False.

- **value:**

The value to set to the item.

Returns:

True if the item is set, or False otherwise.

Examples:

Add a list box on the form, and write the following code in the form global area:

```
ListBox1.AddItem({1, 2, 3})
TextWindow.WriteLine(ListBox1.SetItemAt(0, 0))
TextWindow.WriteLine(ListBox1.SetItemAt(3, 30))
TextWindow.WriteLine(ListBox1.SetItemAt(4, 40))
TextWindow.WriteLine(Array.ToString(ListBox1.Items))
```

When you run the program, the text window will show:

False

True

False

{1, 2, 30}

The MainMenu Type

Represents the Menu control, which shows menu items on a bar, so the user can click any of them to drop down a list of sub menu items.

The form designer doesn't support adding a main menu at design time, but you can use the [Form.AddMainMenu](#) method to add it in runtime.

Note that the MainMenu control inherits all the properties, methods and events of the [Control](#) type.

Besides, the MainMenu control has some new members that are listed below:

MainMenu.AddItem(itemName, text, shortcut) As MenuItem

Adds a menu item to the current menu.

Parameters:

- **itemName:**

The name of the new menu item.

- **text:**

The title of the menu item.

- **shortcut:**

The keyboard shortcut keys, like Ctrl+N, or use "" if there is no shortcut.

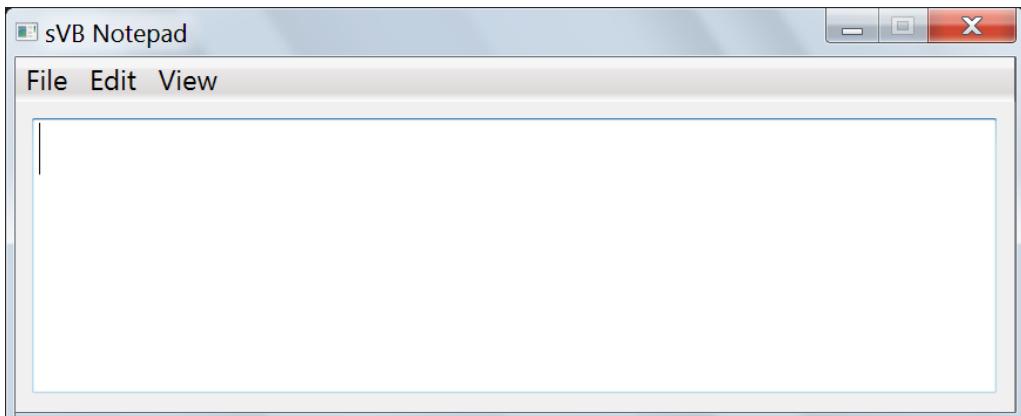
Returns:

The menu item that have been added.

Examples:

You can see this method in action in the "sVB Notepad" project in the samples folder. It is used in the global area of the frmMain.sb to create the main menu items. This is a part of this code:

```
M = Me.AddMainMenu("mainmenu")
MenuItem = M.AddItem("menuItem", "File", "")
MenuItem = M.AddItem("menuItem", "Edit", "")
MenuItem = M.AddItem("MenuItem", "View", "")
```



>MainMenu.MenuItems As Array

Returns an array containing the child menu items of the current menu.

The Math Type

The Math class provides lots of useful mathematics related methods.

The Math type has these members:

Math.Abs(number) As Double

Gets the absolute value (the positive value) of the given number. For example, -32.233 will return 32.233.

Parameter:

- **number:**

The number to get the absolute value for. This argument can be omitted if you call this method as an extension property of a numeric variable.

Returns:

The absolute value of the given number.

Examples:

X = 1.5

```
TextWindow.WriteLine({  
    X.Abs,  
    Math.Abs(-1.5),  
    Math.Abs(-X)  
})
```

```
1.5  
1.5  
1.5
```

⌚Math.ArcCos(cosValue) As Double

Gets the angle in radians, of the given cosine value.

Parameter:

- **cosValue:**

The cosine value whose angle is needed. This argument can be omitted if you call this method as an extension property of a numeric variable.

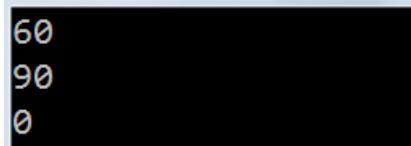
Returns:

The angle (in radians) for the given cosine Value.

Examples:

X = 0.5

```
TextWindow.WriteLine({  
    Math.GetDegrees(X.ArcSin),  
    Math.GetDegrees(Math.ArcSin(0)),  
    Math.GetDegrees(Math.ArcSin(1))  
})
```



60
90
0

⌚Math.ArcSin(sinValue) As Double

Gets the angle in radians, of the given sine value.

Parameter:

- **sinValue:**

The sine value whose angle is needed. This argument can be omitted if you call this method as an extension property of a numeric variable.

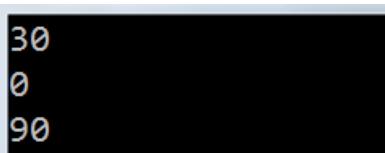
Returns:

The angle (in radians) for the given sine Value.

Examples:

X = 0.5

```
TextWindow.WriteLine({  
    Math.GetDegrees(X.ArcSin),  
    Math.GetDegrees(Math.ArcSin(0)),  
    Math.GetDegrees(Math.ArcSin(1))  
})
```



```
30  
0  
90
```

Math.ArcTan(tanValue) As Double

Gets the angle in radians, of the given tangent value.

Parameter:

- tanValue:

The tangent value whose angle is needed. This argument can be omitted if you call this method as an extension property of a numeric variable.

Returns:

The angle (in radians) for the given tangent Value.

Examples:

X = 0.5

```
TextWindow.WriteLine({  
    Math.GetDegrees(X.ArcTan),  
    Math.GetDegrees(Math.ArcTan(0)),  
    Math.GetDegrees(Math.ArcTan(1))  
})
```

```
26.56505
```

```
0
```

```
45
```

⚙️**Math.Ceiling(number) As Double**

Returns the smallest integer that is greater than or equal to the argument. It rounds up the integer value.

For example, 32.233 will return 33. Also, 44 will return 44.

Parameter:

- **number:**

The number whose ceiling is required. This argument can be omitted if you call this method as an extension property of a numeric variable.

Returns:

The ceiling value of the given number.

Examples:

```
X = 1.5
```

```
TextWindow.WriteLine({  
    X.Ceiling,           ' 2  
    Math.Ceiling(1.7),   ' 2  
    Math.Ceiling(2),     ' 2  
    Math.Ceiling(-1.5),  ' -1  
    Math.Ceiling(-1.7),  ' -1  
    Math.Ceiling(-2)     ' -2  
})
```

```
2
```

```
2
```

```
2
```

```
-1
```

```
-1
```

```
-2
```

Math.Cos(angle) As Double

Gets the cosine of the given angle in radians. Remember that radian angles can have values between 0 and $2 * \text{Math.Pi}$, and you can use the `Math.GetRadians` method to convert an angle from degrees to radians.

Parameter:

- **angle:**

The angle (in radians) whose cosine is needed. This argument can be omitted if you call this method as an extension property of a numeric variable.

Returns:

The cosine of the given angle.

Examples:

```
X = 0
Y = 90
Z = Math.Cos(Y.Radians)
TextWindow.WriteLine({
    X.Cos,
    Math.Cos(Math.Pi),
    Z,
    Math.Round2(Z, 5)
})
```

```
1
-1
-0.000000000000034914833611094
0.00000
```

Note that calculations on decimal numbers can have a tiny accuracy issue that you need to take care of. In the third result above, zero is represented as a very verey very small negative number, which is practically 0, so, we rounded this

number to get the actual zero. So, be careful when you compare two decimal values (like the results from division and other math operations), because using $>$, $<$ or $=$ can give you unexpected results if you don't round the decimal numbers to an suitable decimal positions like 5 or 7.

You can also subtract the two values you want to compare, and check that the absolute value of the result is smaller than a very very neglectable value such as 0.000000000001.

The next code samples shows you 3 ways to check if Z equals X, but only the last two ways will return the right result:

```
X = 0
Y = 90
Z = Math.Cos(Y.Radians)
TextWindow.WriteLine({
    Z = X,                                ' False
    Math.Round2(Z, 5) = X,                  ' True
    Math.Abs(X - Z) <= 0.0000000001 ' True
})
```

Math.Decimal(hex) As double

Converts the given hexadecimal number to a decimal number.

You can use the Math.Hex method to convert a decimal number to a hexadecimal number.

Parameter:

- **hex:**

The hexadecimal number whose decimal value is required. This argument can be omitted if you call this method as an extension method (with the name ToDeciaml) of a numeric or string variables, because hex numbers can be a pure digital numbers like 10 which is the hex representation of 16 in the

decimal system, or it can contain letters from A to F which makes it a string.

Returns:

The decimal value the hex number if it is valid, or an empty string otherwise.

Examples:

N = 51

X = "AF"

```
TextWindow.WriteLine({  
    X.ToDecimal(),           ' 175  
    N.ToDecimal(),           ' 81  
    Math.Decimal("1F"),      ' 31  
    Math.Decimal(10),        ' 16  
    Math.Decimal("EE"),      ' 238  
    Math.Decimal("2B"),      ' 43  
    Math.Decimal("40")       ' 64  
})
```

Math.Degrees As Double

This is an extension property of array variables. It converts the current angle from radians to degrees.

You can also use the [Math.GetDegrees](#) method to do the same job.

Math.E As Double

Gets the natural logarithmic base, which equals 2.7182818284590451.

Math.Floor(number) As Double

Returns the largest integer that is less than or equal to the argument. It rounds down the integer value.

For example, 32.233 will return 32. Also, 44 will return 44.

Parameter:

- **number:**

The number whose floor value is required. This argument can be omitted if you call this method as an extension property of a numeric variable.

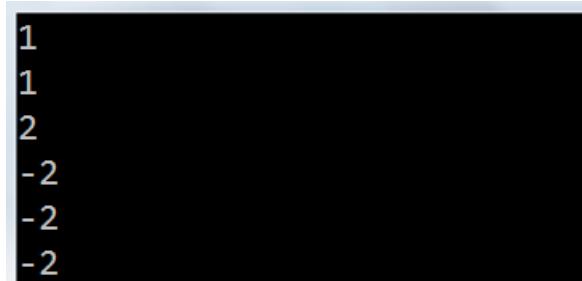
Returns:

The floor value of the given number.

Examples:

X = 1.5

```
TextWindow.WriteLine({  
    X.Floor,          ' 1  
    Math.Floor(1.7), ' 1  
    Math.Floor(2),   ' 2  
    Math.Floor(-1.5), ' -2  
    Math.Floor(-1.7), ' -2  
    Math.Floor(-2)   ' -2  
})
```



```
1  
1  
2  
-2  
-2  
-2
```

Math.GetDegrees(angle) As Double

Converts a given angle from radians to degrees. You can manually do this using this relation:

$$\text{degrees} = \text{radians} * 180 / \text{Math.Pi}.$$

Parameter:

- **angle:**

The angle in radians. This argument can be omitted if you call this method as an extension property (with the name Degrees) of a numeric variable.

Returns:

The converted angle in degrees.

Examples:

```
Pi = Math.Pi
TextWindow.WriteLine({
    Math.GetDegrees(0),          ' 0
    Math.GetDegrees(Pi / 6),     ' 30
    Math.GetDegrees(Pi / 4),     ' 45
    Math.GetDegrees(Pi / 3),     ' 60
    Math.GetDegrees(Pi / 2),     ' 90
    Math.GetDegrees(3 * Pi / 4), ' 135
    Pi.Degrees,                 ' 180
    Math.GetDegrees(5 * Pi / 4), ' 225
    Math.GetDegrees(3 * Pi / 2), ' 270
    Math.GetDegrees(7 * Pi / 4), ' 315
    Math.GetDegrees(2 * Pi)      ' 0 (= 360)
})
```

Math.GetRadians(angle) As Double

Converts a given angle in degrees to radians. You can manually do this using this relation:

$$\text{radians} = \text{degrees} * \text{Math.Pi} / 180.$$

Parameter:

- **angle:**

The angle in degrees. This argument can be omitted if you call this method as an extension property (with the name Radians) of a numeric variable.

Returns:

The converted angle in radians.

Examples:

```
A = 180
TextWindow.WriteLine({
    Math.GetRadians(0),
    Math.GetRadians(30),
    Math.GetRadians(45),
    Math.GetRadians(60),
    Math.GetRadians(90),
    A.Radians,
    Math.GetRadians(270),
    Math.GetRadians(360)
})
```

```
0
0.523598775598299
0.785398163397448
1.0471975511966
1.5707963267949
3.14159265358979
4.71238898038469
0
```

Math.GetRandomNumber(maxNumber) As Double

Gets a random number between 1 and the specified maxNumber (inclusive).

Parameter:

- **maxNumber:**

The maximum number for the requested random value. This argument can be omitted if you call this method as an extension property (with the name Random) of a numeric variable.

Returns:

A Random number that is less than or equal to the specified max.

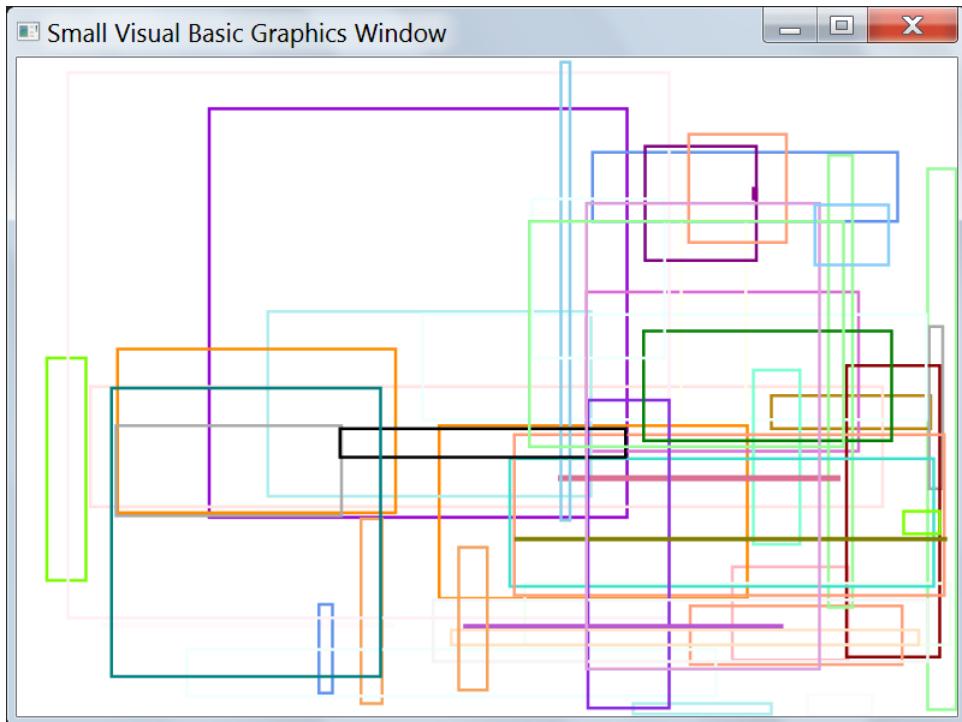
Examples:

This program draws rectangles on the screen with random positions, sizes, and colors:

```
While True
    DrawRect()
    Program.Delay(100)
Wend

Sub DrawRect()
    GraphicsWindow.PenColor = Color.GetRandomColor()
    w = GraphicsWindow.Width
    h = GraphicsWindow.Height
    x = w.Random
    y = h.Random
    GraphicsWindow.DrawRectangle(
        x,
        y,
        Math.GetRandomNumber(w - x),
        Math.GetRandomNumber(h - y)
    )
EndSub
```

When you run this code, you will get something like this:



You may also try to fill the rectangles, by using the PenColor property instead of the BrushColor property, and using the FillRectangle method instead of the DrawRectangle method:



Math.Hex(decimal) As String

Converts the given decimal number to its hexadecimal representation. Every digit in the hexadecimal number can have a value from 1 to 15. We already know the symbols for the values from 0 to 9, and the letters from A to F are used to represent the values from 10 to 15. You can use the Math.Decimal to convert a hexadecimal number to a decimal number.

Parameter:

- **decimal:**

The decimal number whose hexadecimal value is required. This argument can be omitted if you call this method as an extension property of a numeric variable.

Returns:

A string that represents the hexadecimal value the number.

Examples:

X = 10

```
TextWindow.WriteLine({  
    X.Hex,                      ' A  
    Math.Hex(12),                ' C  
    Math.Hex(15),                ' F  
    Math.Hex(16),                ' 1E  
    Math.Hex(30),                ' 10  
    Math.Hex(43),                ' 2B  
    Math.Hex(64)                 ' 40  
})
```

Math.Log(number) As Double

Gets the logarithm (base 10) value of the given number.

For example, $\text{Log}(100) = 2$, because $100 = 10^2$.

Parameter:

- **number:**

The number whose logarithm value is required. This argument can be omitted if you call this method as an extension property of a numeric variable.

Returns:

The log value of the given number.

Examples:

```
X = 10
TextWindow.WriteLine({
    X.Log,                      ' 1
    Math.Log(100),              ' 2
    Math.Log(1000),             ' 3
    Math.Log(0.001),            ' -3
    Math.Log(Math.Power(10, 4)) ' 4
})
```

Math.Max(number1, number2) As Double

Compares two numbers and returns the greater of the two.

Parameters:

- **number1:**

The first of the two numbers to compare.

- **number2:**

The second of the two numbers to compare.

Returns:

The greater value of the two numbers.

Examples:

```
TextWindow.WriteLine({  
    Math.Max(1, 2),      ' 2  
    Math.Max(0, 3),      ' 3  
    Math.Max(-4, -3),    ' -3  
    Math.Max(5, 5)       ' 5  
})
```

Math.Min(number1, number2) As Double

Compares two numbers and returns the smaller of the two.

Parameters:

- **number1:**

The first of the two numbers to compare.

- **number2:**

The second of the two numbers to compare.

Returns:

The smaller value of the two numbers.

Examples:

```
TextWindow.WriteLine({  
    Math.Min(1, 2),      ' 1  
    Math.Min(0, 3),      ' 0  
    Math.Min(-4, -3),    ' -4  
    Math.Min(5, 5)       ' 5  
})
```

Math.NaturalLog(number) As Double

Gets the natural logarithm value of the given number. The base of this logarithm is Math.E .

Parameter:

- **number:**

The number whose natural logarithm value is required.

Returns:

The natural log value of the given number. This argument can be omitted if you call this method as an extension property of a numeric variable.

Examples:

```
E = Math.E
TextWindow.WriteLine({
    Math.Round(E.NaturalLog),           ' 1
    Math.NaturalLog(E.Power(2)),        ' 2
    Math.NaturalLog(E.Power(-2)),       ' -2
    Math.NaturalLog(1)                 ' 0
})
```

Math.Pi As Double

Gets the value of Pi, which equals 3.1415926535897931.

Math.Power(baseNumber, exponent) As Double

Raises the base number to the specified power.

Parameters:

- **baseNumber:**

The number to be raised to the exponent power. This argument can be omitted if you call this method as an extension method of a numeric variable.

- **exponent:**

The power to raise the base number.

Returns:

The base number raised to the specified exponent.

Examples:

```
X = 5
TextWindow.WriteLine({
    X.Power(2),      ' 25
    Math.Power(2, 3), ' 8
    Math.Power(10, 4), ' 10000
    Math.Power(10, -2) ' 0.01
})
```

Math.Radians As Double

This is an extension property of array variables. It converts the current angle from degrees to radians.

You can use also use the [Math.GetRadians](#) method to do the same job.

Math.Random As Double

This is an extension property of array variables. It gets a random number between 1 and the current number (inclusive).

You can use also use the [Math.GetRandomNumber](#) method to do the same job.

Math.Remainder(dividend, divisor) As Double

Divides the first number by the second and returns the remainder.

Note that you can use the Mod operator to do the same job of this method.

Parameters:

- **dividend:**

The number to divide. It can be positive, negative or zero.

This argument can be omitted if you call this method as an extension method of a numeric variable.

- **divisor:**

The number that divides. If this value is 0, it will cause an error in your program, so you should avoid it.

Returns:

The remainder after the division, which can be positive or negative. For example: the remainder of dividing -10 by -3 is -1, because: $-10/-3 = 3$, where $3 * -3 = -9$, so the remainder is $-10 - (-9) = -10 + 9 = -1$.

Examples:

```
X = 5
TextWindow.WriteLine({
    X.Remainder(2),           ' 1
    X Mod 2,                 ' 1
    Math.Remainder(2, 3),     ' 2
    2 Mod 3,                 ' 2
    Math.Remainder(7, 4),     ' 3
    7 Mod 4,                 ' 3
    Math.Remainder(-9, -2),   ' -1
    - 9 Mod -2              ' -1
})
```

Math.Round(number) As Double

Rounds a given number to the nearest integer. For example 32.233 is rounded to 32.0 while 32.566 is rounded to 33.

Parameter:

- **number:**

The number whose approximation is required. This argument can be omitted if you call this method as an extension property of a numeric variable.

Returns:

The rounded value of the given number.

Examples:

```
X = 5.5
TextWindow.WriteLine({
    X.Round,                  ' 6
    Math.Round(0.01),         ' 0
    Math.Round(12.00009),     ' 13
    Math.Round(-2.8)          ' -3
})
```

Math.Round2(number, decimalPlaces) As Double

Rounds a given number to the given decimal places.

Parameter:

- **number:**

The number whose approximation is required. This argument can be omitted if you call this method as an extension method (with the name Round) of a numeric variable.

- **decimalPlaces:**

The number of decimal places to keep in the number.

Returns:

The rounded value of the given number.

Examples:

```
X = 5.21562
TextWindow.WriteLine({
    X.Round(2),                      ' 5.22
    Math.Round2(0.01, 1),             ' 0.0
    Math.Round2(12.0009, 3),          ' 12.001
    Math.Round2(-2.8, 0)              ' -3
})
```

Math.Sin(angle) As Double

Gets the sine of the given angle in radians. Gets the cosine of the given angle in radians. Remember that radian angles can have values between 0 and $2 * \text{Math.Pi}$, and you can use the `Math.GetRadians` method to convert an angle from degrees to radians.

Parameter:

- **angle:**

The angle whose sine is needed (in radians). This argument can be omitted if you call this method as an extension property of a numeric variable.

Returns:

The sine of the given angle.

Examples:

```
Pi = Math.Pi
X = Pi / 2
Z = Math.Sin(Pi)
TextWindow.WriteLine({
    X.Sin,
    Math.Sin(3 * X),
    Z,
    Math.Round2(Z, 5),
    Math.Sin(Math.GetRadians(30)),
    Math.Sin(Math.GetRadians(60))
})
```

```
1
-1
0.000000000000032310851043327
0.00000
0.5
0.86602540378444
```

Math.SquareRoot(number) As Double

Gets the square root of a given number.

Parameter:

- **number:**

The number whose square root value is needed. This argument can be omitted if you call this method as an extension property of a numeric variable.

Returns:

The square root value of the given number, or an empty string if the input is a string or a negative number.

Examples:

```
X = 100
```

```
TextWindow.WriteLine({  
    X.SquareRoot,  
    Math.SquareRoot(16),  
    Math.SquareRoot(20),  
    Math.SquareRoot(-9)  
})
```

```
10  
4  
4.47213595499958  
  
Press any key to continue...
```

Math.Tan(angle) As Double

Gets the tangent of the given angle in radians. Gets the cosine of the given angle in radians. Remember that radian angles can have values between 0 and $2 * \text{Math.Pi}$, and you can use the `Math.GetRadians` method to convert an angle from degrees to radians.

Parameter:

- **angle:**

The angle whose tangent is needed (in radians). This argument can be omitted if you call this method as an extension property of a numeric variable.

Returns:

The tangent of the given angle.

Examples:

```
Pi = Math.Pi
X = Pi / 2
Z = Math.Tan(Pi)
TextWindow.WriteLine({
    X.Tan,
    Math.Tan(3 * X),
    Z,
    Math.Round2(Z, 5),
    Math.Tan(Math.GetRadians(30)),
    Math.Tan(Math.GetRadians(60))
})
```

```
618987100438218
216235710084101
-0.000000000000032310851043327
0.00000
0.577350269189626
1.73205080756889
```

Math.UseRadianAngles As Boolean

Indicates whither or not to consider angle values to be in radian when sent to Sin, Cos and Tan functions or returned from ArcSin, ArcCos, and ArcTan functions.

The default value is True, but you can set it to False to use degrees instead.

Note that this property will also affect how the [Evaluator](#) works.

Examples:

```
Math.UseRadianAngles = True
A = Math.GetRadians(30)
TW.WriteLine({
    "Using angles in radian:",
    Math.Cos(A),           ' 0.886
    Math.Sin(A),           ' 0.5
    Math.Tan(A),           ' 577
    Math.GetDegrees(Math.ArcCos(0.5)),   ' 60
    Math.GetDegrees(Math.ArcSin(0.5)),   ' 30
    Math.GetDegrees(Math.ArcTan(1))     ' 45
})
```

```
Math.UseRadianAngles = False
TW.WriteLine({
    "-----",
    "Using angles in degrees:",
    Math.Cos(30),           ' 0.886
    Math.Sin(30),           ' 0.5
    Math.Tan(30),           ' 577
    Math.ArcCos(0.5),       ' 60
    Math.ArcSin(0.5),       ' 30
    Math.ArcTan(1)          ' 45
})
```

The MenuItem Type

Represents the MenuItem control, which shows a menu item on the main menu bar or on the dropdown list of a parent menu item. The user can click the menu item to perform the task you programmed in the OnClick event handler.

You can also set the [Checkable property](#) to True to allow the user to check or uncheck the menu item, hence you can use the [Checked property](#) and the [OnCheck event](#) to respond the user choices.

The form designer doesn't support adding menu items at design time, but you can use the [MainMenu.AddItem](#) method to add an item to the main menu, or use the [MenuItem.AddItem](#) to add an item to a menu item.

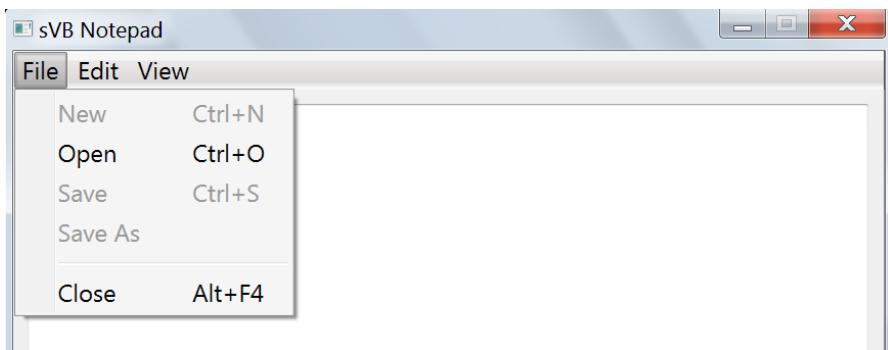
You can also use the [MenuItem.AddSeparator](#) to add a separator line to the menu item.

Note that the MenuItem control inherits all the properties, methods and events of the [Control](#) type.

Besides, the MenuItem control has some new members that are listed below:

① **MenuItem.AddItem(itemName, text, shortcut)** As MenuItem

Adds a sub menu item (a command) to the current menu item, like adding the New, Open, Save, Save As and Close commands to the file menu.



Parameters:

- **itemName:**

The name of the new sub menu item.

- **text:**

The title of the new sub menu item

- **shortcut:**

The keyboard shortcut keys, like Ctrl+N to be displayed on the title of the menu item.

Note that you don't need to program famous windows shortcuts for the TextBox like Ctrl+C, Ctrl+X, Ctrl+V and Ctrl+A, but other shortcuts like Ctrl+N, Ctrl+O and Ctrl+S will not work until you manually program them in the OnPreviewKeyDown event handler of the form, like we did in the frmMain.sb in the "sVB Notepad" project in the samples folder:

```

Sub FrmMain_OnPreviewKeyDown()
    If Keyboard.CtrlPressed = False Then
        Return
    EndIf

    key = Event.LastKey
    If key = Keys.N Then
        MenuNew_OnClick()
        Event.Handled = True
    ElseIf key = Keys.O Then
        MenuOpen_OnClick()
        Event.Handled = True
    ElseIf key = Keys.S Then
        MenuSave_OnClick()
        Event.Handled = True
    ElseIf key = Keys.F Then
        MenuFind_OnClick()
        Event.Handled = True
    ElseIf key = Keys.B Then
        MenuBold.Checked = (MenuBold.Checked = False)
    ElseIf key = Keys.I Then
        MenuItalic.Checked = (MenuItalic.Checked = False)
    ElseIf key = Keys.U Then
        MenuUnderline.Checked =
            (MenuUnderline.Checked = False)
    EndIf
EndSub

```

Returns:

The menu item that have been added.

Example:

You can see this method in action in the "sVB Notepad" project in the samples folder. It is used in global area of the frmMain.sb to add sub items to the main menu items. For example, this is how we add items to the File menu:

```

M = Me.AddMainMenu("mainmenu")
MenuFile = M.AddItem("menuFile", "File", "")
MenuNew = MenuFile.AddItem(
    "menuNew", "New", "Ctrl+N")
MenuNew.OnClick = MenuNew_OnClick
MenuOpen = MenuFile.AddItem(
    "menuOpen", "Open", "Ctrl+O")
MenuOpen.OnClick = MenuOpen_OnClick
MenuSave = MenuFile.AddItem(
    "MenuSave", "Save", "Ctrl+S")
MenuSave.OnClick = MenuSave_OnClick
MenuSaveAs = MenuFile.AddItem(
    "MenuSaveAs", "Save As", "")
MenuSaveAs.OnClick = MenuSaveAs_OnClick
MenuFile.AddSeparator()
MenuClose = MenuFile.AddItem(
    "MenuClose", "Close", "Alt+F4")
MenuClose.OnClick = MenuClose_OnClick
MenuFile.OnOpen = MenuFile_OnOpen

```

MenuItem.AddSeparator()

Adds a separator line between menu items.

Example:

You can see this method in action in the "sVB Notepad" project in the samples folder. It is used in global area of the frmMain.sb to add sub items to the main menu items. For example, this is how we add separators between the commands of the Edit menu:

```

M = Me.AddMainMenu("mainmenu")
MenuEdit = M.AddItem("menuEdit", "Edit", "")
MenuUndo = MenuEdit.AddItem(
    "menuUndo", "Undo", "Ctrl+Z")
MenuUndo.OnClick = MenuUndo_OnClick
MenuRedo = MenuEdit.AddItem(
    "menuRedo", "Redo", "Ctrl+Y")

```

```

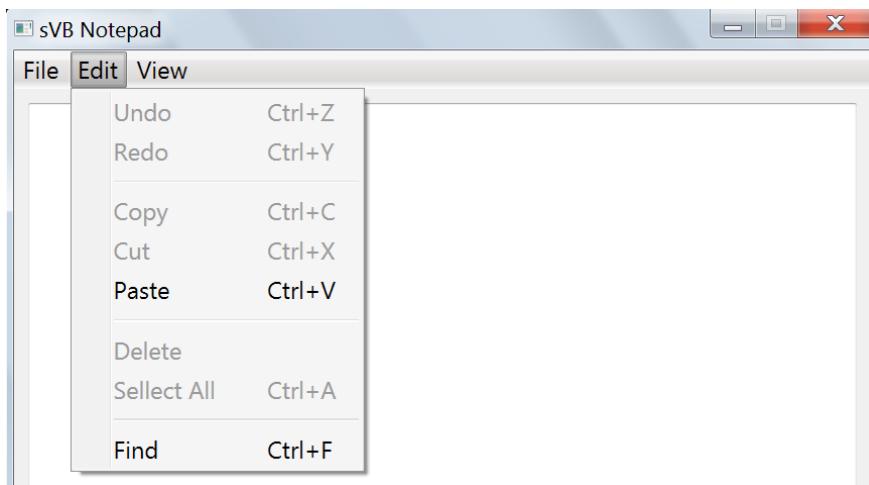
MenuRedo.OnClick = MenuRedo_OnClick
MenuEdit.AddSeparator()

MenuCopy = MenuEdit.AddItem(
    "MenuCopy", "Copy", "Ctrl+C")
MenuCopy.OnClick = MenuCopy_OnClick
MenuCut = MenuEdit.AddItem(
    "MenuCut", "Cut", "Ctrl+X")
MenuCut.OnClick = MenuCut_OnClick
MenuPaste = MenuEdit.AddItem(
    "MenuPaste", "Paste", "Ctrl+V")
MenuPaste.OnClick = MenuPaste_OnClick
MenuEdit.AddSeparator()

MenuDelete = MenuEdit.AddItem(
    "MenuDelete", "Delete", "")
MenuDelete.OnClick = MenuDelete_OnClick
MenuSelAll = MenuEdit.AddItem(
    "MenuSelAll", "Select All", "Ctrl+A")
MenuSelAll.OnClick = MenuSelAll_OnClick
MenuEdit.AddSeparator()

MenuFind = MenuEdit.AddItem(
    "MenuFind", "Find", "Ctrl+F")
MenuFind.OnClick = MenuFind_OnClick
MenuEdit.OnOpen = MenuEdit_OnOpen

```



MenuItem.Checkable As Boolean

When it is True, user can interact with the control.

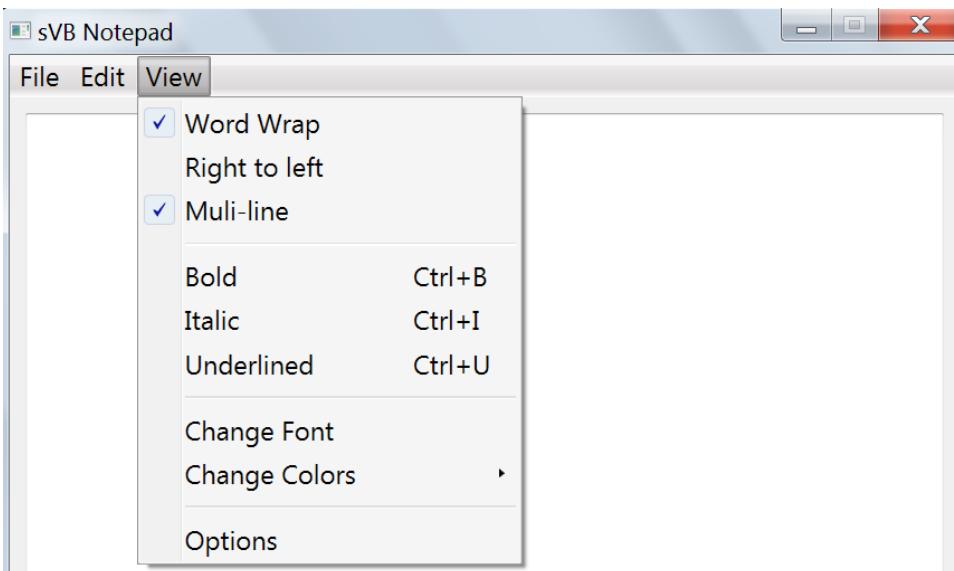
When it is False, the control is disabled, and user can't interact with it.

Example:

You can see this method in action in the "sVB Notepad" project in the samples folder. It is used in global area of the frmMain.sb to add checkable menu items to the View menu:

```
MenuItemView = M.AddItem("MenuItemView", "View", "")  
MenuItemWrap = MenuItemView.AddItem(  
    "MenuItemWrap", "Word Wrap", "")  
MenuItemWrap.Checkable = True  
MenuItemWrap.Checked = True  
MenuItemWrap.OnCheck = MenuItemWrap_OnCheck  
  
MenuItemRtl = MenuItemView.AddItem(  
    "MenuItemRtl", "Right to left", "")  
MenuItemRtl.Checkable = True  
MenuItemRtl.OnCheck = MenuItemRtl_OnCheck  
  
MenuItemMultiline = MenuItemView.AddItem(  
    "MenuItemMultiline", "Multi-line", "")  
MenuItemMultiline.Checkable = True  
MenuItemMultiline.OnCheck = MenuItemMultiline_OnCheck  
MenuItemMultiline.Checked = True  
MenuItemView.AddSeparator()  
  
MenuItemBold = MenuItemView.AddItem(  
    "MenuItemBold", "Bold", "Ctrl+B")  
MenuItemBold.Checkable = True  
MenuItemBold.OnCheck = MenuItemBold_OnCheck  
  
MenuItemItalic = MenuItemView.AddItem(  
    "MenuItemItalic", "Italic", "Ctrl+I")  
MenuItemItalic.Checkable = True  
MenuItemItalic.OnCheck = MenuItemItalic_OnCheck
```

```
MenuUnderline = MenuView.AddItem(  
    "MenuUnderline", "Underlined", "Ctrl+U")  
MenuUnderline.OnCheck = MenuUnderline_OnCheck  
MenuUnderline.Checkable = True
```



➊ MenuItem.Checked As Boolean

When True, the menu item is checked.

When False, the menu item is unchecked.

Note that you need to set the [Checkable property](#) to True, to allow the user to check the menu item.

Example:

You can see this method in action in the "sVB Notepad" project in the samples folder. It is used in the OnCheck event handlers in the frmMain.sb to apply the checkable menu states on the TextBox when the user checks or unchecks a menu item. For example:

```
Sub MenuMuliline_OnCheck()  
    TxtEditor.MultiLine = MenuMuliline.Checked  
EndSub
```

MenuItem.MenuItems As Array

Returns as array containing the child menu items of the current menu item.

Separators will not be included.

MenuItem.OnCheck

This event is fired when the checked state is changed.

For more info, read about [handling control events](#).

Example:

You can see this event in action in the "sVB Notepad" project in the samples folder. It is used in the frmMain.sb to apply the checkable menu states on the TextBox when the user checks or unchecks a menu item. For example:

```
Sub MenuWrap_OnCheck()
    TxtEditor.WordWrap = MenuWrap.Checked
EndSub
```

```
Sub MenuRtl_OnCheck()
    TxtEditor.RightToLeft = MenuRtl.Checked
EndSub
```

MenuItem.OnOpen

This event is fired when the submenu is opened.

For more info, read about [handling control events](#).

Example:

You can see this event in action in the "sVB Notepad" project in the samples folder. It is used in the frmMain.sb to update the checkable menu states according to the TextBox

properties. For example, this is how it is used with the View menu:

```
Sub MenuView_OnOpen()
    MenuRtl.Checked = TxtEditor.RightToLeft
    MenuWrap.Checked = TxtEditor.WordWrap
    MenuMuliline.Checked = TxtEditor.MultiLine
    MenuBold.Checked = TxtEditor.FontBold
    MenuItalic.Checked = TxtEditor.FontItalic
    MenuUnderline.Checked = TxtEditor.Underlined
EndSub
```

The Mouse Type

The mouse class provides allows you to get or set the mouse related properties, like the cursor position, pointer, etc.

The Mouse type has these members:

Mouse.HideCursor()

Hides the mouse cursor on the screen. Use the Mouse.ShowCursor method to show it again.

Mouse.IsLeftButtonDown As Boolean

Returns True if the left button is pressed, otherwise False.

Example:

The following program allows the user to draw on the graphics window when he moves the mouse while clicking the left mouse button:

```
GraphicsWindow.MouseMove = OnMouseMove
GraphicsWindow.MouseDown = OnMouseDown
PrevX = 0
PrevY = 0

Sub OnMouseMove
    If Mouse.IsLeftButtonDown Then
        x = GraphicsWindow.MouseX
        y = GraphicsWindow.MouseY
        GraphicsWindow.DrawLine(PrevX, PrevY, x, y)
        PrevX = x
        PrevY = y
    EndIf
EndSub
```

```
Sub OnMouseDown  
    PrevX = GraphicsWindow.MouseX  
    PrevY = GraphicsWindow.MouseY  
EndSub
```

Mouse.IsRightButtonDown As Boolean

Gets whether or not the right button is pressed.

Mouse.ShowCursor()

Shows the mouse cursors on the screen, after being hidden by the Mouse.HideCursor method.

Mouse.X As Double

Gets or sets the mouse cursor's x co-ordinate relative to the screen.

Note that you can use the [Control.MouseX](#) property to get mouse x-pos relative to the control.

Example:

You can see this property in action in the `Mouse pos` project in the samples folder. It is used in the OnMouseMove event handler of the form to show the mouse position relative to the screen:

```
msg = "Pos relative to Screen: ("  
Label1.Text = msg + Mouse.X + "," + Mouse.Y + ")"
```

But this can't track the mouse movement outside the form. Unfortunately, there is no event to track the mouse move on the screen, but we can workaround this by using a timer to display the mouse position on the screen every 20 milliseconds. You can activate this timer by clicking the `Trace

Screen Pos` button, which by the way sets the Mouse.X property to 0 to move the mouse x-pos to the left edge of the screen, so, the user can notice that he can track the mouse movement even outside the form area:

Mouse.X = 0

Mouse.Y As Double

Gets or sets the mouse cursor's y co-ordinate relative to the screen.

Note that you can use the [Control.MouseY](#) property to get mouse y-pos relative to the control.

Example:

You can see this property in action in the `Mouse pos` project in the samples folder. It is used in the OnMouseMove event handler of the form to show the mouse position relative to the screen:

```
msg = "Pos relative to Screen: ("  
Label1.Text = msg + Mouse.X + "," + Mouse.Y + ")"
```

But this can't track the mouse movement outside the form. Unfortunately, there is no event to track the mouse move on the screen, but we can workaround this by using a timer to display the mouse position on the screen every 20 milliseconds. You can activate this timer by clicking the `Trace Screen Pos` button, which by the way sets the Mouse.Y property to 0 to move the mouse y-pos to the top edge of the screen, so, the user can notice that he can track the mouse movement even outside the form area:

Mouse.Y = 0

The Program Type

The Program class provides helpers to control the program execution.

The Program type has these members:

Program.ArgumentCount As Double

Gets the number of command-line arguments passed to this program.

Program.Delay(milliSeconds)

Delays program execution by the specified amount of milliseconds. You should always try to use this method because it has a better performance, but if any controls freez, you should use the [Program.WinDelay](#) method instead.

Parameter:

- **milliSeconds:**

The amount of delay in milliseconds.

Example:

Add a label on the form, and write this code:

```
For I = 1 To 60
    Label1.Text = I
    Program.Delay(1000)
Next
```

When you run the project, the label will show 1 and increment it every 1 second until it reaches 60.

Program.Directory As String

Gets the directory that contains the executable file (exe) of the program.

Example:

This property is used in every form.sb.gen file to get the path of the form.xaml file, which is used to load the form design in runtime:

```
_Path = Program.Directory + "\Form1.xaml"  
Form1 = Forms.LoadForm("Form1", _Path)
```

Program.End()

Ends the program.

Program.GetArgument(index) As String

Returns the specified argument passed to this program.

Parameter:

- **index:**

Index of the argument.

Returns:

The command-line argument at the specified index.

Program.GetSetting(section, name, defaultValue) As String

Gets a value from the windows registry.

Parameters:

- **section:**

The registry section name that you used when saving the setting.

- **name:**

The registry key that you used when saving the setting.

- **defaultValue:**

The value to return if the data is not found in the registry.

Returns:

A string that contains the required setting.

Example:

Add two textboxes on the form, and write this code:

```
TextBox1.Text = Program.GetSetting(  
    "Form1", "TextBox1", "")  
TextBox2.Text = Program.GetSetting(  
    "Form1", "TextBox2", "")  
  
Sub Form1_OnClosing()  
    Program.SaveSetting(  
        "Form1", "TextBox1", TextBox1.Text)  
    Program.SaveSetting(  
        "Form1", "TextBox2", TextBox2.Text)  
EndSub
```

Press F5 to run the project. The code in the global area will get the settings from the registry and set them to the two

textboxes, but there is no settings saved yet, so the default values will be returned, which are empty strings.

Write anything you want in the textboxes and close the form. The code in the OnClosing event handler will save the values of the two textboxes in the registry, so, when you run the program again, you will see those values in the two textboxes again!

Program.SaveSetting(section, name, value)

Saves a value in the windows registry.

Parameters:

- **section:**

The registry section name, which refers to the category that the data lies under. For example, you may use "Form1" as a section name when you save property values of this form.

- **name:**

The registry key where the data is saved in. Shows a key name that refers to the meaning of the date, like "Width" and "Height" if you are saving the width and the height of the form.

- **value:**

The value to save in the registry.

Example:

See the sample of the [Program.GetSetting](#) method.

Program.WinDelay(milliSeconds)

Delays program execution by the specified amount of milliseconds.

You should use the [Program.Delay](#) method instead, because it has a better performance, but if any controls freeze due to using it, you should use the Program.WinDelay.

Parameter:

- **milliSeconds:**

The amount of delay in milliseconds.

Example:

Add a label on the form, and write this code:

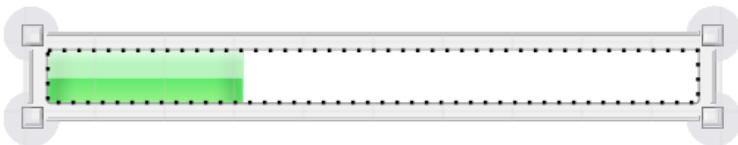
```
For I = 1 To 60
    Label1.Text = I
    Program.WinDelay(1000)
Next
```

When you run the project, the label will show 1 and increment it every 1 second until it reaches 60.

The ProgressBar Type

Represents the ProgressBar control, that indicates how much of the task has done.

Use the Minimum and Maximum properties to set the progress range, and use the Value property to set the current progress. You can use the form designer to add a progress bar to the form by dragging it from the toolbox.



It is also possible to use the [Form.AddProgressBar](#) method to create a new progress bar and add it to the form at runtime. By default, the progress bar is drawn horizontally, but you can rotate it (using the rotation thumb in the form designer, or using the Angle property in code).

Note that the ProgressBar control inherits all the properties, methods and events of the [Control](#) type.

Besides, the ProgressBar control has some new members that are listed below:

ProgressBar.Maximum As Double

Gets or sets the progress maximum value in current ProgressBar. The default value is 100.

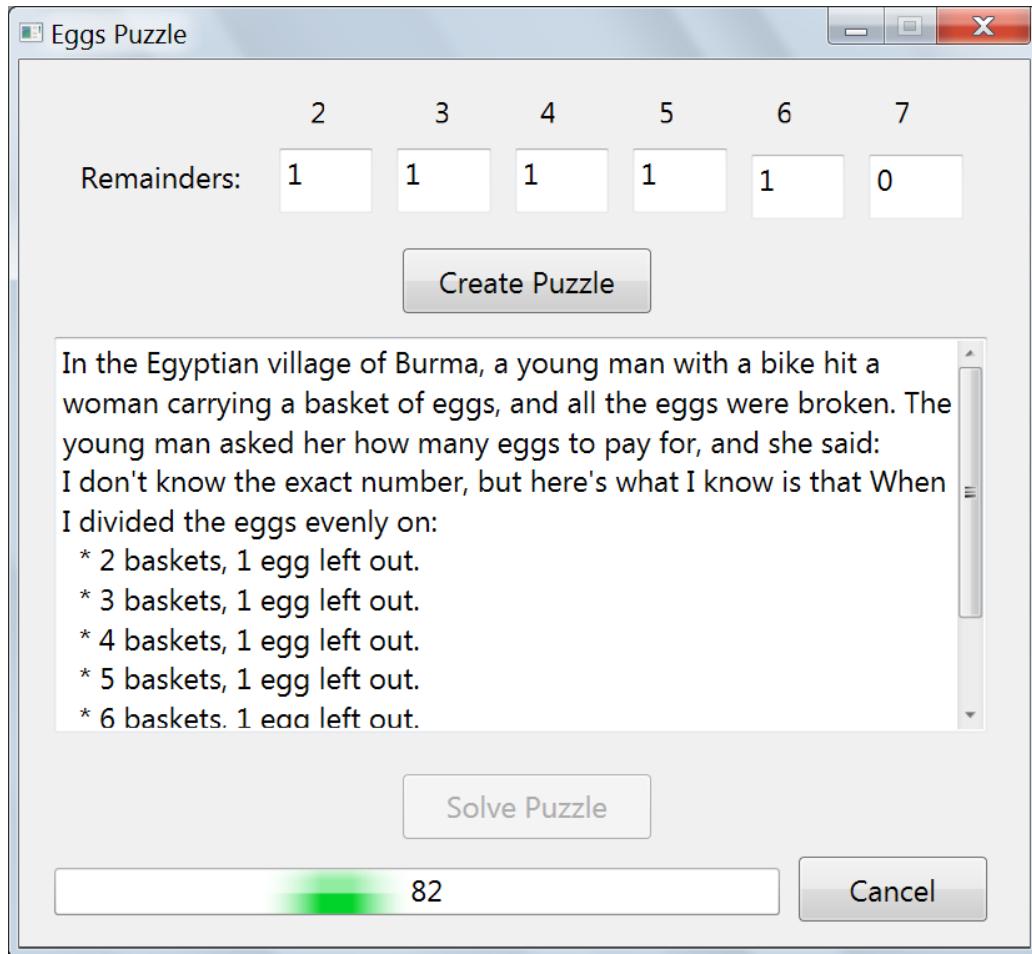
If you set the max value to 0, this will mean you don't know when the progress will end (indeterminate), so, it will show an infinitely moving marquee.

Examples:

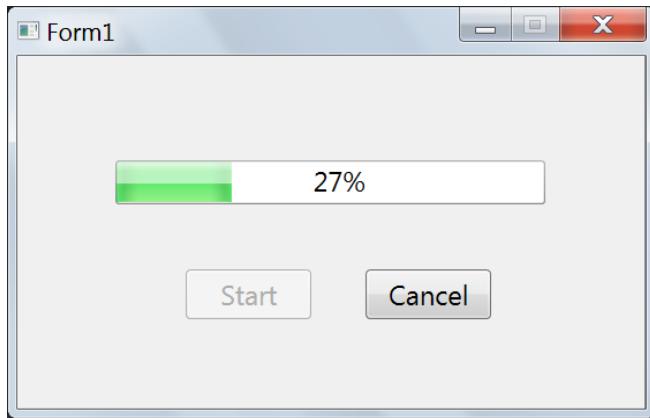
You can see this property in action in the "Eggs Puzzle" project in the samples folder. We set the maximum value of the progress bar to 0

at the form global code area, to show infinitely moving marquee while we trying to solve the puzzle, because we don't know exactly when we will find the result:

ProgressBar1.Maximum = 0



Note that we didn't set the value of this property in the Progress project in the samples folder, which means it will keep its default value (which is 100). This will allow you to use the ProhressBar.Value property to set the current progress, as you see in the below picture. Note that the percentage number you see over the progress bar in both samples is displayed in separate label.



🎨 ProgressBar.Minimum As Double

Gets or sets the progress minimum value in current ProgressBar.
The default value is 0.

🎨 ProgressBar.Value As Double

Gets or sets the progress value in current ProgressBar.

Examples:

You can see this property in action in the Progress project in the samples folder. It is used in the Progress sub to increase the current progress value by 1 , which happens every timer the timer ticks.

```
Sub Progress()
    v = ProgressBar1.Value + 1
    If v <= 100 Then
        ProgressBar1.Value = v
        LblValue.Text = v + "%"
    Else
        Timer1.Pause()
        ProgressBar1.Visible = False
    EndIf
EndSub
```

The RadioButton Type

Represents a RadioButton control, that the user can check or uncheck. Use the Checked property and OnCheck event to respond to the user choices.

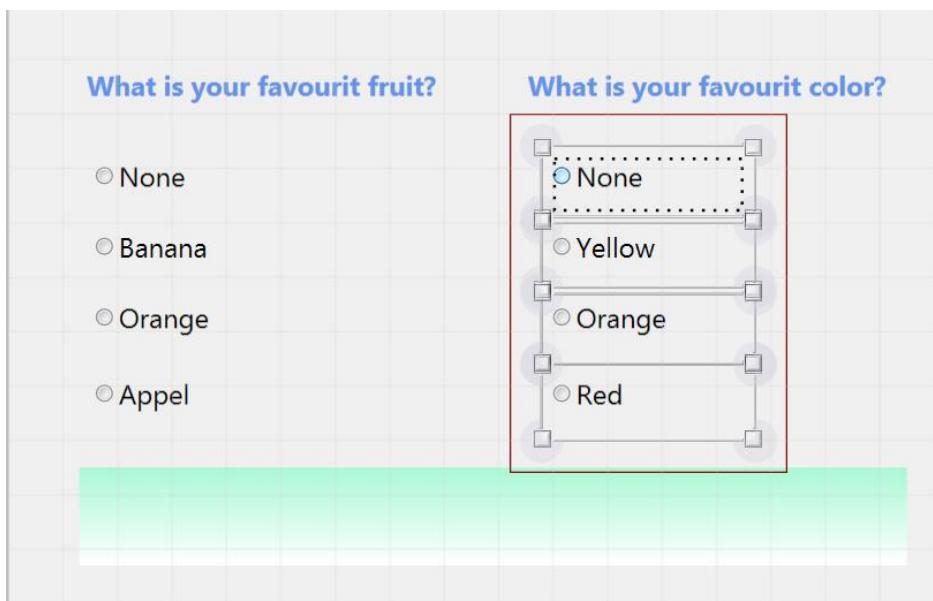
You can use the form designer to add a radio button to the form by dragging it from the toolbox.

It is also possible to use the [Form.AddRadioButton](#) method to create a new radio button and add it to the form at runtime.

Radio buttons work together as a group, where each group can contain only one checked radio button.

By default, all radio buttons you add to the form will be grouped together, but you can group any number of radio buttons together by setting their GroupName property to the same group name.

You can also use the form designer to group radio buttons by selecting them, right-clicking one of the selected radio buttons, and clicking the Group command from the context menu.



Note that the RadioButton control inherits all the properties, methods and events of the [Control](#) type.

Besides, the RadioButton control has some new members that are listed below:

RadioButton.Checked As Boolean

Gets or sets the state of the radio button. It can have one of possible three values:

- **True:** the radio button is checked.
- **False:** the radio button is unchecked.
- **An empty string "":** the radio button is not checked nor unchecked (indeterminate state).

Note that only one radio button in the same group can be Checked, so, when a radio button is checked (either by the user, or by setting this property to true from code), the radio button that was previously checked will be automatically unchecked.

Examples:

You can see this property in action in the "Radio Button sample" project in the samples folder. It is used in the global area of the form to check the default options "No fruit" and "No color" at the start of the program:

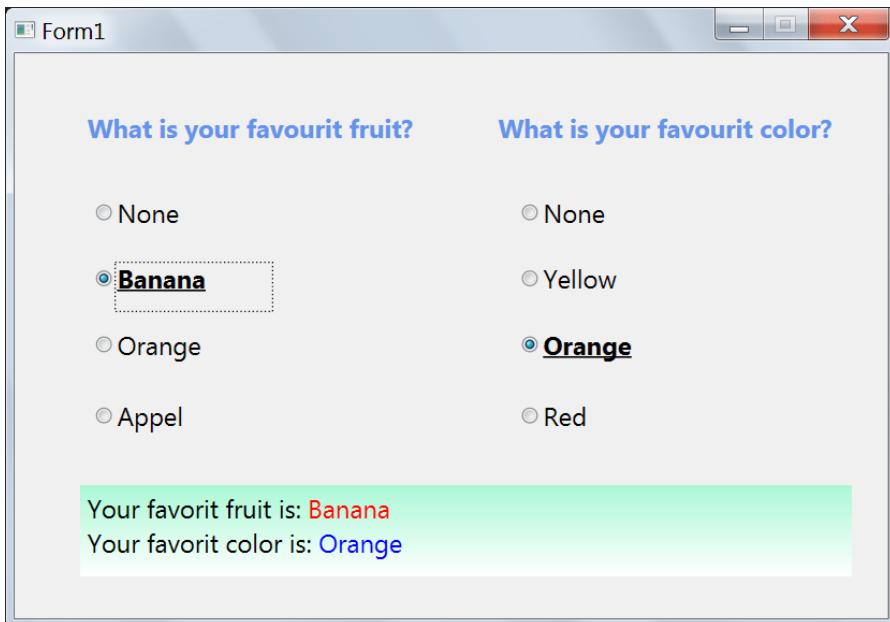
```
RdoNoFruit.Checked = True  
RdoNoColor.Checked = True
```

It is also used in the SelectItem function to change the font style of the checked radio button to bold and underlined, and reset it for the unchecked radio button to the normal style. The SelectItem function is called from the OnCheck event handler of each radio button, which is fired when the state of the radio button is changed (either checked or unchecked):

```

Function SelectItem()
    checkedRadioButton = Event.SenderControl
    If checkedRadioButton.Checked Then
        checkedRadioButton.FontBold = True
        checkedRadioButton.Underlined = True
        Return checkedRadioButton.Text
    Else
        checkedRadioButton.FontBold = False
        checkedRadioButton.Underlined = False
        Return ""
    EndIf
EndFunction

```



RadioButton.GroupName As String

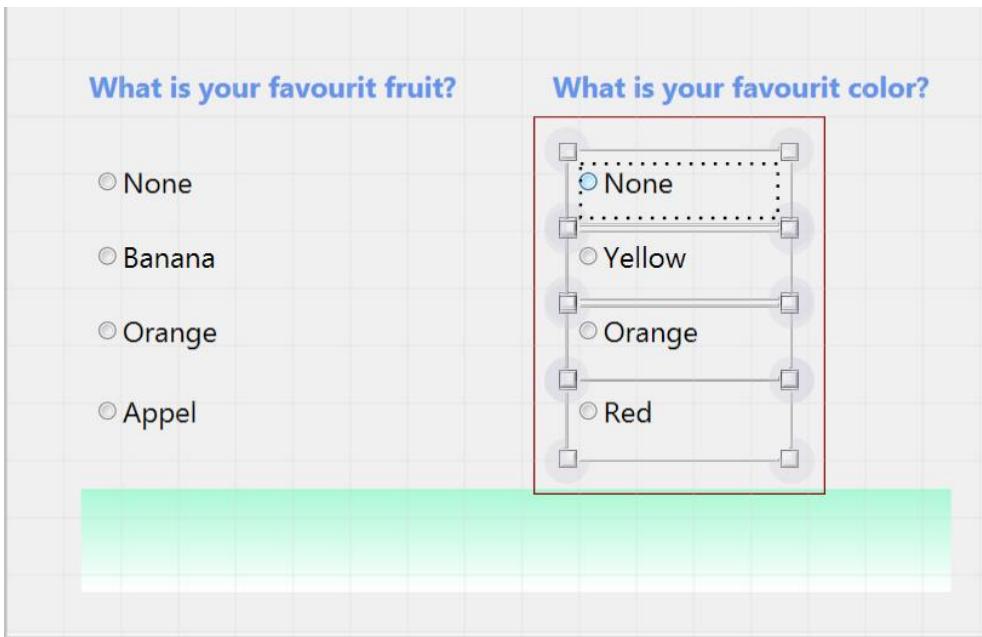
Gets or sets the name of the radio buttons group.

Radio buttons that belong to the same group can have only one selected button at a time, so, if the user checks one button, the previously checked button will be automatically unchecked.

By default, all radio buttons you add to the form will be grouped together to a group with an empty name "", but you can create

as many groups as you need, by changing this property value to the same group name (like group1) in all the radio buttons you want to add to the group.

You can also use the form designer to group radio buttons, by selecting them, then right-clicking one of the selected radio buttons, and clicking the Group command from the context menu. This will give the selected buttons the same group name (which will be auto-created for you). This is what we did in the form designer in the "Radio Button sample" project in the samples folder to create a new group for colors radio buttons, while we left the fruit radio buttons together in the default (unnamed) group:



⚡ RadioButton.OnCheck

This event is fired when the checked state is changed, either checked or unchecked. So, when the user checks a radio button, this event will fire twice:

- Once for the old selected radio button which will be unchecked.
- And another for the newly checked radio button.

Note that this is the default event for the RadioButton control, so, you can double-click any RadioButton on the form designer (say RadioButton1 for example), to get this event handler written for you in the code editor:

```
Sub RadioButton1_OnCheck()
```

```
EndSub
```

For more info, read about [handling control events](#).

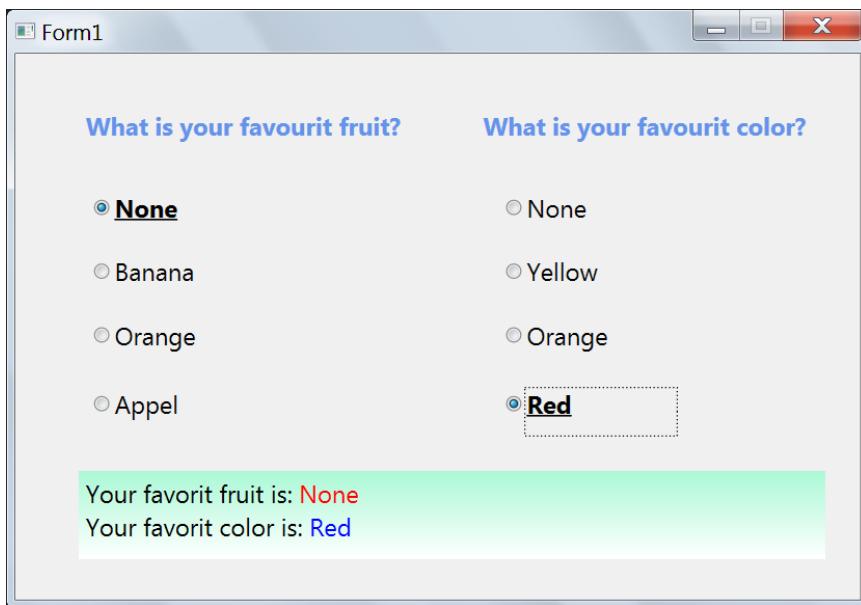
Examples:

You can see this property in action in the "Radio Button sample" project in the samples folder. We used one handler for each radio button group to change the selected radio button style, and set the FavFruit and FavColor variables to the selected fruit and color respectively, which are used in the print subroutine to print the user choices un the label:

```
Sub Fruits_OnCheck()
  x = SelectItem()
  If x <> "" Then
    FavFruit = x
    Print()
  EndIf
EndSub

Sub Colors_OnCheck()
  x = SelectItem()
  If x <> "" Then
    FavColor = x
    Print()
  EndIf
EndSub
```

Note that the `SelectItem` function returns an empty string "" if the event is fired by an unchecked radio button.



🎨 RadioButton.Text As String

Gets or sets the text that is displayed by the RadioButton.

🎨 RadioButton.Underlined As Boolean

Gets or sets whether or not to draw a line under the text that is displayed by the radio button.

🎨 RadioButton.WordWrap As Boolean

Gets or sets whether or not the text will be continue on the next line if it exceeds the width of the radio button. The default value is True, so the radio button title will be wrapped over lines by default. You can use the [RadioButton.FitContentHeight](#)

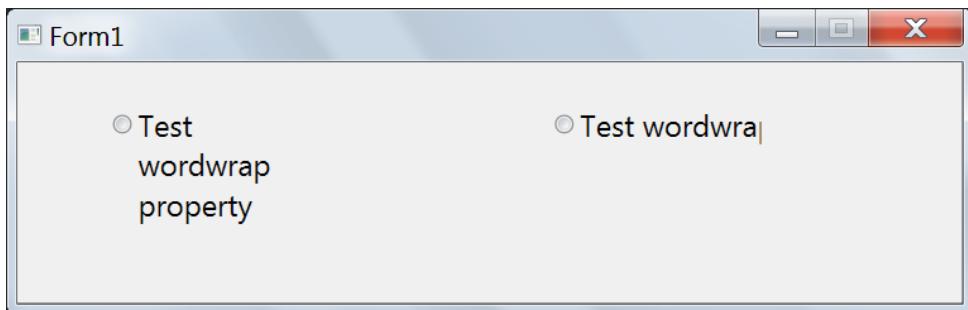
method to ensure that the radio button height fits the wrapped lines.

Example:

Add two radio buttons on the form, and add this code to the form global area:

```
radioButton1.Text = "Test wordwrap property"  
radioButton1.FitContentHeight()  
radioButton2.Text = "Test wordwrap property"  
radioButton2.WordWrap = False
```

The result will be as shown in the picture:



The ScrollBar Type

Represents the ScrollBar control, that allows the user to scroll a value within a range.

Use the Minimum and Maximum properties to set the scroll range.

Use the Value property to set the current scroll position.

Use the OnScroll event to take action when the scroll position changes.

You can use the form designer to add a scroll bar to the form by dragging it from the toolbox.

It is also possible to use the [Form.AddScrollBar](#) method to create a new scroll bar and add it to the form at runtime.

Note that the ScrollBar control inherits all the properties, methods and events of the [Control](#) type.

Besides, the ScrollBar control has some new members that are listed below:

ScrollBar.Maximum As Double

Gets or sets the maximum value of the current ScrollBar. The default value is 100.

ScrollBar.Minimum As Double

Gets or sets the minimum value of the current ScrollBar. The default value is 0.

ScrollBar.OnScroll

This event is fired when the scrollbar value changes.

For more info, read about [handling control events](#).

ScrollBar.Value As Double

Gets or sets the value of current ScrollBar.

Example:

Add a scroll bar and a label on the form, and add this code to the form global area:

```
Me.Top = 0
ScrollBar1.Minimum = 150
ScrollBar1.Maximum = 600
ScrollBar1.OnScroll = ScrollBar1_OnScroll

Sub ScrollBar1_OnScroll()
    Label1.Text = Math.Round(ScrollBar1.Value)
    Me.Height = ScrollBar1.Value
EndSub
```

Press F5 to run the project, and drag the scroll bar thumb or click its arrows to change its value. The form height will change to the new value, and the label will display it.

We set the Minimum and Maximum properties of the scroll bar to make sure that the form height will not be less than 150 nor greater than 600.



The Shapes Type

The Shape object allows you to add, move and rotate shapes on the Graphics window.

The Shapes type has these members:

Shapes.AddEllipse(width, height) As Shape

Adds an ellipse shape with the specified width and height.

Parameters:

- **width:**

The width of the ellipse shape.

- **height:**

The height of the ellipse shape.

Returns:

The name of the ellipse shape that was just added to the Graphics Window. You can send this name to the Move, Rotate and other methods to apply them on the shape, or you can treat the variable that holds this name as a shape object and call these methods directly via it, like:

```
s.Move(200, 100)
```

Examples:

```
GraphicsWindow.PenColor = Colors.Black  
GraphicsWindow.PenWidth = 5  
GraphicsWindow.BrushColor = Colors.Red  
Shapes.AddEllipse(200, 100)
```

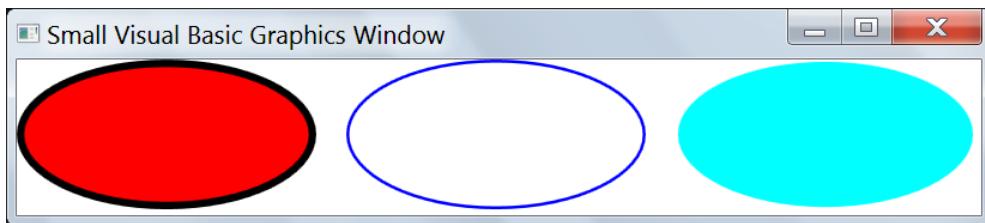
```
GraphicsWindow.PenColor = Colors.Blue  
GraphicsWindow.PenWidth = 2
```

```

GraphicsWindow.BrushColor = Colors.None
E2 = Shapes.AddEllipse(200, 100)
Shapes.Move(E2, 220, 0)

GraphicsWindow.PenColor = Colors.Transparent
GraphicsWindow.PenWidth = 3
GraphicsWindow.BrushColor = Colors.Cyan
E3 = Shapes.AddEllipse(200, 100)
E3.Move(440, 0)

```



Shapes.AddGeometricPath() As Shape

Adds the geometric path, last created using the GeometricPath type, to shapes. This will draw the path on the graphics window using its pen and brush.

Returns:

The name of the geometric path that was just added to the Graphics Window. You can send this name to the Move, Rotate and other methods to apply them on the shape, or you can treat the variable that holds this name as a shape object and call these methods directly via it, like:

s.Move(200, 100)

Example:

You can see this method in action in the "Geometric Path" project in the samples folder. It is used to draw three geometric paths on the screen. For example, this is the part of the code that draws the blue figure you see in the

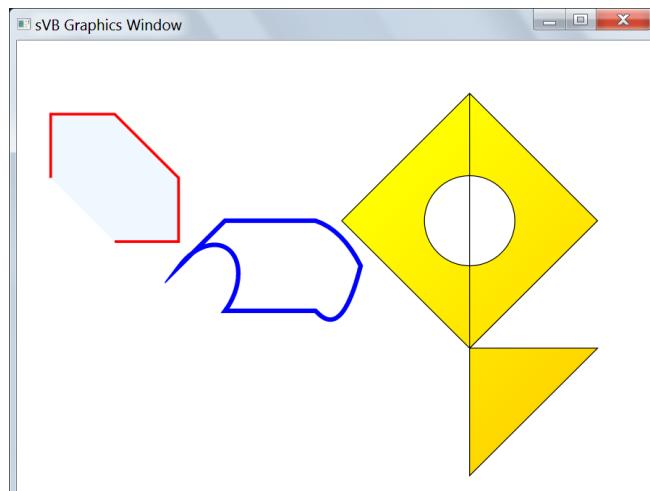
following picture:

```
Create a new empty geometric path
GeometricPath.CreatePath()
GeometricPath.CreateFigure(100, 100, True)
GeometricPath.AddLineSegment(50, 150, True)
GeometricPath.AddArcSegment(
    100, 200,      ' End point
    20, 30,        ' Radius
    30,            ' Angle
    False, True, True)

GeometricPath.AddLineSegment(200, 200, True)
GeometricPath.AddBezierSegment(
    210, 210, ' First control point
    230, 230, ' Seccond control point
    250, 150, ' End point
    True)

GeometricPath.AddQuadraticBezierSegment(
    230, 110, ' Contol point
    200, 100, ' End point
    True)

GW.PenColor = Colors.Blue
GW.PenWidth = 5
GW.BrushColor = Colors.None
Sh2 = Shapes.AddGeometricPath()
Sh2.Move(130, 100)
```



Shapes.AddImage(imageName) As Shape

Adds an image as a shape that can be moved, animated or rotated.

Parameter:

- **imageName:**

The name of the image to draw. This is the name that you get from the [ImageList.LoadImage](#) method.

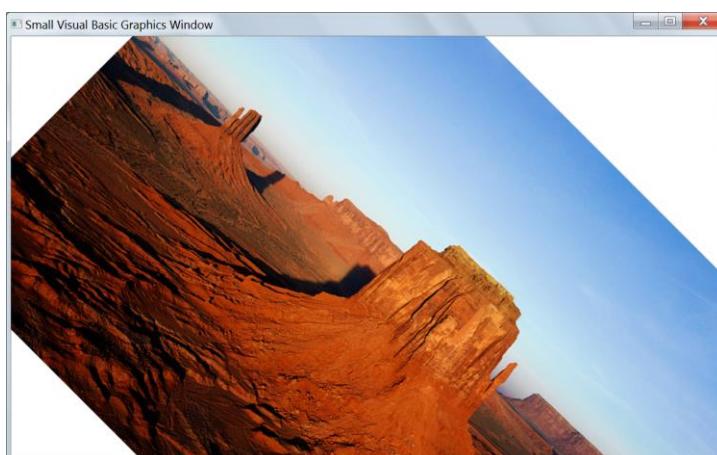
Returns:

The name of the image that was just added to the Graphics Window. You can send this name to the Move, Rotate and other methods to apply them on the shape, or you can treat the variable that holds this name as a shape object and call these methods directly via it, like:

s.Move(200, 100)

Example:

```
Path = "%PUBLIC%\Pictures\Sample  
Pictures\Desert.jpg"  
Image1 = ImageList.LoadImage(Path)  
Img = Shapes.AddImage(Image1)  
Img.Rotate(45)
```



Shapes.AddLine(x1, y1, x2, y2) As Shape

Adds a line between the specified two points.

Parameters:

- **x1:**

The x co-ordinate of the first point.

- **y1:**

The y co-ordinate of the first point.

- **x2:**

The x co-ordinate of the second point.

- **y2:**

The y co-ordinate of the second point.

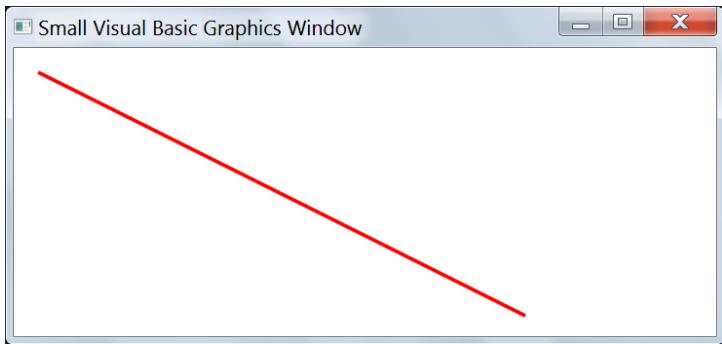
Returns:

The name of the line that was just added to the Graphics Window. You can send this name to the Move, Rotate and other methods to apply them on the shape, or you can treat the variable that holds this name as a shape object and call these methods directly via it, like:

s.Move(200, 100)

Example:

```
GraphicsWindow.PenColor = Colors.Red
GraphicsWindow.PenWidth = 3
Line = Shapes.AddLine(
    0, 0,
    400, 200
)
Line.Move(20, 20)
```



Shapes.AddPolygon(pointsArr) As Shape

Adds a polygon shape represented by the given points array.

Parameter:

- **pointsArr:**

An array of points representing the heads of the polygon. Each item in this array is an array containing the x and y of the point.

Returns:

The name of the polygon shape that was just added to the Graphics Window. You can send this name to the Move, Rotate and other methods to apply them on the shape, or you can treat the variable that holds this name as a shape object and call these methods directly via it, like:

`s.Move(200, 100)`

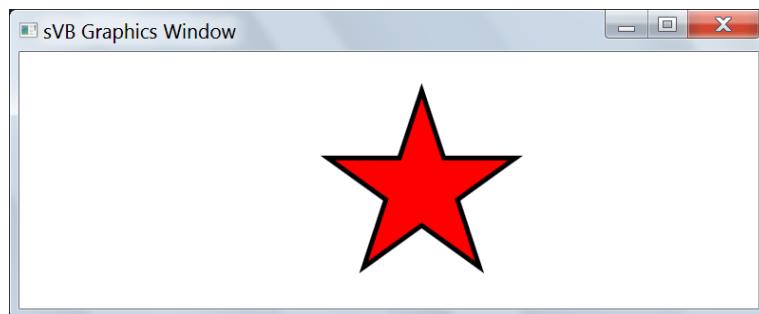
Example:

You can see this method in action in the "Draw Pentagon" project in the samples folder. This is the code that draws the red star, which is represented by 10 points:

```

StarPoints = {
    {40.00, 0.50},
    {49.34, 28.76},
    {79.50, 28.76},
    {55.10, 46.24},
    {64.35, 74.50},
    {40.00, 57.02},
    {15.60, 74.50},
    {24.90, 46.24},
    {0.50, 28.76},
    {30.66, 28.76}
}
GraphicsWindow.PenColor = Colors.Black
GraphicsWindow.PenWidth = 2
GraphicsWindow.BrushColor = Colors.Red
Star = Shapes.AddPolygon(StarPoints)
Star.Move(300, 70)
Star.Zoom(2, 2)

```



⚙️ Shapes.AddRectangle(width, height) As Shape

Adds a rectangle shape with the specified width and height.

Parameters:

- **width:**

The width of the rectangle shape.

- **height:**

The height of the rectangle shape.

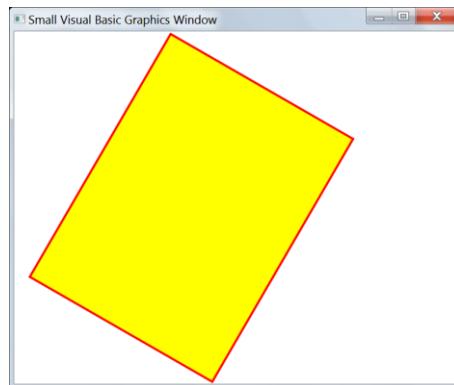
Returns:

The name of the rectangle shape that was just added to the Graphics Window. You can send this name to the Move, Rotate and other methods to apply them on the shape, or you can treat the variable that holds this name as a shape object and call these methods directly via it, like:

```
s.Move(200, 100)
```

Example:

```
GraphicsWindow.PenColor = Colors.Red  
GraphicsWindow.PenWidth = 3  
GraphicsWindow.BrushColor = Colors.Yellow  
R = Shapes.AddRectangle(300, 400)  
Shapes.Move(R, 100, 50)  
R.Rotate(30)
```



Shapes.AddText(text) As Shape

Adds some text as a shape that can be moved, animated or rotated.

Parameter:

- **text:**

The text to add.

Returns:

The name of the text shape that was just added to the Graphics Window. You can send this name to the Move, Rotate and other methods to apply them on the shape, or you can treat the variable that holds this name as a shape object and call these methods directly via it, like:

s.Move(200, 100)

Example:

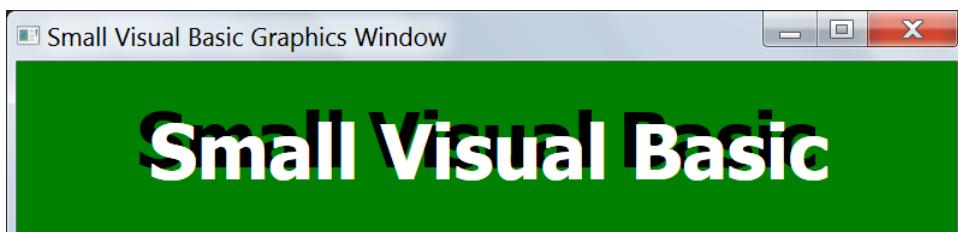
```
GraphicsWindow.BackgroundColor = Colors.Green  
GraphicsWindow.FontBold = True  
GraphicsWindow.FontSize = 50
```

'Draw a black shadow text

```
GraphicsWindow.PenColor = Colors.Black  
GraphicsWindow.BrushColor = Colors.Black  
Txt = Shapes.AddText("Small Visual Basic")  
Shapes.Move(Txt, 80, 20)
```

'Draw white text shifted by (8,8)

```
GraphicsWindow.PenColor = Colors.White  
GraphicsWindow.BrushColor = Colors.White  
Txt = Shapes.AddText("Small Visual Basic")  
Txt.Move(88, 28)
```



Shapes.AddTriangle(x1, y1, x2, y2, x3, y3) As Shape

Adds a triangle shape represented by the specified points.

Parameters:

- **x1:**

The x co-ordinate of the first point.

- **y1:**

The y co-ordinate of the first point.

- **x2:**

The x co-ordinate of the second point.

- **y2:**

The y co-ordinate of the second point.

- **x3:**

The x co-ordinate of the third point.

- **y3:**

The y co-ordinate of the third point.

Returns:

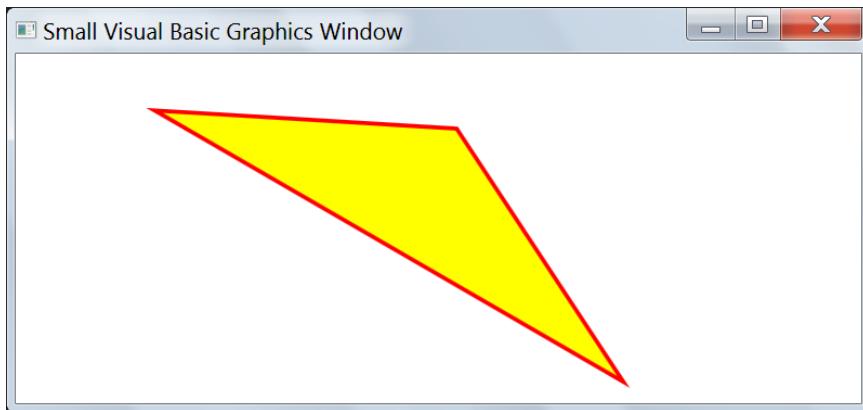
The name of the triangle shape that was just added to the Graphics Window. You can send this name to the Move, Rotate and other methods to apply them on the shape, or you can treat the variable that holds this name as a shape object and call these methods directly via it, like:

s.Move(200, 100)

Example:

```
GraphicsWindow.PenColor = Colors.Red  
GraphicsWindow.PenWidth = 3  
GraphicsWindow.BrushColor = Colors.Yellow
```

```
R = Shapes.AddTriangle(  
    200, 0,  
    0, 100,  
    400, 100  
)  
R.Move(100, 50)  
R.Rotation(30)
```



⚙️ Shapes.Animate(shapeName, x, y, duration)

Animates a shape with the specified name to a new position.

Parameters:

- **shapeName:**

The name of the shape to move. You can omit this argument when you call this method from a Shape variable.

- **x:**

The x co-ordinate of the new position.

- **y:**

The y co-ordinate of the new position.

- **duration:**

The time for the animation, in milliseconds.

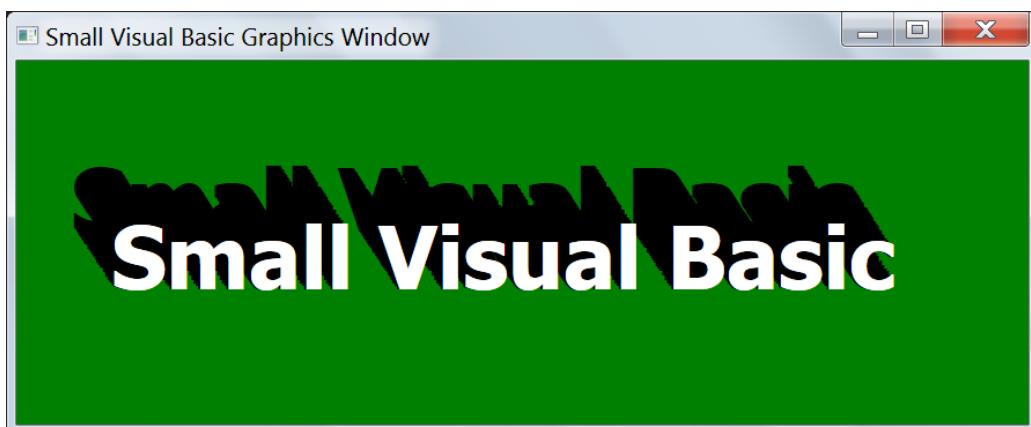
Example:

The following program will draw the text "Small Visual Basic" twice with a black and white colors, and animate the two texts with a small shift in duration and destination point, so, you can see that a flat text that is turning to a 3D text slowly:

```
GraphicsWindow.Width = 700
GraphicsWindow.BackgroundColor = Colors.Green
GraphicsWindow.FontBold = True
GraphicsWindow.FontSize = 60

'Draw a black shadow texts
GraphicsWindow.PenColor = Colors.Black
GraphicsWindow.BrushColor = Colors.Black
For I = 1 To 25
    Txt = Shapes.AddText("Small Visual Basic")
    Shapes.Animate(Txt, 92, 142, 2000 + I * 50)
Next

' Draw white text shifted by (8,8)
GraphicsWindow.PenColor = Colors.White
GraphicsWindow.BrushColor = Colors.White
Txt = Shapes.AddText("Small Visual Basic")
Txt.Animate(100, 150, 2000)
```



Shapes.GetLeft(shapeName) As Double

Gets the left co-ordinate of the specified shape.

Parameter:

- **shapeName:**

The name of the shape. You can omit this argument when you use this method as an extension property with the name Left of a Shape variable.

Returns:

The left co-ordinate of the shape.

Example:

When you run this code, the first message box will show 0, because the rectangle is added at the points (0,0) by default, then the second message box will show 200, which is the left position of the rectangle after calling the Move method:

```
Rect = Shapes.AddRectangle(100, 50)
GW.ShowMessage(Rect.Left, "Left")
Rect.Move(200, 150)
GW.ShowMessage(Rect.Left, "Left")
```

Shapes.GetOpacity(shapeName) As Double

Gets the opacity of a shape.

Parameter:

- **shapeName:**

The name of the shape. You can omit this argument when you use this method as an extension property with the name Opacity of a Shape variable.

Returns:

The opacity of the object as a value between 0 and 100, where 0 is completely transparent and 100 is completely opaque.

Shapes.GetText(shapeName) As String

Gets the text of a text shape added by the AddText method.

Parameters:

- **shapeName:**

The name of the text shape. You can omit this argument when you use this method as an extension property with the name Text of a Shape variable.

Returns:

The shape text if it is a text shape, or "" otherwise.

Shapes.HideShape(shapeName)

Hides an already added shape.

Parameter:

- **shapeName:**

The name of the shape. You can omit this argument when you use this method as an extension method with the name Hide of a Shape variable.

Shapes.Move(shapeName, x, y)

Moves the shape with the specified name to a new position.

Parameters:

- **shapeName:**

The name of the shape to move. You can omit this argument when you use this method as an extension method of a Shape variable.

- **x:**

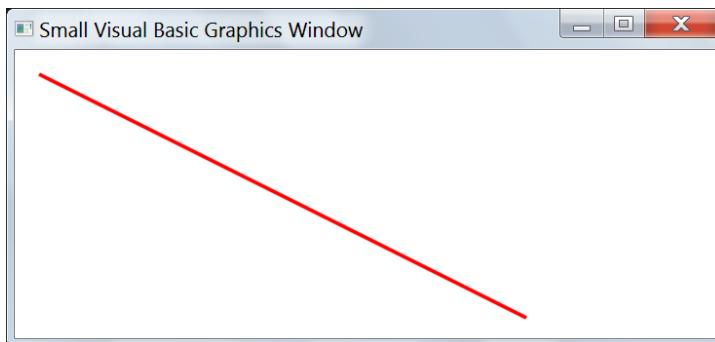
The x co-ordinate of the new position.

- **y:**

The y co-ordinate of the new position.

Example:

```
GraphicsWindow.PenColor = Colors.Red  
GraphicsWindow.PenWidth = 3  
Line = Shapes.AddLine(  
    0, 0,  
    400, 200  
)  
Line.Move(20, 20)
```



Shapes.Remove(shapeName)

Removes a shape from the Graphics Window.

Parameter:

- **shapeName:**

The name of the shape that needs to be removed. You can omit this argument when you use this method as an extension method of a Shape variable.

Shapes.Rotate(shapeName, angle)

Rotates the shape with the specified name to the specified angle.

Parameters:

- **shapeName:**

The name of the shape to rotate. You can omit this argument when you use this method as an extension method of a Shape variable.

- **angle:**

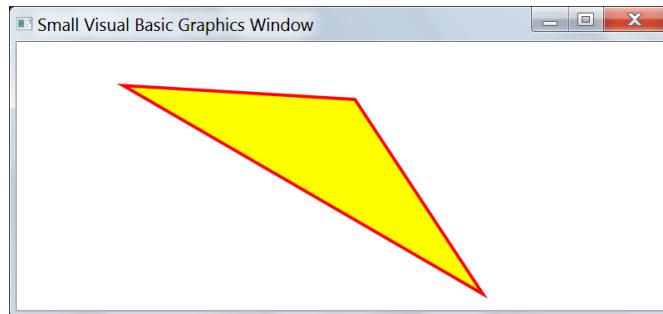
The angle to rotate the shape.

Example:

```
GraphicsWindow.PenColor = Colors.Red
GraphicsWindow.PenWidth = 3
GraphicsWindow.BrushColor = Colors.Yellow
R = Shapes.AddTriangle(
    200, 0,
    0, 100,
    400, 100
)
```

R.Move(100, 50)

R.Rotate(30)



⚙️ **Shapes.SetLeft(shapeName, x)**

Changes the left co-ordinate of the specified shape.

Parameter:

- **shapeName:**

The name of the shape. You can omit this argument when you use this method as an extension property with the name Left of a Shape variable.

- **x:**

The new value of the shape left pos.

Example:

```
Rect = Shapes.AddRectangle(100, 50)
```

```
Rect.Left = 100
```

⚙️ **Shapes.GetTop(shapeName) As Double**

Gets the top co-ordinate of the specified shape.

Parameter:

- **shapeName:**

The name of the shape. You can omit this argument when

you use this method as an extension property with the name Top of a Shape variable.

Returns:

The top co-ordinate of the shape.

Example:

When you run this code, the first message box will show 0, because the rectangle is added at the points (0,0) by default, then the second message box will show 150, which is the top position of the rectangle after calling the Move method:

```
Rect = Shapes.AddRectangle(100, 50)
GW.ShowMessage(Rect.Top, "Top")
Rect.Move(200, 150)
GW.ShowMessage(Rect.Top, "Top")
```

Shapes.SetOpacity(shapeName, level)

Sets how opaque a shape should render.

Parameters:

- **shapeName:**

The name of the shape. You can omit this argument when you use this method as an extension property with the name Opacity of a Shape variable.

- **level:**

The opacity level ranging from 0 to 100, where 0 is completely transparent and 100 is completely opaque.

Example:

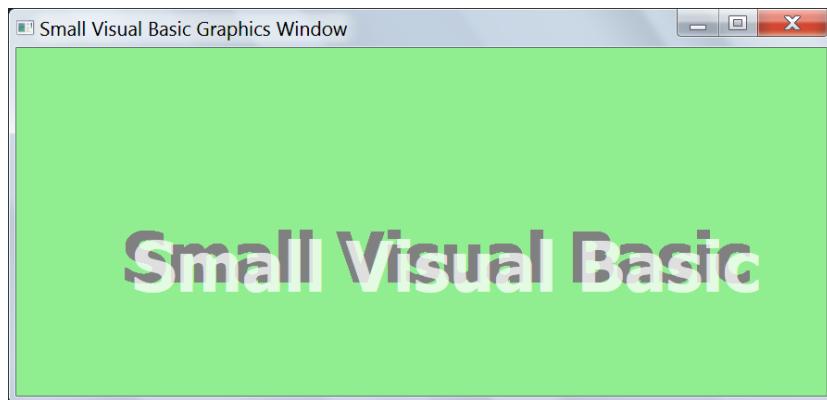
The following program will draw the text "Small Visual Basic" twice with a black and white colors, and animate the two

texts with a small shift in duration and destination point, so, you can see that a flat text that is turning to a 3D text slowly. The white text in front is 25% transparent, which gives a lovely effect:

```
GraphicsWindow.Width = 700
GraphicsWindow.BackgroundColor = Colors.LightGreen
GraphicsWindow.FontBold = True
GraphicsWindow.FontSize = 60

'Draw a black shadow texts
GraphicsWindow.PenColor = Colors.Gray
GraphicsWindow.BrushColor = Colors.Gray
For I = 1 To 25
    Txt = Shapes.AddText("Small Visual Basic")
    Shapes.Animate(Txt, 92, 142, 3000 + I * 100)
Next

' Draw white text shifted by (8,8)
GraphicsWindow.PenColor = Colors.White
GraphicsWindow.BrushColor = Colors.White
Txt = Shapes.AddText("Small Visual Basic")
Txt.Opacity = 75
Txt.Animate(100, 150, 3000)
```



Shapes.SetText(shapeName, text)

Sets the text of a text shape added by the AddText method.

Parameters:

- **shapeName:**

The name of the text shape. You can omit this argument when you use this method as an extension property with the name Text of a Shape variable.

- **text:**

The new text value to set.

Shapes.SetTop(shapeName, y)

Changes the top co-ordinate of the specified shape.

Parameter:

- **shapeName:**

The name of the shape. You can omit this argument when you use this method as an extension property with the name Top of a Shape variable.

- **y:**

The new value of the shape top pos.

Example:

```
Rect = Shapes.AddRectangle(100, 50)  
Rect.Top = 100
```

Shapes.ShowShape(shapeName)

Shows a previously hidden shape.

Parameter:

- **shapeName:**

The name of the shape. You can omit this argument when you use this method as an extension method with the name Show of a Shape variable.

Shapes.Zoom(shapeName, scaleX, scaleY)

Scales the shape using the specified zoom levels.

Parameters:

- **shapeName:**

The name of the shape to zoom. You can omit this argument when you use this method as an extension method of a Shape variable.

- **scaleX:**

The x-axis zoom level. Minimum is 0.1 and maximum is 20.

- **scaleY:**

The y-axis zoom level. Minimum is 0.1 and maximum is 20.

Example:

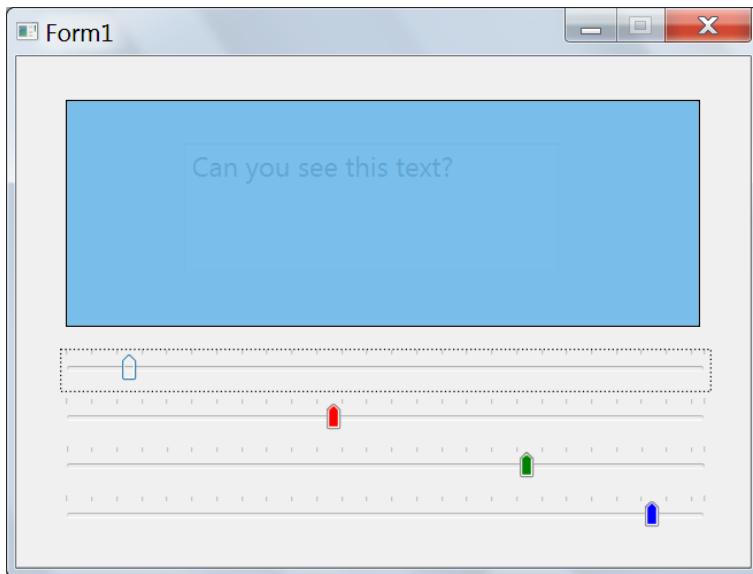
The following program draws a rectangle and allows the user to zoom it in and out by 5% each time he presses the + and - keys from the keyboard:

```
Rec = Shapes.AddRectangle(200, 100)
Z = 1
GraphicsWindow.KeyDown = OnKeyDown
```

```
Sub OnKeyDown()
    If Keyboard.LastKey = Keys.OemPlus Then
        Z = Z * 1.05
        If Z > 20 Then
            Z = 20
        EndIf
        Rec.Zoom(Z, Z)
    ElseIf Keyboard.LastKey = Keys.OemMinus Then
        Z = Z / 1.05
        If Z < 0.1 Then
            Z = 0.1
        EndIf
        Rec.Zoom(Z, Z)
    EndIf
EndSub
```

The Slider Type

Represents the Slider control, that allows the user to choose a value within a range. The following picture shows 4 sliders used to compose a color from its ARGB components, where each component accepts values in the 0-255 range. You can find this design in the "Slider Color Composer" project in the samples folder.



It is easy to work with sliders:

- Use the Minimum and Maximum properties to set the slider range.
- Use the Value property to set the current slide position.
- Use the OnSlide event to take action when the slider value changes.

You can use the form designer to add a slider to the form by dragging it from the toolbox.

It is also possible to use the [Form.AddSlider](#) method to create a new slider and add it to the form at runtime.

Note that the Slider control inherits all the properties, methods and events of the [Control](#) type.

Besides, the Slider control has some new members that are listed below:

Slider.Maximum As Double

Gets or sets the maximum value of the current Slider. The default value is 100.

Slider.Minimum As Double

Gets or sets the minimum value of the current Slider. The default value is 0.

Slider.OnSlide

This event is fired when the slider value changes.

For more info, read about [handling control events](#).

Example:

You can see this event in action in the "Slider Color Composer" project in the samples folder. We used on handler to handle this event for the 4 sliders, to compose the color from their values:

```
SldTrans.OnSlide = OnSlide  
SldRed.OnSlide = OnSlide  
SldGreen.OnSlide = OnSlide  
SldBlue.OnSlide = OnSlide  
  
Sub OnSlide()  
    LblPreview.BackColor = Color.FromArgb(  
        255 - SldTrans.Value,  
        SldRed.Value,  
        SldGreen.Value,  
        SldBlue.Value  
    )  
EndSub
```

Slider.ThumbColor As Color

Gets or sets the color used to draw the thumb of the slider.

Example:

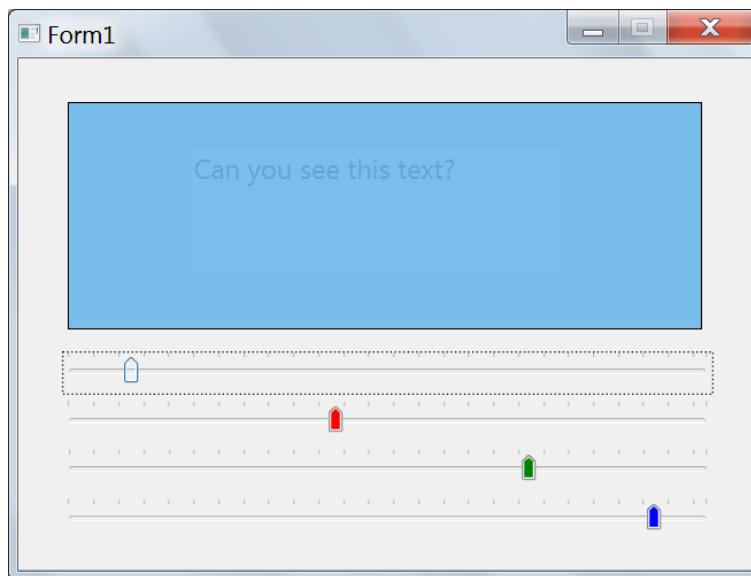
You can see this event in action in the "Slider Color Composer" project in the samples folder. It is used to give the thumb of each slider the color that it represents:

```
SldTrans.ThumbColor = Colors.Transparent
```

```
SldRed.ThumbColor = Colors.Red
```

```
SldGreen.ThumbColor = Colors.Green
```

```
SldBlue.ThumbColor = Colors.Blue
```



Slider.TickFrequency As Double

Gets or sets the distance between slide ticks. For example, if you set this property to 5 on a 0-100 range slider, it will show 20 ticks.

Note that you can change the ticks color by using the ForeColor property, so you can hide the ticks by setting the ForeColor property to Colors.None or Colors.Transparent.

Slider.SnapToTick As Boolean

Gets or sets whether or not the thumb movement snaps to tick marques when the user slides it.

Set this property to True and set the Tick frequency property to a proper value to show marques and force the user to slide only to marquee positions.

The default value is False, which gives the freedom to the user to slide to positions between the marques.

Slider.TrackColor As Color

Gets or sets the color used to draw the track of the slider bar.

Slider.Value As Double

Gets or sets the value of current Slider, which indicates the thumb position on the slider track.

The Sound Type

The Sound object provides operations that allow the playback of sounds.

Some sample sounds are provided along with the library.

The Sound type has these members:

Sound.Load(filePath) As Sound

Loads an audio file, so you can call the Play method to play it later without suffering from any initial delay.

Parameter:

- **filePath:**

The path for the audio file. This could either be a local file (like c:\music\track1.mp3) or a file on the network (like http://contoso.com/track01.wma)

You can load mp3, wav and wma files. Other file formats may or may not be valid depending on the audio codecs installed on the user's computer.

Returns:

- If the audio file is loaded correctly, this method will return the file path, so you can send it to the Play, Pause and Stop methods. Note that sVB treats this path as a sound object, so you can call the Play, pause and Stop methods directly from it.
- Otherwise this method will return an empty string.

Example:

```
chimes = Sound.Load("C:\Windows\Media\chimes.wav")
Sound.Play(chimes)
```

Sound.Pause(filePath)

Pauses playback of an audio file. If the file was not already playing, this operation will not do anything.

Parameter:

- **filePath:**

The path for the audio file like "c:\music\track1.mp3". You can omit this argument when you call this method from a Sound variable.

Example:

```
Track1 = Sound.Load("c:\music\track1.mp3")
Track1.Play()
Program.Delay(10)
Track1.Pause()
```

Sound.Play(filePath)

Plays the given audio file.

If the file is currently played or paused, this operation will replay it from the start. Use the Resume method to continue playing the paused file.

Parameter:

- **filePath:**

The path for the audio file like "c:\music\track1.mp3".

You can omit this argument when you call thid method from a sound object.

Note that if the file exists in the same folder of your application, you can use a relative path like "track1.mp3" with no need to mention the application directory path. This also works if the file exists in a subfolder inside your application's

folder, hence you can use a relative path starting with the subfolder like "sounds\track1.mp3".

The file type could be an mp3, wav or wma file. Other file formats may or may not play depending on the audio codecs installed on the user's computer.

Example:

```
Track1 = Sound.Load("c:\music\track1.mp3")
Track1.Play()
```

Sound.PlayAndWait(filePath)

Plays an audio file and waits until it is finished playing. This could be an mp3, wav or wma file. Other file formats may or may not play depending on the audio codecs installed on the user's computer.

If the file was already paused, this operation will resume from the position where the playback was paused.

Parameter:

- **filePath:**

The path for the audio file like "c:\music\track1.mp3". You can omit this argument when you call this method from a sound object.

Note that if the file exists in the same folder of your application, you can use a relative path like "track1.mp3" with no need to mention the application directory path. This also works if the file exists in a subfolder inside your application's folder, hence you can use a relative path starting with the subfolder like "sounds\track1.mp3".

Example:

```
Track1 = Sound.Load("c:\music\track1.mp3")
Track1.PlayAndWait()
```

Sound.PlayBeep()

Plays the beep beep sound.

Sound.PlayBeepAndWait()

Plays the beep beep sound.

Sound.PlayBellRing()

Plays the bell ring sound and waits for it to finish.

Sound.PlayBellRingAndWait()

Plays the Bell ring sound and waits for it to finish.

Sound.PlayChime()

Plays the chime sound.

Sound.PlayChimeAndWait()

Plays the chime sound and waits for it to finish.

Sound.PlayChimes()

Plays the chimes sound.

Sound.PlayChimesAndWait()

Plays the chimes sound and waits for it to finish.

Sound.PlayClick()

Plays the click sound.

Sound.PlayClickAndWait()

Plays the click sound and waits for it to finish.

Sound.PlayDing()

Plays the ding sound.

Sound.PlayDingAndWait()

Plays the ding sound and waits for it to finish.

Sound.PlayMusic(notes)

Plays musical notes.

Parameter:

- notes:**

A set of musical notes to play. The format is a subset of the [Music Macro Language supported by QBasic](#). You can send these notes in a string separated by spaces, or as items of a string array.

Examples:

You can send tones as one string separated by spaces:

```
SKimigayo = "04 D2C2D2E2 G2E2D1 E2G2A2G4A4"  
        + " 05D204B2A2G2 E2G2A1 05D2C2D1 04E2G2A2G2"  
        + " E2R4G4D1 A205C2D1 C2D204A2G2 A2G4E4D1"  
Sound.PlayMusic(SKimigayo)
```

Or you can send these tones as an array:

```
Sound.PlayMusic({  
    "04",  
    "D2C2D2E2",  
    "G2E2D1",  
    "E2G2A2G4A4",  
    "05D204B2A2G2",  
    "E2G2A1",  
    "05D2C2D1",  
    "04E2G2A2G2",  
    "E2R4G4D1",  
    "A205C2D1",  
    "C2D204A2G2",  
    "A2G4E4D1"  
})
```

Sound.Resume(filePath)

Resumes playing the given audio from the position where the playback was paused.

If the file hasn't not played yet, this operation will play it from start.

Parameter:

- **filePath:**

The path for the audio file like "c:\music\track1.mp3". You can omit this argument when you call this method from a sound variable.

Example:

```
Track1 = Sound.Load("c:\music\track1.mp3")
Track1.Play()
Program.Delay(10)
Track1.Stop()
Program.Delay(10)
Track1.Resume()
```

Sound.Stop(filePath)

Stops playback of an audio file. If the file was not already playing, this operation will not do anything.

Parameter:

- **filePath:**

The path for the audio file like "c:\music\track1.mp3". You can omit this argument when you call this method from a sound variable.

Example:

```
Track1 = Sound.Load("c:\music\track1.mp3")
Track1.Play()
Program.Delay(10)
Track1.Stop()
```

The Stack Type

This object provides a way of storing values just like stacking up plates. You can push a value to the top of the stack and pop it off. You can only pop values one by one off the stack and the last pushed value will be the first one to pop out: Last Input First Out (LIFO).

The Stack type has these members:

Stack.GetCount(stackName) As Double

Gets the count of items in the specified stack.

Parameter:

- **stackName:**

The name of the stack.

Returns:

The number of items in the specified stack.

Stack.PopValue(stackName)

Pops the value from the specified stack.

Parameter:

- **stackName:**

The name of the stack. This name is case-insensitive, which means that "Users" and "users" will refer to the same stack name.

Returns:

The value from the stack. If the stack is empty or there is no stack with the given name, this method will return an empty string "".

Example:

```
Stack.PushValue("Numbers", 1)
Stack.PushValue("numbers", 2)
Stack.PushValue("numbers", 3)
TextWindow.WriteLine(Stack.PopValue("numbers")) ' 3
Stack.PushValue("Numbers", 4)
TextWindow.WriteLine(
    Stack.PopValue("Numbers"), ' 4
    Stack.PopValue("numbers"), ' 2
    Stack.PopValue("numbers") ' 1
)
```

⚙️ Stack.PushValue(stackName, value)

Pushes a value to the specified stack.

Parameters:

- **stackName:**

The name of the stack. If the stack name doesn't exist, it will be created.

- **value:**

The value to push.

Example:

One famous application of the stack is reversing the array:

```
Names = {"Ali", "Hala", "Yaser", "Mohammed"}
```

```
ForEach Name In Names
    Stack.PushValue("Names", Name)
Next

For I = 1 To Stack.GetCount("names")
    Names[I] = Stack.PopValue("names")
Next

TextWindow.WriteLine(Names)
```



Mohammmad
Yaser
Hala
Ali

The Text Type

The Text object provides helpful operations for working with Text.

The Text type has these members:

Text.Append(text1, text2) As String

Appends two text inputs.

You can use the + operator to do the same job, but this method insures the two texts are always joined even they are two numbers, while the + operator will arithmetically add them whenever it is possible.

Parameters:

- **text1:**

The first part of the text to be appended. This argument can be omitted if you call this method as an extension method of a string variable.

- **text2:**

The second part of the text to be appended. You can send an array to append all its items.

Returns:

The appended text containing both the specified parts.

Examples:

X = ". "

Y = "test"

```
TextWindow.WriteLine({
    Y.Append(X),                                ' test.
    Text.Append(Y, X),                            ' test.
    "." + "1" + 4,                             ' 4.1
    Text.Append(".", {"1", "4"}),                ' .14
    X.Append({"1", "4"})                         ' .14
})
```

⌚Text.Contains(text, subText) As Boolean

Gets whether or not a given text contains the specified subText.

Note that the comparison is case-sensitive.

Parameters:

- **text:**

The larger text to search within. This argument can be omitted if you call this method as an extension method of a string variable.

- **subText:**

The sub-text to search for.

Returns:

True if the subtext was found at any position in the given text.

Examples:

```
sVB = "Small Visual Basic"
TextWindow.WriteLine({
    sVB.Contains("Visual"),                      ' True
    sVB.Contains("visual"),                     ' False
    Text.Contains(sVB.LowerCase, "visual"),      ' True
    sVB.Contains("")                           ' False
})
```

Text.ConvertToLowercase(text) As String

Converts the given text to lower case.

Note that this method is kept for backward compatibility with Small Basic, but it is recommended to use the [ToLower method](#) instead, not only because it is much shorter, but also because it is used in VB.NET.

Parameter:

- **text:**

The text to convert to lower case. This argument can be omitted if you call this method as an extension property (with the name LowerCase) of a string variable.

Returns:

The lower case version of the given text.

Text.ConvertToUpperCase(text) As String

Converts the given text to upper case.

Note that this method is kept for backward compatibility with Small Basic, but it is recommended to use the [ToUpper method](#) instead, not only because it is much shorter, but also because it is used in VB.NET.

Parameter:

- **text:**

The text to convert to upper case. This argument can be omitted if you call this method as an extension property (with the name UpperCase) of a string variable.

Returns:

The upper case version of the given text.

① **Text.EndsWith(text, subText) As Boolean**

Gets whether or not a given text ends with the specified subText. The comparison is case-sensitive.

Parameters:

- **text:**

The larger text to search within. This argument can be omitted if you call this method as an extension method of a string variable.

- **subText:**

The sub-text to search for.

Returns:

True if the subtext was found at the end of the given text.

Examples:

```
sVB = "Small Visual Basic"  
TextWindow.WriteLine({  
    sVB.Contains("Basic"),           ' True  
    sVB.Contains("basic"),          ' False  
    Text.Contains(sVB.LowerCase, "basic"), ' True  
    sVB.Contains("")               ' False  
})
```

Text.Format(text, values) As String

Formats the string by replacing the [1], [2], ... [n] placeholders by the corresponding items from the given values array.

Parameters:

- **text:**

The string to Format. Use [1], [2],... [n] placeholders in the string, to refer the values[1], values[2], ... values[n].

This argument can be omitted if you call this method as an extension method of a string variable.

- **values:**

An array to use its elements to replace the [1], [2],... [n] placeholders if found in the text.

Returns:

The formatted string after substituting the [1], [2],... [n] placeholders with elements from the values array.

Example:

```
Student = "[1]. Name: [2], Age: [3]."
TextWindow.WriteLine({
    Text.Format(Student, {1, "Adam", 15}),
    Student.Format({2, "Mona", 10}),
    Student.Format({3, "Yasser", 12})
})
```

```
1. Name: Adam, Age: 15.
2. Name: Mona, Age: 10.
3. Name: Yasser, Age: 12.
```

Text.GetCharacter(characterCode) As String

Gets the corresponding character for the given Unicode character code, which can then be used with regular text.

Parameter:

- **characterCode:**

The character code (Unicode based) for the required character.

Returns:

A Unicode character that corresponds to the code specified.

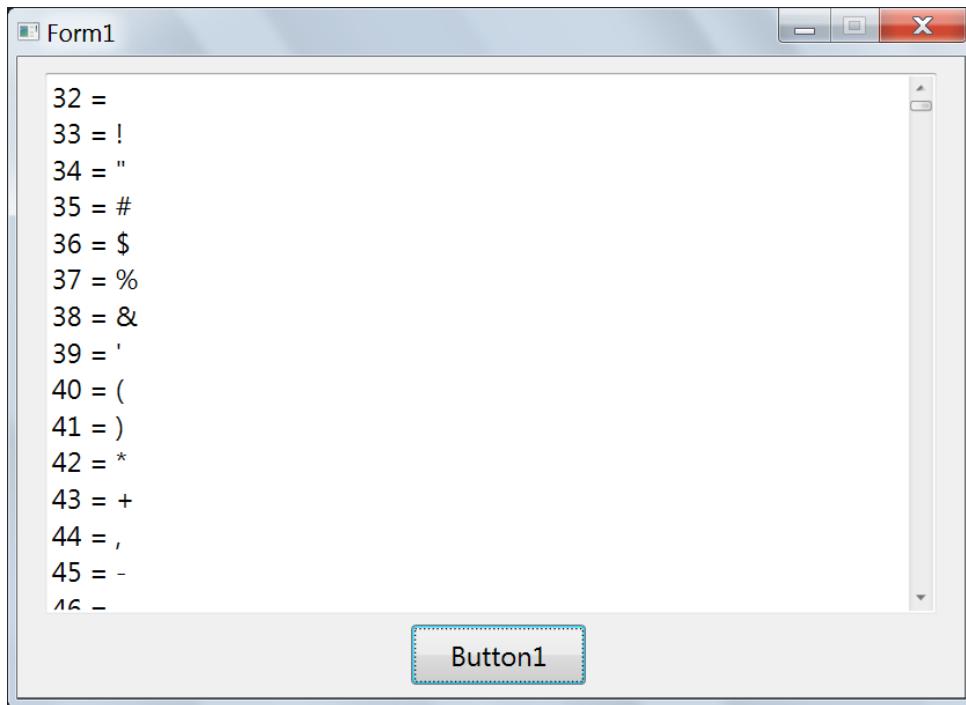
Examples:

Add a textbox and a button on a form, double-click the button and write this code in the Button1_OnClick event handler:

```
For code = 32 To 1000
    c = Text.GetCharacter(code)
    TextBox1.Append(code)
    TextBox1.Append(" = ")
    TextBox1.AppendLine(c)
```

[Next](#)

Run the program and click the button. This will print 1000 Unicode character in the textbox, except the first 31 characters, because most of them are control characters that can't be printed. The code 32 represents the space character, and you can see what next codes represent!



You can print many more characters, but take care that appending thousands of lines to the textbox will be slow, and useless as you can't read thousands of lines. So, I recommend you to view ranges of characters, where each range contains 100 characters only. You can easily add a textbox to enter the range start in, and modify the above code to make the for loop start from the range start and end at range start + 99.

💡 **Text.GetCharacterAt(text, pos) As String**

Gets the character existing in the given position in the text.

Note that you can directly use the indexer `text[pos]` to get the character.

Parameters:

- **text:**

The input text. This argument can be omitted if you call this method as an extension method of a string variable.

- **pos:**

The position of the character.

Returns:

The character existed in the given position, or an empty string "" if the position is invalid.

Examples:

In the following code, the -1, 0, and 5 positions are invalid, so, the GetCharacterAt method will return an empty string "" (and nothing will be written on the text window in the corresponding lines), while the 1 and 4 positions are valid, so, the GetCharacterAt method will return T and t. We also used the indexer to get the e and s at the 2 and 3 positions:

```
X = "Test"  
TextWindow.WriteLine({  
    Text.GetCharacterAt(X, -1),  
    X.GetCharacterAt(0),  
    X.GetCharacterAt(1),      ' T  
    X[2],                  ' e  
    X[3],                  ' s  
    X.GetCharacterAt(4),      ' t  
    X.GetCharacterAt(5)  
})
```

⌚Text.GetCharacterCode(character) As Double

Gets the corresponding character code for the given Unicode character.

Parameter:

- **character:**

The character whose code is requested.

Returns:

A Unicode based code that corresponds to the character specified.

Example:

The Unicode 100000 is represented with a pair of surrogates. We don't need to be aware of that when we use the GetCharacter and GetCharacterCode methods to do the conversions for us. The following examples shows that converting the Unicode 100000 to a character will return a 2 length string, and you can send this string back to the GetCharacterCode method to get the Unicode 100000 again!

```
x = Text.GetCharacter(100000)
TextWindow.WriteLine(x.Length) '2
c = Text.GetCharacterCode(x)
TextWindow.WriteLine(c) ' 100000
```

① **Text.IndexOf(text, subText, start, isBackward) As Double**

Finds the position where a sub-text appears in the specified text. The search is case-sensitive.

Parameters:

- **text:**

The text to search in. This argument can be omitted if you call this method as an extension method of a string variable.

- **subText:**

The text to search for.

- **start:**

The text position to start searching from.

- **isBackward:**

Use True if you want to search from start back to the first position in the text (= 1), or False if you want to go forward to the end of the text.

Returns:

The position at which the sub-text appears in the specified text, or 0 if the text doesn't exist.

Example:

```
SVB = "Small Visual Basic"
TextWindow.WriteLine({
    SVB.IndexOf("Visual", 1, False),      ' 7
    SVB.IndexOf("visual", 1, False),       ' 0
    Text.IndexOf(
        SVB.LowerCase, "visual", 1, False), ' 7
    SVB.IndexOf("s", 2, False),           ' 9
    SVB.IndexOf("s", SVB.Length, True)    ' 16
})
```

⌚Text.GetLength(text) As Double

Gets the count of the characters in the given text.

Parameter:

- **text:**

The text whose length is needed. This argument can be omitted if you call this method as an extension property (with the name Length) of a string variable.

Returns:

The length of the given text.

Examples:

```
X = "Test"
TextWindow.WriteLine({
    Text.GetLength(""),           ' 0
    Text.GetLength(X),           ' 4
    X.Length                    ' 4
})
' Print all text characters
For I = 1 To X.Length
    TextWindow.WriteLine(X[I])
Next

' Or just use ForEach:
ForEach C In X
    TextWindow.WriteLine(C)
Next
```

Text.GetTextSubText(text, start, length) As String

Gets a sub-text from the given text.

Parameters:

- **text:**

The text to derive the sub-text from. This argument can be omitted if you call this method as an extension method (with the name SubText) of a string variable.

- **start:**

The start position of the subtext.

- If the start is less than 1, the subtext will start from the first character of the text.
- If the start is greater than the length of the text, this method will return an empty string "".

• **length:**

Specifies the length of the sub text.

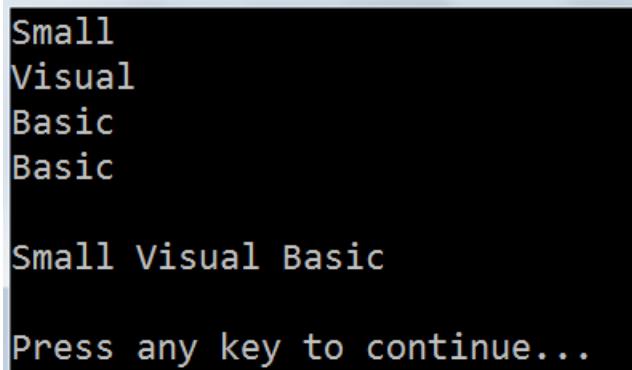
- If the length is less than 1, this method will return an empty string "".
- If the substring length exceeds the text length, the extra length will be ignored and the substring will stop at the end of the text.

Returns:

The requested sub-text.

Examples:

```
SVB = "Small Visual Basic"
TextWindow.WriteLine({
    Text.GetSubText(SVB, 1, 5),      ' Small
    SVB.SubText(7, 6),              ' Visual
    SVB.SubText(14, 5),             ' Basic
    SVB.SubText(14, 100),            ' Basic
    SVB.SubText(14, 0),
    SVB.SubText(-1, SVB.Length),   ' Small Visual Basic
    SVB.SubText(100, 5)
})
```



Text.GetTextSubTextToEnd(text, start) As String

Gets a sub-text from the given text from a specified position to the end.

Parameters:

- **text:**

The text to derive the sub-text from. This argument can be omitted if you call this method as an extension method (with the name SubTextToEnd) of a string variable.

- **start:**

The start position of the subtext.

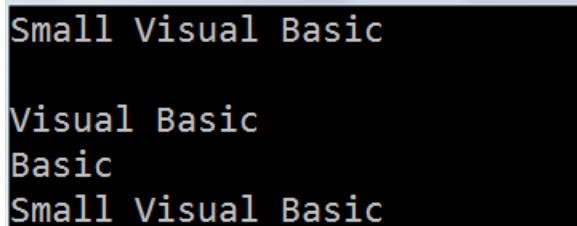
- If the start is less than 1, the subtext will start from the first character of the text.
- If the start is greater than the length of the text, this method will return an empty string "".

Returns:

The requested sub-text.

Examples:

```
SVB = "Small Visual Basic"
TextWindow.WriteLine({
    Text.GetTextSubTextToEnd(SVB, 1),
    SVB.SubTextToEnd(100),
    SVB.SubTextToEnd(7),
    SVB.SubTextToEnd(14),
    SVB.SubTextToEnd(-1)
})
```



```
Small Visual Basic
Visual Basic
Basic
Small Visual Basic
Small Visual Basic
```

Text.IsEmpty(text) As Boolean

Returns True if the given text is empty, or False otherwise.

Note that you can directly use the = operator to compare the text with an empty string "".

Parameter:

- **text:**

The input text. This argument can be omitted if you call this method as an extension property of a string variable.

Returns:

True if the text is empty, or False otherwise.

Text.IsNumeric(text) As Boolean

Checks if the string contains a numeric value.

Parameter:

- **text:**

The string to check its value. This argument can be omitted if you call this method as an extension property of a string variable.

Returns:

True if text is a number, or False otherwise.

Examples:

X = "1.4"

Y = "100\$"

```
TextWindow.WriteLine({
    X.IsNumeric,           ' True
    Y.IsNumeric,           ' False
    Text.IsNumeric(""),    ' False
    Text.IsNumeric(-0.6),  ' True
    Text.IsNumeric("One")  ' False
})
```

⌚Text.IsSubText(text, subText) As Boolean

Gets whether or not a given subText is a subset of the larger text. It is similar to the [Text.Contains](#) Method.

Parameters:

- **text:**

The larger text within which the sub-text will be searched.

- **subText:**

The sub-text to search for.

Returns:

True if the subtext was found within the given text, or False otherwise.

Examples:

```
SVB = "Small Visual Basic"
TextWindow.WriteLine({
    Text.IsSubText(SVB, "Visual"),           ' True
    Text.IsSubText(SVB, "visual"),            ' False
    Text.IsSubText(SVB.LowerCase, "visual"),  ' True
    Text.IsSubText(SVB, "")                  ' False
})
```

Text.LowerCase As String

This is an extension property of string variables. It returns a new text with all its characters are the lower case form of the current text. You can use also use the [Text.ToLower](#) method to do the same job.

Text.Length As String

This is an extension property of string variables. It returns the length of the current text. You can use also use the [Text.GetLength](#) method to do the same job.

Text.NewLine As String

Returns the new line characters (Chars.Cr + Chars.Lf) so you can add them to the text to continue writing in a new line.

This property is similar to the Chars.CrLf property.

Text.SetCharacterAt(text, pos, newText) As String

Changes the character existing at the given position to the given new text.

Note that you can directly use the indexer to set the new text:

text[pos] = newText

but in this case the input text will be affected with the change directly. In fact the above statement is equivalent to:

text = Text.SetCharacterAt(text, pos, newText)

Parameters:

- **text:**

The input text. This argument can be omitted if you call this method as an extension method of a string variable.

• pos:

The position of the character:

- Use 0 to insert the new text before the first character.
- Use a negative number to insert the new text before the first character with more spaces added between them.
- Use a number greater than the length of the text to append the new text after the last character, with extra spaces added between them to reach the given position.

• newText:

The new text to set at the given position.

Send an empty string "" to remove the current character from the text, or send one or more characters to replace it.

Returns:

A new text with the char changed to the given value. The input text will not change.

Examples:

```
X = "test"
TextWindow.WriteLine({
    Text.SetCharacterAt(X, -1, "a"),      'a test
    X.SetCharacterAt(0, "a"),              'atest
    Text.SetCharacterAt(X, 1, "a"),       'aest
    X.SetCharacterAt(2, "..e"),          't..est
    X.SetCharacterAt(5, "s"),            'tests
    X.SetCharacterAt(6, "#1")           'test #1
})
```

' Or use the indexer to do the same:

```
X = "test"
X[-1] = "a"
```

```
TextWindow.WriteLine(X) 'a test
X = "test"
X[0] = "a"
TextWindow.WriteLine(X) 'atest

X = "test"
X[1] = "a"
TextWindow.WriteLine(X) 'aest

X = "test"
X[2] = "..e"
TextWindow.WriteLine(X) 't..est

X = "test"
X[5] = "s"
TextWindow.WriteLine(X) 'tests

X = "test"
X[6] = "#1"
TextWindow.WriteLine(X) 'test #1
```

⚙️ **Text.Split(text, separator, trim, removeEmpty) As Array**

Splits the given text at the given separator.

Parameters:

- **text:**

The input text. This argument can be omitted if you call this method as an extension method of a string variable.

- **separator:**

One or more characters to split the text at. The separator will not appear in the result.

You can also send an array to use its items as separators.

- **trim:**

Use True to trim white spaces (spaces, tabs and line characters) from the start and end of the separated strings.

- **removeEmpty:**

Use True to remove empty items from the result array. Empty items can appear if two successive separators exist in the text without anything in between, or because the item contains white spaces and you choose to trim them.

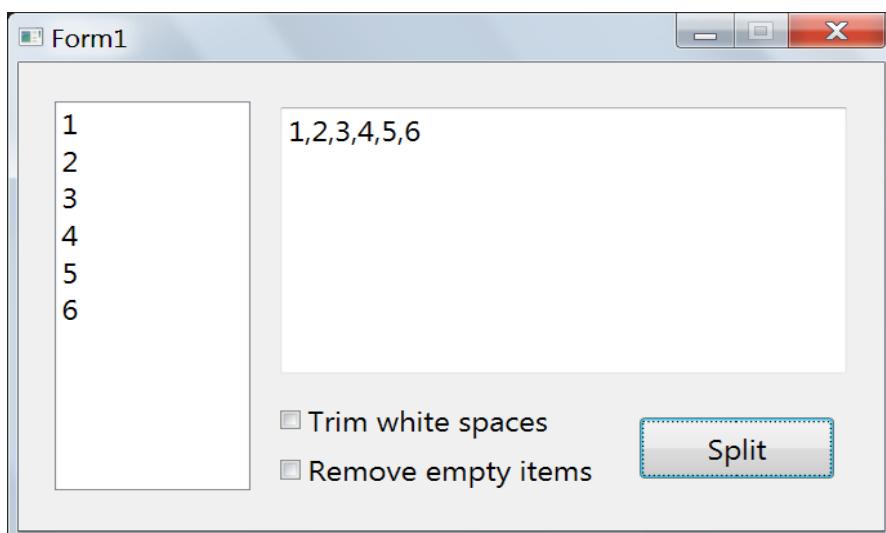
Returns:

An array containing the resulting items.

Example:

You can see this method in action in the "Split text" project in the samples folder. The form allows the user to write items in the textbox separated by commas or new lines, then click the split button to split the items and add them to the list box.

The form also shows two check boxes to allow the user to choose whether or not to trim the white spaces and remove empty items. The following picture shows you this GUI:



The split button click handler simply calls the Text.Split method to get the results array, and adds it to the list using the AddItem Method:

```
Sub Button1_OnClick()
    ListBox1.RemoveAllItems()
    ListBox1.AddItem(
        Text.Split(
            TextBox1.Text,
            {",", CharsCrLf},
            ChkTrim.Checked,
            ChkNoEmpty.Checked
        )
    )
EndSub
```

⌚Text.StartsWith(text, subText) As Boolean

Gets whether or not a given text starts with the specified subText. The comparison is case-sensitive.

Parameters:

- **text:**

The larger text to search within. This argument can be omitted if you call this method as an extension method of a string variable.

- **subText:**

The sub-text to search for.

Returns:

True if the subtext was found at the start of the given text.

Examples:

```
sVB = "Small Visual Basic"
TextWindow.WriteLine({
    sVB.StartsWith("Small"),           ' True
    sVB.StartsWith("small"),          ' False
    Text.StartsWith(sVB.LowerCase, "small"), ' True
    sVB.StartsWith("")                ' False
})
```

Text.ToLower(text) As String

Converts all characters of the input text to lower case.

Parameter:

- **text:**

The input text. This argument can be omitted if you call this method as an extension property (with the name LowerCase) of a string variable.

Returns:

A lower-case text.

Examples:

```
X = "aBC"
TextWindow.WriteLine({
    X.LowerCase,           ' abc
    Text.ToLower(X)       ' abc
})
```

Text.ToNumber(text) As Double

Converts the input text to a number. This method is used by the compiler to insure that the For loop variables (counter, start, end and step) are always valid numbers. You don't need to use this method yourself, because it converts single characters to their Unicode code. Instead, you should use the [Text.IsNumeric](#) method to check if the variable contains a valid number before using it.

Parameter:

- **text:**

The input text. This argument can be omitted if you call this method as an extension method of a string variable.

Returns:

- If the text is numeric, returns its numeric value.
- If text contains only one non numeric character, returns its ASCII code.
- Otherwise, returns 0.

Examples:

The next for loop starts from "a" to "e". sVB compiler will call the Text.ToNumber("a") and Text.ToNumber("e") to convert these two strings to numbers, which will get the Unicode of the two letters a and e. So, we can print the characters from a to e just by getting back the character that the counter I represents:

```
For I = "a" To "e"  
    TextWindow.WriteLine(Text.GetCharacter(I))  
Next
```

⌚Text.ToString() As String

Converts the given value or array to a string.

Parameter:

- **value:**

The input value. This argument can be omitted if you call this method as an extension method of a string variable.

Returns:

The string representation of the input value. For example, the array string can be of the form {1, 2, 3}.

Examples:

```
X = {"a", "b", "c"}  
TextWindow.WriteLine({  
    X,  
    Text.ToString(X)  
})  
  
' 1=a;2=b;3=c;  
' {a, b, c}
```

⌚Text.ToUpper(text) As String

Converts all characters of the input text to upper case.

Parameter:

- **text:**

The input text. This argument can be omitted if you call this method as an extension property (with the name Uppercase) of a string variable.

Returns:

An upper-case text.

Examples:

```
X = "aBc"  
TextWindow.WriteLine({  
    X.UpperCase,           ' ABC  
    Text.ToUpper(X)       ' ABC  
})
```

① **Text.Trim(text) As String**

Removes all leading and trailing white-space characters from the given text. White-space chars include spaces, tabs, and line symbols.

Parameter:

- **text:**

The input text. This argument can be omitted if you call this method as an extension method of a string variable.

Returns:

The trimmed string.

Examples:

```
X = " A "  
TextWindow.WriteLine({  
    "|" + X + "|",          ' | A |  
    "|" + X.Trim() + "|",   ' |A|  
    "|" + Text.Trim(X) + "|' |A|  
})
```



Text.ToUpper As String

This is an extension property of string variables. It returns a new text with all its characters are the upper case form of the current text. You can use also use the [Text.ToUpper](#) method to do the same job.

The TextBox Type

Represents the TextBox control, which allows the user to input text.

- Use the Text property to read the text the user wrote.
- Use the OnTextInput event to intercept the text before it is written to the textbox.
- Use the OnTextChanged event to take action after the text of the textbox changes.
- You can use the form designer to add a textbox to the form by dragging it from the toolbox.
- It is also possible to use the [Form.AddTextBox](#) method to create a new textbox and add it to the form at runtime.

Note that the TextBox control inherits all the properties, methods and events of the [Control](#) type.

Besides, the TextBox control has some new members that are listed below:

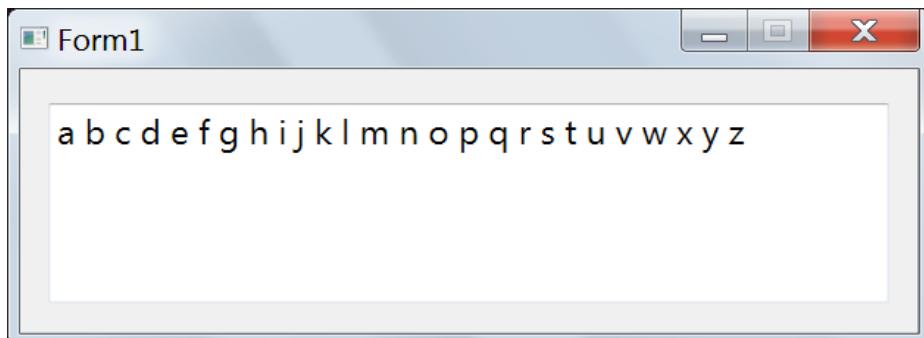
TextBox.Append(text)

Adds the given text at the end of the TextBox.

Parameter:

- **text:**

The text that will be written at the end of the TextBox.



⚙️ TextBox.AppendLine(lineText)

Adds the given text at the end of the TextBox then adds a new line character, so the next text will be written in a new line.

Parameter:

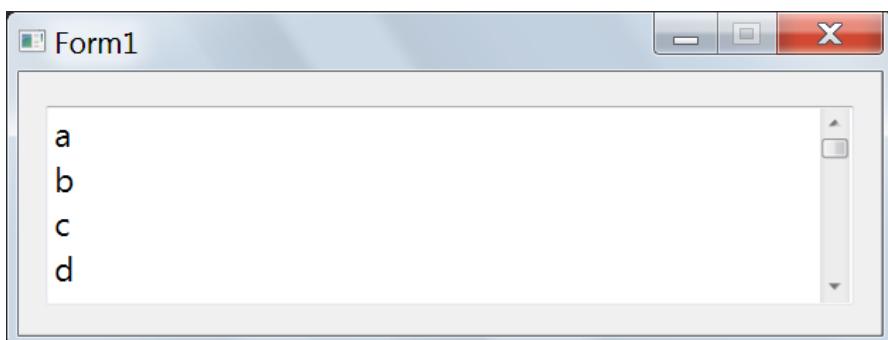
- **lineText:**

The text that will be written at the end of the TextBox followed by a new line character.

Example:

Add a textbox on the form, and write this code:

```
For I = "a" To "z"  
    TextBox1.AppendLine(Chars.GetCharacter(I))  
Next
```



⚙️ TextBox.AppendLines(lines)

Appends the items of the given array as lines, just after the last character of the textbox.

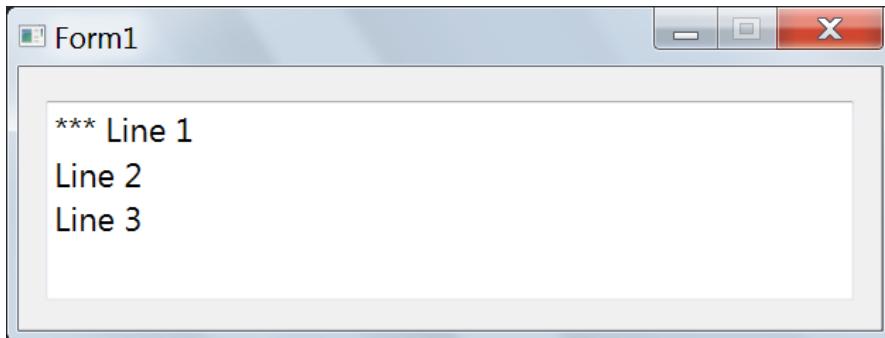
Parameter:

- **lineText:**

An array containing the lines to add to the textbox.

Add a textbox on the form, and write this code:

```
TextBox1.Text = "*** "
TextBox1.AppendLines({
    "Line 1",
    "Line 2",
    "Line 3"
})
```



🎨 TextBox.CanRedo As Boolean

Returns True if you can redo the last action on the TextBox.

Example:

You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the MenuEdit_OnOpen event handler in the frmMain.sb file to enable or disable the Redo command when the user opens the Edit menu:

```
MenuRedo.Enabled = TxtEditor.CanRedo
```

🎨 TextBox.CanUndo As Boolean

Returns True if you can undo the last action on the TextBox.

Example:

You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the MenuEdit_OnOpen event handler in the frmMain.sb file to

enable or disable the Undo command when the user opens the Edit menu:

```
MenuUndo.Enabled = TxtEditor.CanUndo
```

TextBox.CaretIndex As Double

Gets or sets the current caret position in the TextBox.

Note that:

- CaretIndex = 1: sets the caret before the first character in the textbox. Any smaller value will have the same effect.
- CaretIndex = 2: sets the caret after the first character in the textbox.
- CaretIndex = text length: sets the caret before the last character in the textbox.
- CaretIndex = text length + 1: sets the caret after the last character in the textbox, and any bigger value will have the same effect.

Example:

You can set the caret at the start of the textbox like this:

```
TextBox1.CaretIndex = 1
```

You can set the caret at the end of the textbox like this:

```
TextBox1.CaretIndex = TextBox1.Length + 1
```

TextBox.Copy()

Call this method to copy the selected text from the TextBox to the clipboard.

Example:

You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the

MenuCopy_OnClick event handler in the frmMain.sb file to execute the Copy command from Edit menu:

```
Sub MenuCopy_OnClick()
    TxtEditor.Copy()
EndSub
```

TextBox.Cut()

Call this method to cut the selected text from the TextBox and add it to the clipboard.

Example:

You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the MenuCut_OnClick event handler in the frmMain.sb file to execute the Cut command from Edit menu:

```
Sub MenuCut_OnClick()
    TxtEditor.Cut()
EndSub
```

TextBox.Length As Double

Gets the length of the text written in the TextBox.

Example:

Add a textbox on the form, double-click it to go to its OnTextChanged event handler, and write this code:

```
Sub TextBox1_OnTextChanged()
    Me.Text = TextBox1.Length
EndSub
```

Press F5 to run the project, and write some text in the textbox. The form title bar will show the number of

characters you wrote. This number will be changed every time you write or delete characters.

TextBox.Multiline As Boolean

Set this property to True to allow the user to write more than one line in the TextBox.

Examples:

You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the MenuMuliline_OnCheck event handler in the frmMain.sb file to update the multiline state of the textbox when the user checks or unchecks the Muliline option in the View menu:

```
Sub MenuMuliline_OnCheck()
    TxtEditor.Multiline = MenuMuliline.Checked
EndSub
```

It is also used in the MenuView_OnOpen event handler to check or uncheck the Multiline option according to the multiline state of the textbox when the user opens the View menu:

```
MenuMuliline.Checked = TxtEditor.Multiline
```

TextBox.OnSelection

This event is fired when the selected text in the textbox is changed.

For more info, read about [handling control events](#).

TextBox.OnTextChanged

This event is fired when the text is changed.

For more info, read about [handling control events](#).

Examples:

You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the frmMain.sb file to update the set the IsModified flag to true when the user makes any changes to the opened document. The IsModified flag is used to notify the user to save the changes before closing the document:

```
Sub TxtEditor_OnTextChanged()  
    IsModified = True  
EndSub
```

TextBox.OnTextInput

This event is fired just before text is written to the TextBox.

Use the [Event.LastTextInput](#) property to get this text.

Set the [Event.Handled](#) property to True if you want to cancel writing this text to the Textbox.

For more info, read about [handling control events](#).

Example:

See the examples section of the [Event.Handled property](#).

TextBox.Paste()

Call this method to paste the text from the clipboard to the current caret pos in the textbox.

Example:

You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the MenuPaste_OnClick event handler in the frmMain.sb file to execute the Paste command from Edit menu:

```
Sub MenuPaste_OnClick()
    TxtEditor.Paste()
EndSub
```

TextBox.Redo()

Call this method to redo the last action on the TextBox.

Example:

You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the MenuRedo_OnClick event handler in the frmMain.sb file to execute the Redo command from Edit menu:

```
Sub MenuCut_OnClick()
    TxtEditor.Redo()
EndSub
```

TextBox.Select(startPos, length)

Selects a part of the text displayed in the textbox.

Parameters:

- **startPos:**

The pos of the first character you want to select.

- **length:**

The number of characters you want to select. You can use 0 to just move the caret to the start position without selecting any characters.

Example:

You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the Find function in the FrmFind.sb file to select the search result found in the textbox:

```
EditorTextBox.Select(pos, _find.Length)
```

TextBox.SelectAll()

Selects all the text displayed in the textbox.

Example:

You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the MenuSelAll_OnClick event handler in the frmMain.sb file to execute the Select All command from Edit menu:

```
Sub MenuSelAll_OnClick()
    TxtEditor.SelectAll()
EndSub
```

TextBox.SelectedText As String

Gets or sets the selected text in the textbox. If you set this property to an empty string, this will delete the text currently selected in the textbox (if any).

Example:

You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the Replace function in the FrmFind.sb file to replace the search result found in the textbox with the replacement text:

```
EditorTextBox.SelectedText = TxtReplace.Text
```

TextBox.SelectionLength As String

Gets or sets the length of the selected text.

TextBox.SelectionStart As String

Gets or sets the start pos of the selected text.

TextBox.Text As String

Gets or sets the text that is displayed by the textbox.

TextBox.Underlined As Boolean

Gets or sets whether or not to draw a line under the text.

Examples:

You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the MenuUnderline_OnCheck event handler in the frmMain.sb file to update the underlined state of the textbox when the user checks or unchecks the Underlined option in the View menu:

```
Sub MenuUnderline_OnCheck()
    TxtEditor.Underlined = MenuUnderline.Checked
EndSub
```

It is also used in the MenuView_OnOpen event handler to check or uncheck the Underlined option according to the underlined state of the textbox when the user opens the View menu:

```
MenuUnderline.Checked = TxtEditor.Underlined
```

TextBox.Undo()

Call this method to undo the last action on the textbox.

Example:

You can see this property in action in the "sVB Notepad" project in the samples folder. It is used in the MenuUndo_OnClick event handler in the frmMain.sb file to execute the Undo command from Edit menu:

```
Sub MenuCut_OnClick()
    TxtEditor.Undo()
EndSub
```

TextBox.WrapWord As Boolean

Gets or sets whether or not to wrap words to the next line when they exceed the textbox width.

The default value is False, so the horizontal scroll bar will appear when any line exceeds the textbox width.

If you set this property to true, the horizontal scroll bar will never appear, and the text will be wrapped over next lines.

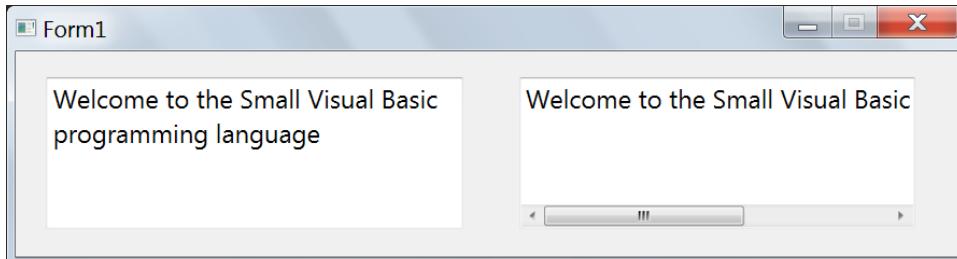
Example:

Add two textboxes on the form, and set their Text property to: "Welcome to the Small Visual Basic programming language".

Use this code to enable word wrapping in the first textbox:

```
TextBox1.WrapWord = True
```

When you run the project, this is what you will see:



The TextWindow Type

Provides text-related input and output functionalities, like writing and reading texts and numbers to and from the text-based text window.

Note that sVB (starting from v2.8.6.1) allows you to use **TW** as a shortcut for the **TextWindow** type. For example, the following statement:

`Tw.WriteLine("Hello")`

Is a valid shortcut for this statement:

`TextWindow.WriteLine("Hello")`

The TextWindow type has these members:

TextWindow.BackgroundColor As Color

Gets or sets the background color of the text to be output in the text window.

TextWindow.Clear()

Clears the TextWindow.

TextWindow.CursorLeft As Double

Gets or sets the cursor's column position on the text window.

TextWindow.CursorTop As Double

Gets or sets the cursor's row position on the text window.

TextWindow.ForegroundColor As Color

Gets or sets the foreground color of the text to be output in the text window.

TextWindow.Hide()

Hides the text window. Content will be preserved until the window is shown again.

TextWindow.Left As Double

Gets or sets the Left position of the Text Window.

TextWindow.Pause()

Waits for user input before returning.

TextWindow.PauseIfVisible()

Waits for user input only when the TextWindow is already open.

TextWindow.PauseThenClose()

Waits for user to press any key to close the TextWindow if it its open, otherwise it closes it directly.

TextWindow.PauseWithoutMessage()

Waits for user input before returning.

TextWindow.Read() As String

Reads a line of text from the text window. This function will not return until the user hits the ENTER key from keyboard.

Note: You can type ? instead of any string expression in your code line, and the code editor will replace it with this method when you leave the line. For example, the expression:

`x = ?`

will be converted to:

`x = TW.Read()`

Returns:

The text that was read from the text window.

TextWindow.ReadNumber() As Double

Reads a number from the text window. This function will not return until the user hits the ENTER key from keyboard.

Note: You can type #? instead of any string expression in your code line, and the code editor will replace it with this method when you leave the line. For example, the expression:

`x = #?`

will be converted to:

`x = TW.ReadNumber()`

Returns:

The number that was read from the text window.

TextWindow.Show()

Shows the text window to enable interactions with it.

Note that calling any method of the TextWindow object will show it.

TextWindow.Title As String

Gets or sets the title for the text window.

TextWindow.Top As Double

Gets or sets the top position of the text window.

TextWindow.Write(data)

Writes a text or a number to the text window.

Unlike the WriteLine method, this will not append a new line character, which means that anything written to the text window after this call appear on the same line.

Parameter:

- **data:**

The text or number to write to the text window.

Note:

You can type ?: at the beginning of your code line, and the code editor will replace it with this method when you leave the line. For example, the expression:

? : "Hello sVB!"

will be converted to:

TW.WriteLine("Hello sVB!")

TextWindow.WriteLine(data)

Writes a text or a number to the text window, with a new line character appended to the output, so that the next time something is written to the text window, it will go in a new line.

Parameter:

- **data:**

The text or number to write to the text window.

Note:

You can type ? at the beginning of your code line, and the code editor will replace it with this method when you leave the line. For example, the expression:

? "Hello sVB!"

will be converted to:

`TW.WriteLine("Hello sVB!")`

You can also use the ? and #? for the Read and ReadNumber methods in the same line. For example, the expression:

??

will be converted to:

`TW.WriteLine(TW.Read())`

And the expression:

?#?

will be converted to:

`TW.WriteLine(TW.ReadNumber())`

Which allows you to compose this interesting expression:

?#?%#?

which will be converted to:

`TW.WriteLine(TW.ReadNumber() Mod TW.ReadNumber())`

Now paste this program in the sVB code editor and see what you get:

```
? : "Enter the first number: "
n1 = #?
?: "Enter the second number: "
n2 = #?
?: "n1/n2 = " & Math.Floor(n1/n2)
? ", remainder = " & n1%n2
```

TextWindow.WriteLine(lines)

Writes the items of the given array to the TextWindow, and appends a new line after each of them.

Parameter:

- **lines:**

An array of text lines.

Note:

You can type ? at the beginning of your code line followed by an array initializer or a direct command separated values, and the code editor will replace it with this method when you leave the line. For example, the expression:

```
? "Hello from sVB!", "What's your name?"
```

will be converted to:

```
TW.WriteLine({"Hello sVB!", "What's your name?"})
```

The Thread Type

The Thread object allows you to run a subroutine in a new thread. It has only one member which is the SubToRun event.

Note that you should avoid using any global variables in the code that you run in separate threads, because this can make multiple threads race to read and change those global variables, which can give you wrong results.

Instead, you should copy global variables into local variables at the start of the thread code, and avoid changing global variables from threads. This makes threads useful in scenarios like drawing on the graphics window, or reading and writing many separate files, so that each thread can be totally independent on the others.

See the example provided by the SubToRun event.

The thread typee has these two members:

Thread.InitializationDelay As Double

Gets or sets the time in milliseconds that the main thread will be paused for, after the new thread starts to allow it to read the global variables.

The default value is 10, but you can increase it if the thread needs to read many global variables.

Use 0 if you don't need any delay.

Thread.SubToRun

This is not actually an event, but it is more like a property that allows you to set its value to the subroutine you need to run in a new thread. When you do that, this subroutine will be called immediately in a new thread to allow you to execute a task in parallel to your normal code.

You can set this handler as many times as you need to create many threads, but not too many (at most 100 threads) because that can make your system hang or block some of these threads. Note also that you can't pass arguments to the handler, so you will need to use global variables for that, so this method will pause your code for the period defined by the InitializationDelay property, to allow the new thread to start and read the global variables, but if this is not enough for your code, you may need to call the Program.Delay again to increase the delay.

Warning:

Threads will not be created in debug mode to allow you to trace the code. In this case your program may be slower and some of its logic may not function correctly.

Example:

The "Mandelbrot Set" project in the samples folder uses multi-threading to draw the pixels of the Mandelbrot set. This makes the drawing process faster by about 35%.

To do that, the height of the GW is divided into intervals, so that each thread draws the pixels of its own interval.

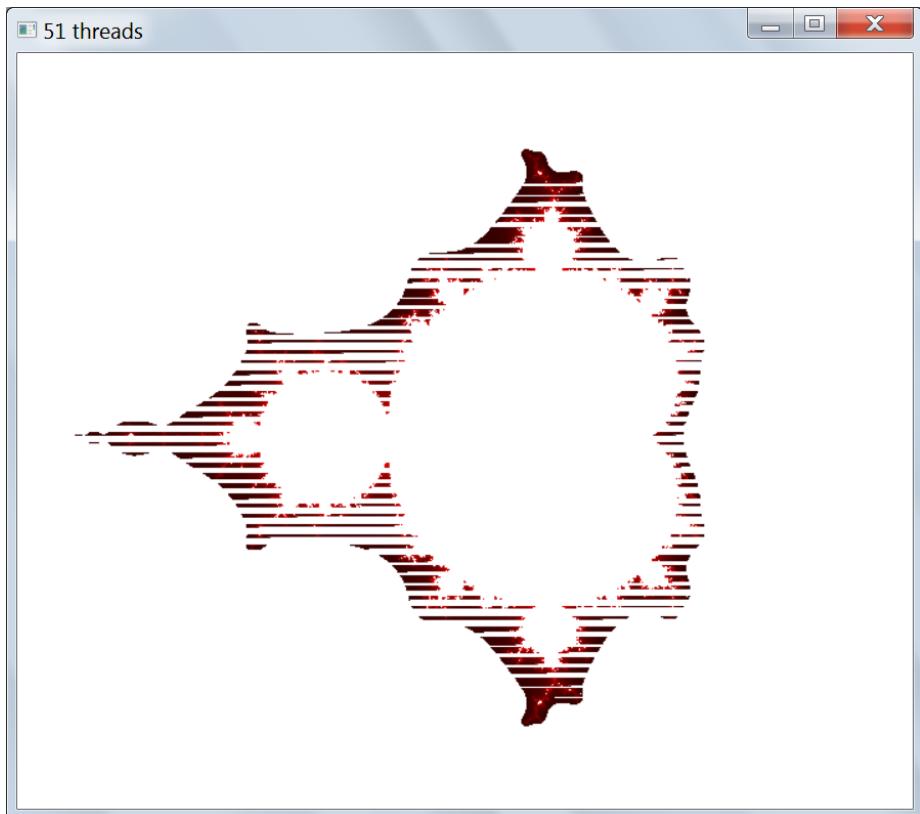
You can change the value of the Interval variable, but note that smaller the value, the larger the number of the threads that will be created.

This is the part of the Draw subroutine, that starts a thread of each interval:

```
For y = 0 To GW.Height Step Interval + 1
    CurY = y
    Thread.SubToRun = DrawPixels
    Next
```

This will launch the DrawPixels subroutine many times to run simultaneously in parallel threads. Each version of the DrawPixels subroutine captures the values of CurY that is set before calling it to use to calculate the start and end of its own drawing interval:

```
Sub DrawPixels()
    start = CurY
    end = start + Interval
    For y = start To end
        ' Draw pixels
    Next
EndSub
```

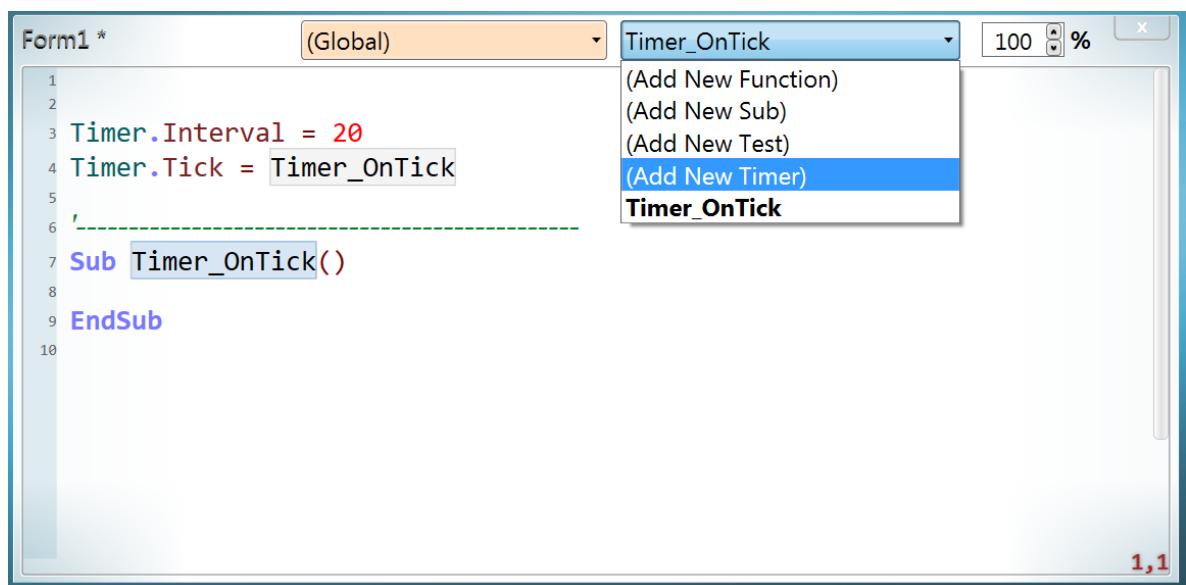


The Timer Type

The Timer object provides an easy way for doing something repeatedly with a constant interval between.

Note that this is a single timer for all the program. It is easy to operate with, but sometimes you need to do many things in different intervals, which will need some extra work to share this timer between them, and in some cases this may not be easy (like having different tasks on different forms). This is why sVB introduced the [WinTimer](#) type, to allow you to add as many timers as you need on any form in your project, including the Graphics Window.

You can easily add a timer to your code by clicking the (Add Timer) command from the top-right dropdown list of the code editor. This will add timer with a 20ms interval (which you can change) and a Timer_OnTick subroutine to handle its Tick event (where you can add your code).



A screenshot of the Visual Studio code editor showing the creation of a new timer. The code editor window has 'Form1 *' at the top left, '(Global)' at the top center, and a dropdown menu at the top right set to 'Timer_OnTick'. The main code area contains:

```
1
2
3 Timer.Interval = 20
4 Timer.Tick = Timer_OnTick
5 '
6 -----
7 Sub Timer_OnTick()
8
9 EndSub
10
```

The context menu for 'Timer_OnTick' is open, showing options: (Add New Function), (Add New Sub), (Add New Test), (Add New Timer), and Timer_OnTick. The '(Add New Timer)' option is highlighted with a blue selection bar.

The Timer type has these members:

Timer.Interval As Double

Gets or sets the interval (in milliseconds) specifying how often the timer should raise the Tick event. This value can range from 10 to 100000000.

Note that setting this value starts/resumes the timer.

Timer.Pause()

Pauses the timer. Tick events will not be raised.

Timer.Resume()

Resumes the timer from a paused state. Tick events will now be raised.

Timer.Tick

Raises an event when the timer ticks.

Example:

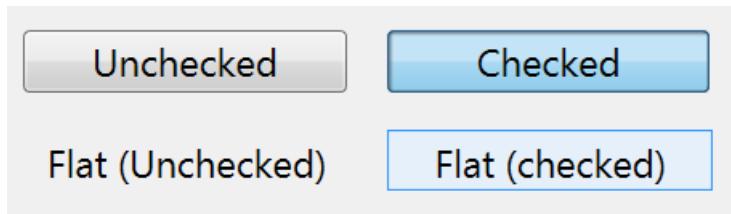
Add a label and a button on the form, and set their Text property to "Clock" and "Stop". Write this code to show the system time on the label and update it every 1 second using a timer:

```
ShowClock()  
Timer.Interval = 1000  
Timer.Tick = ShowClock  
  
Sub ShowClock()  
    Label1.Text = Clock.Time  
EndSub
```

```
Sub Button1_OnClick()
    If Button1.Text = "Stop" Then
        Timer.Pause()
        Button1.Text = "Continue"
    Else
        ShowClock()
        Timer.Resume()
        Button1.Text = "Stop"
    EndIf
EndSub
```

The ToggleButton Type

Represents a ToggleButton control, that the user can check or uncheck. The toggle button looks like a normal button, but when the user clicks it, it stays down (checked), and when he clicks it again, the button returns to its normal state (unchecked).



- Use the Checked property and OnCheck event to respond to the user choices.
- Set the IsFlat property to True to give the button a flat appearance.
- You can use the form designer to add a toggle button to the form by dragging it from the toolbox.
- It is also possible to use the Form.AddToggleButton method to create a new toggle button and add it to the form at runtime.

Note that the ToggleButton control inherits all the properties, methods and events of the [Control](#) type.

Besides, the ToggleButton control has some new members that are listed below:

ToggleButton.Checked As Boolean

This property can have one of three possible values:

- **True:** the toggle button is checked.
 - **False:** the toggle button is unchecked.
 - **An empty string "":** the toggle button is not checked nor unchecked (indeterminate state).
-

ToggleButton.Flat As Boolean

Gets or sets whether or not to show the toggleButton with a flat style.

ToggleButton.OnCheck

This event is fired when the checked state is changed.

Note that this is the default event for the ToggleButton control, so, you can double-click any ToggleButton on the form designer (say ToggleButton1 for example), to get this event handler written for you in the code editor:

```
Sub ToggleButton1_OnCheck()
```

```
EndSub
```

For more info, read about [handling control events](#).

Example:

You can see this event in action in the "Toggle Buttons" project in the samples folder. It is used to apply the font effects on the textbox when each toggle button state is changed:

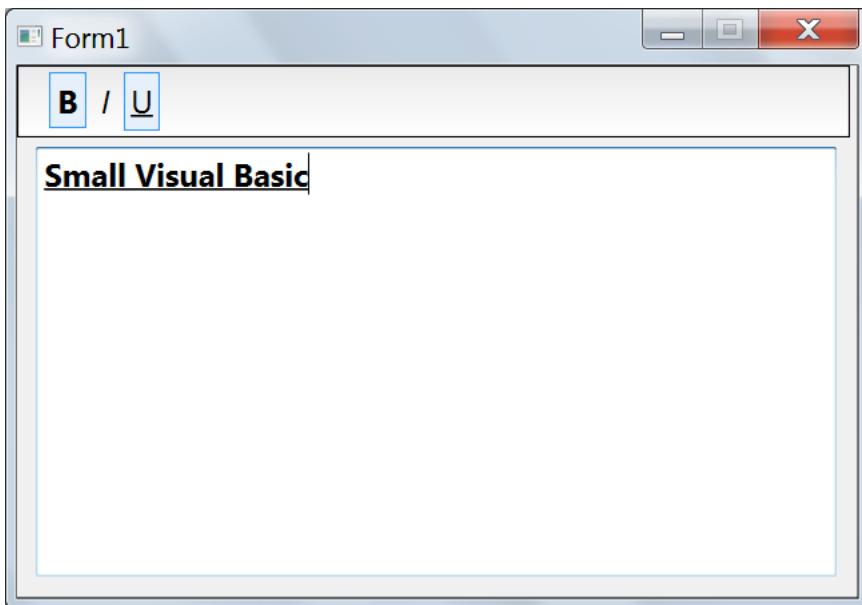
```

Sub TglBold_OnCheck()
    TextBox1.FontBold = TglBold.Checked
EndSub

Sub TglItalic_OnCheck()
    TextBox1.FontItalic = TglItalic.Checked
EndSub

Sub TglUnderlined_OnCheck()
    TextBox1.Underlined = TglUnderlined.Checked
EndSub

```



ToggleButton.Text As String

Gets or sets the test that is displayed on the toggleButton.

ToggleButton.Underlined As Boolean

Gets or sets whether or not to draw a line under the text.

ToggleButton.WordWrap As Boolean

Gets or sets whether or not to the text will be continue on the next line if it exceeds the width of the control.

You can use the [ToggleButton.FitContentHeigh](#) method to ensure that the toggle button height fits the wrapped lines.

The Turtle Type

The Turtle provides Logo-like functionality to draw shapes by manipulating the properties of a pen and drawing primitives.

- Use the Turn or DirectTurn methods to make the turtle change its direction by a given angle.
- Use the Move or DirectMove methods to make the turtle go forward for a given distance.
- Use the MoveTo or DirectMoveTo method to make the turtle move towards a given point.
- Use the PenUp method to prevent the turtle from drawing a line while it is moving, and the PenDown method to make it resume drawing.
- Use the GraphicsWindow.PenColor and GraphicsWindow.PenWidth properties to change the color and thickness of the drawn lines.
- Use the CreateFigure and FillFigure methods to paint an area drawn by the turtle, with the color defined by the GraphicsWindow.BrushColor property.
- Use the Speed property to control how fast the turtle can move and turn.
- Set the UseAnimation property to False to prevent animating the turtle while drawing curves, to make it faster, or call the DirectMove, DirectMoveTo and DirectTurn methods to have a full control on when to enable or disable animation.

The Turtle type has these members:

Turtle.Angle As Double

Gets or sets the current angle of the turtle. While setting, this will turn the turtle instantly to the new angle.

Turtle.CreateFigure()

Uses the next turtle movements to create a closed figure, so that you can fill it with a color by calling the [FillFigure](#) method.

Turtle.DirectMove(distance)

Moves the turtle by a specified distance directly without using animation. If the pen is down, it will draw a line as it moves. Use this method when you draw curves to avoid the animation overhead.

Parameter:

- **distance:**

The distance to move the turtle.

Turtle.DirectMoveTo(newX, newY)

Turns and moves the turtle to the specified location directly without using animation. If the pen is down, it will draw a line as it moves. Use this method when you draw curves to avoid the animation overhead.

Parameters:

- **newX:**

The x co-ordinate of the destination point.

- **newY:**

The y co-ordinate of the destination point.

Turtle.DirectTurn(angle)

Turns the turtle by the specified angle (in degrees) directly without using animation. Use this method when you draw curves to avoid the animation overhead.

Parameter:

- **angle:**

The angle to turn the turtle. If the angle is positive, the turtle turns to its right. If it is negative, the turtle turns to its left.

Turtle.FillFigure()

Closes the figure the Turtle created after calling the [CreateFigure](#) method, and fills it with the GraphicsWindow.BrushColor.

After calling this method, the figure is completed, and you need to create a new figure if you want to fill a new area.

If there is no figure, calling this method will do nothing.

Example:

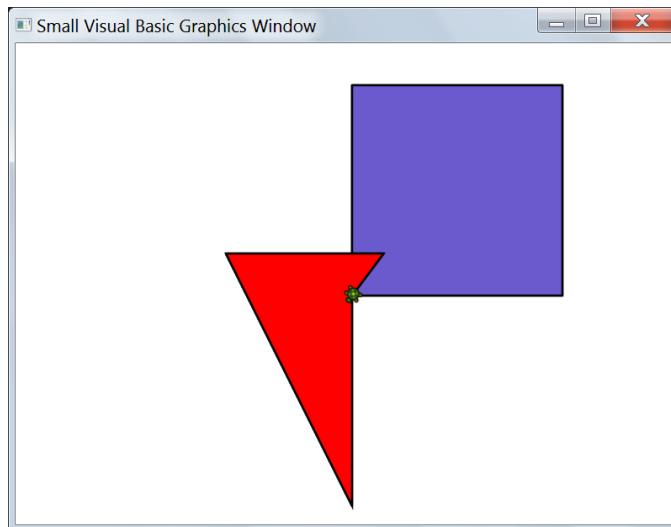
```
X = Turtle.X
```

```
Y = Turtle.Y
```

```
Turtle.CreateFigure()  
Turtle.Move(200)  
Turtle.TurnRight()  
Turtle.Move(200)  
Turtle.TurnRight()  
Turtle.Move(200)  
Turtle.TurnRight()
```

```
Turtle.Move(200)
Turtle.FillFigure()

Turtle.CreateFigure()
Turtle.TurnLeft()
Turtle.Move(200)
Turtle.MoveTo(200, 200)
Turtle.MoveTo(350, 200)
Turtle.MoveTo(X, Y)
GraphicsWindow.BrushColor = Colors.Red
Turtle.FillFigure()
```



🎨 Turtle.Height As Double

Gets or sets the Turtle height. The default value is 16.

Note that changing the turtle height will also set its width to the same value.

⚙️ Turtle.Hide()

Hides the Turtle and disables interactions with it.

Turtle.Move(distance)

Moves the turtle to a specified distance. If the pen is down, it will draw a line as it moves.

Note that when you set the UseAnimation property to True, the Move method will call the DirectMove method to move instantly without animation.

Parameter:

- **distance:**

The distance to move the turtle.

Turtle.MoveTo(newX, newY)

Turns and moves the turtle to the specified location. If the pen is down, it will draw a line as it moves.

Note that when you set the UseAnimation property to True, the MoveTo method will call the DirectMoveTo method to move instantly without animation.

Parameters:

- **newX:**

The x co-ordinate of the destination point.

- **newY:**

The y co-ordinate of the destination point.

Turtle.PenDown()

Sets the pen down to enable the turtle to draw as it moves.

Turtle.PenUp()

Lifts the pen up to stop drawing as the turtle moves.

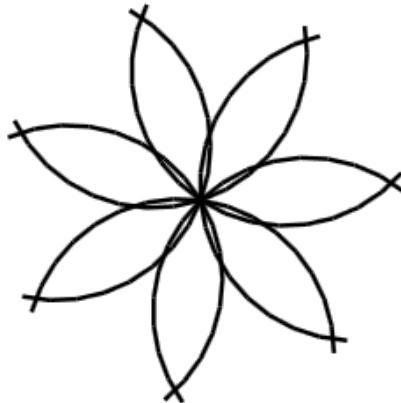
Example:

The following code uses the turtle to draw a flower by drawing intersected half circles. We control which parts of the circle are drawn by setting the pen up and down:

```
N = 7
Sides = N * 5
Length = 2 * Math.Pi * 70 / Sides
Angle = 360 / Sides
Turtle.Speed = 10
M = Math.Floor(N / 2)
A = M * Math.Pi / N
X = Sides * Math.Cos(A)

For J = 1 To N
    For I = 1 To Sides
        If I < X + 1 Then
            Turtle.PenDown()
        ElseIf I > Sides - X Then
            Turtle.PenDown()
        Else
            Turtle.PenUp()
        EndIf
        Turtle.Move(Length)
        Turtle.Turn(Angle)
    Next
    Turtle.Turn(360 / N)
Next

Turtle.Hide()
```



⌚ Turtle.Show()

Shows the Turtle to enable interactions with it.

🎨 Turtle.Speed As Double

Specifies how fast the turtle should move.

Valid values are 1 to 15 and the default value is 5.

If Speed is set to 50, the turtle moves and rotates very fast, but using many successive moves and turns (like when you draw curves) can slow down the turtle, because of the animation overhead. In such case, you can turn off the animation by setting the UseAnimation property to False.

⌚ Turtle.Turn(angle)

Turns the turtle by the specified angle (in degrees). If the angle is positive, the turtle turns to its right. If it is negative, the turtle turns to its left.

Note that when you set the UseAnimation property to True, the Turn method will call the DirectTurn method to turn instantly without animation.

Parameter:

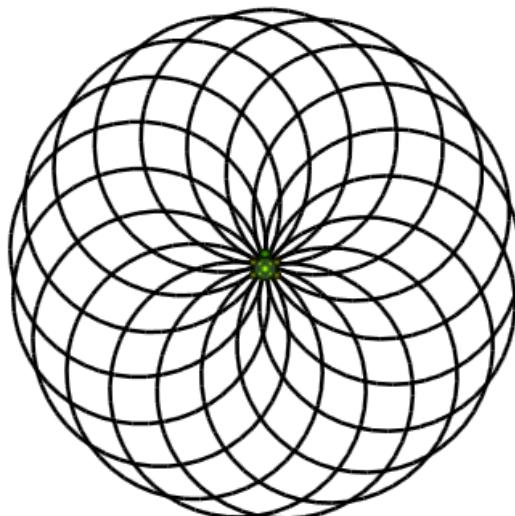
- **angle:**

The angle to turn the turtle.

Example:

The following code uses the turtle to draw a sphere by drawing intersected circles:

```
Sides = 50
Length = 400 / Sides
Angle = 360 / Sides
Turtle.Speed = 10
_Shapes = 17
For J = 1 To _Shapes
    For I = 1 To Sides
        Turtle.Move(Length)
        Turtle.Turn(Angle)
    Next
    Turtle.Turn(360 / _Shapes)
Next
```



⌚ Turtle.TurnLeft()

Turns the turtle 90 degrees to the left.

Turtle.TurnRight()

Turns the turtle 90 degrees to the right.

Turtle.UseAnimation As Boolean

Gets or sets a value that indicates whether or not to animate the turtle moves and turns. The default value is True.

Note that setting this property value to False, will make the Move, MoveTo and Turn methods call the DirectMove, DirectMoveTo and DirectTurn methods to avoid using animation. You can still call this Direct... methods manually regardless of the value of the UseAnimation property, which allows you to mix animated and direct moves and turns to get the effect you desire.

Turtle.Width As Double

Gets or sets the Turtle width. The default value is 16.

Note that changing the turtle width will also set its height to the same value.

Turtle.X As Double

Gets or sets the X location of the Turtle. While setting, this will move the turtle instantly to the new location.

Turtle.Y As Double

Gets or sets the Y location of the Turtle. While setting, this will move the turtle instantly to the new location.

The UnitTest Type

The UnitTest type provides methods to test your code. It is defined as an external library, which is created with Small Visual Basic, and you can find its source code in the UnitTest project in the samples folder.

The UnitTest type has these members:

UnitTest.AddError(msg)

Adds a test error message to the Errors array.

Parameter:

- **msg:**

The message that describes the test error.

UnitTest.AssertEqual(actualValue, expectedValue, testName) As String

Checks whether or not the actual value resulted from the test is the expected value.

Parameters:

- **actualValue:**

The actual value of the test. You can send a single value or an array of values.

- **expectedValue:**

The expected value from the test. You can send a single value or an array of values.

- **testName:**

The name of the test, to be displayed in the result.

Returns:

A string message that displays the test name and its result, saying whether it passed or failed and why.

Example:

You can see this method in action in the "UnitTest Sample" project in the samples folder. It is used in the Form2.sb file to test the ReverseString function (written in the same file) like this:

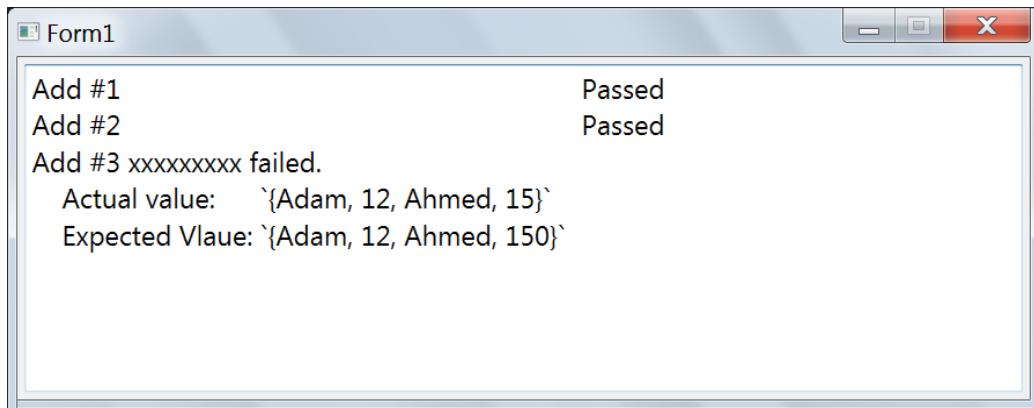
```
Function Test_ReverseString()
    Return UnitTest.AssertEqual(
        {
            ReverseString("123456"),
            ReverseString("abcdef"),
            ReverseString("1 + 3"),
            ReverseString("")
        },
        {"654321", "fedcba", "3 + 1", ""},
        "ReverseString"
    )
EndFunction
```

This test passes, so, its result is the string:

"ReverseString	Passed"
----------------	---------

Where the tabs are used to format the test name and its result in two columns.

In the other hand, tests written in the Form1.sb shows you how the result looks like when a test fails:



⌚UnitTest.AssertFalse(actualValue, testName) As String

Checks whether or not the actual value resulted from the test is False.

Parameters:

- **actualValue:**

The actual value of the test. You can send a single value or an array of values.

- **testName:**

The name of the test, to be displayed in the result.

Returns:

A string message that displays the test name and its result, saying whether it passed or failed and why.

⌚UnitTest.AssertTrue(actualValue, testName) As String

Checks whether or not the actual value resulted from the test is True.

Parameters:

- **actualValue:**

The actual value of the test. You can send a single value or an array of values.

- **testName:**

The name of the test, to be displayed in the result.

Returns:

A string message that displays the test name and its result, saying whether it passed or failed and why.

Example:

```
X = "."
Y = "test"
```

```
TextWindow.WriteLine({
    UnitTest.AssertTrue(
        {
            Y.Append(X) = "test.",
            Text.Append(Y, X) = "test."
        },
        "Text.Append"
    ),
    UnitTest.AssertTrue(
        {
            "." + "1" + 4 = 4.1,
            Text.Append(".", {"1", "4"}) = "=.14",
            X.Append({"1", "4"}) = ".14"
        },
        "Text.Append"
    )
})
```

The consol window will show the following results:

```
Text.Append  
Text.Append xxxxxxxx failed.  
    Actual value: `'{True, False, True}`  
    Expected Vlaue: `'{True, True, True}`
```

Passed

UnitTest.Errors As Array

Returns an array containing the test errors that was added by the AddError, AssertEqual, AssertFalse, or AssertTrue methods.

UnitTest.RunAllTests(resultsTextBox)

Runs all the test functions written in the Global file and every form in the project, and shows the overall test report in the given resultsTextBox.

Notes:

1. You should remove (or comment out) any call to the [Form.RunTests](#) or UnitTest.RunFormTests Methods in any form before calling the UnitTest.RunAllTests method, because running the form tests will show it to ensure it is loaded and initialized, and this will execute and call to the [Form.RunTests](#) or UnitTest.RunFormTests Methods written in the form global area, which means that this form will run the tests twice.
2. You can't use the sVB Debugger to debug test functions that are called using this method, because the UnitTest type is defined in an external library. If you want to debug test functions, call them individually or use the [Form.RunTests](#) method for each form to be able to debug its test functions.

Parameter:

- **resultsTextBox:**

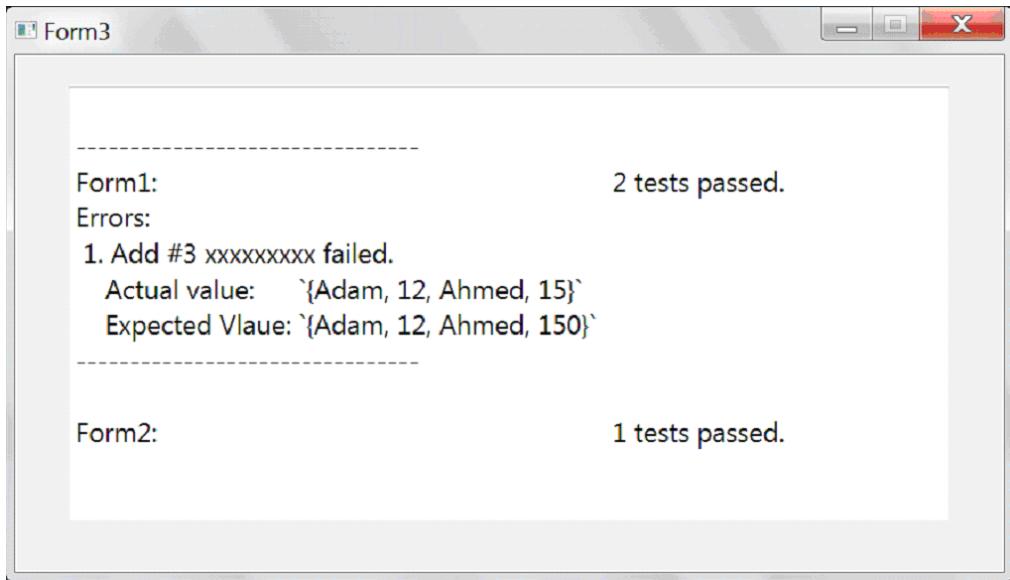
The textbox that will display the test results.

Example:

You can see this method in action in the "UnitTest Sample" project in the samples folder. It is used in the FrmTests.sb file to run all the tests written in Form1 and Form2.sb files, by just executing this single line of code:

UnitTest.RunAllTests(textBox1)

This is what you will see when you run the project:



💡 **UnitTest.RunGlobalTests(formName)**

Runs the tests of the global.sb file, and shows the results on the given form.

Note that this method actually shows the form and calls its [Form.RunGlobalTests](#) Method.

Parameter:

- **formName:**

The name of the form that will show the test results.

UnitTest.RunFormTests(formName)

Runs the tests of the given form.

Note that this method actually shows the form and calls its [Form.RunTests](#) Method.

Parameter:

- **formName:**

The name of the tested form.

Example:

See the [Form.RunTests](#) Method for more info about how to write test methods and an example.

The WinTimer Type

Represents the WinTimer control, which provides an easy way for doing something repeatedly with a constant interval between. Use the Interval property to set the time between ticks, and write the code you want to be executed in the OnTick event handler. This object is different than the [Timer object](#), as you can add many WinTimers as you want to each form in the project, while there is only a single Timer object which was suitable to be used in single Small Basic code files.

You can't use the form designer to add Win timers to the form, but you can use the [Form.AddTimer](#) method to add a new timer to the form. You can also use the [Controls.AddTimer](#) method to add a new timer to the graphics window.

The "Stop Watch" project in the samples folder shows you how to add two timers on the form:

```
ClockTimer = Me.AddTimer("clockTimer", 1000)  
ClockTimer.OnTick = ShowClock  
StopWatch = Me.AddTimer("StopWatch", 10)  
StopWatch.Pause()  
StopWatch.OnTick = ShowStopwatch
```

Note that you can add a win timer to your code by clicking the (Add Timer) command from the top-right dropdown list of the code editor. At the first time this will add a classic Small Basic timer, but after that every click will add a new win timer, which will be added to the form if the current code file belongs to a form, or to the GW controls if the current file is a single Small Basic file without a form.

Each timer will have a 20ms interval (which you can change) and a Timer#_OnTick subroutine to handle its Tick event, where the # is a

placeholder for the timer number like Timer1_0bTck (and you can change this name to a more expressive name).

The screenshot shows a code editor window titled "Form1 *". The code is as follows:

```
7 Timer1 = Me.AddTimer("timer1", 20)
8 Timer1.OnTick = Timer1_OnTick
9
10
11 Timer2 = Me.AddTimer("timer2", 20)
12 Timer2.OnTick = Timer2_OnTick
13
14 '
15 Sub Timer2_OnTick()
16
17 EndSub
18
19 '
20 Sub Timer1_OnTick()
21
22 EndSub
```

A context menu is open over the line "Timer2.OnTick = Timer2_OnTick". The menu items are:

- (Add New Function)
- (Add New Sub)
- (Add New Test)
- (Add New Timer)** (highlighted in blue)
- Timer_OnTick
- Timer1_OnTick
- Timer2_OnTick

The status bar at the bottom right shows the coordinates 15, 14.

The WinTimer members are listed below:

⚡ **WinTimer.Interval As Double**

Gets or sets the interval (in milliseconds) specifying how often the timer should raise the Tick event.

Setting this property will enable the timer, and the tick event will be raised once you provide a handler for it.

⚡ **WinTimer.OnTick**

This event is fired when user releases the left mouse-button

For more info, read about [handling control events](#).

WinTimer.Pause()

Pauses the timer, so that the Tick events will not be raised until you call the Resume method.

WinTimer.RemoveOnTickHandler()

Removes the OnTick event handler of the current timer.

Note that sVB provides a syntax to do the same job, by setting the OnTick event to **Nothing**, which is translated to a call to the RemoveOnTickHandler method when the program is compiled.

So, this code:

```
WinTimer1.OnTeck = Nothing
```

is equivilant (and will be compiled to) this code:

```
WinTimer.RemoveOnTickHandler()
```

WinTimer.Resume()

Resumes the timer from a paused state. Tick events will now be raised.

Example:

You can see this method in action in the "Stop Watch" project in the samples folder. It is used in the BtnStart_OnClick event handler to make the stopwatch continue working after it was paused:

```
Sub BtnStart_OnClick()
    If Counting Then
        Stopwatch.Pause()
        BtnStart.Text = "Resume"
        Counting = False
        BtnReset.Enabled = True
```

```
Else
    StopWatch.Resume()
    LastTick = Date.Now
    BtnStart.Text = "Puase"
    Counting = True
    BtnReset.Enabled = False
EndIf
EndSub
```