The KBasic Manual - A Quick Reference Guide for the Development of KBasic Applications

First Edition

This edition applies to release 1.6 of KBasic and to all subsequent releases and modifications until otherwise indicated in newer editions. Make sure you are using the correct edition for your version of the product. The term "KBasic" as used in this publication refers to the KBasic product set (January 2010).

About This Manual

Welcome to "The KBasic Manual", your guide to the development of KBasic applications. In this manual you will find all the necessary information you need to successfully build your own KBasic programs. If you read the entire manual and complete all the examples, you will be able to write both simple and complex KBasic programs, whether they are written in standard BASIC only or with the modern GUI features of the graphical user interfaces that are available with forms. If you are new to KBasic, read the first few chapters to learn about this new and exciting programming language. Then try out the examples in this manual.

Purpose of This Manual

This manual is not meant to be a detailed or comprehensive reference manual. The principal purpose of this manual is to act as a quick and practical reference while you are programming. For a complete reference of the KBasic programming language look inside KBasic's IDE (Integrated Development Environment) under *Help* ⇒ *Online Documentation* ⇒ *Language Reference*. Here you will find a complete and detailed listing of all objects, their events and procedures, methods and characteristics as well as a complete reference of all operators and commands. Apart from precise syntax definitions, you will also find a detailed explanation of the available parameters in each case.

For the less experienced KBasic programmer, this manual includes an extensive introduction to the KBasic programming language with numerous examples of the most important elements of the language, including the more complex object oriented aspects of KBasic.

Why You Should Read This Manual

This manual was written for both the absolute beginner and for the person with some background knowledge in BASIC. Even experienced Visual Basic developers should feel very comfortable with this manual.

The KBasic Manual:

- is an asset for anyone doing serious development with KBasic
- gives you a solid background in the fundamentals of KBasic
- takes you step-by-step through the features of the language

This manual is meant for people, who:

- want to learn a simple and powerful programming language for Linux, Windows, or Mac OS X
- are experienced C/C++, Java, or Visual Basic developers, who want to switch to KBasic or want to extend their own knowledge and abilities
- have heard of KBasic as a new programming language for Linux, Mac OS X, and Windows and want to learn it

Note of Thanks

I am very grateful to all the people who helped me complete this manual. Special thanks to Nadja, my girlfriend, and to everyone who bought KBasic Professional, including all those people who supported me in other ways. A big thanks also to my parents.

About the Author

Bernd Noetscher is a software developer and the main developer of the KBasic programming language. In his spare time he goes dancing, reads many books, and plays the piano. He is also interested in theatre and cinema. His private website is hosted at www.berndnoetscher.de

Supporters

Thank you! Nadja Hepp for proofreading this manual and for beta testing. Thanks to everyone on the Internet who submitted corrections and suggestions, especially Christopher Scott Wyatt and his wife Susan; you have been tremendously helpful in improving the quality of this manual and I could not have done it without you. Thank you all!

Give Us Feedback

Please help me improve this manual. To do this, we need your help. As the reader, you are potentially a most important contributor and valued critic of this manual. We respect your opinion and would like to know what improvements we could make. We would also like to get positive comments, too. In addition, if you find any mistakes in the text or errors in the example programs, please send an e-mail to info@kbasic.com. Bear in mind that due to the high volume of mail we receive, we may not be able to reply to every message we get. Thank you very much.

Please Note: We cannot help you with technical problems related to the topics covered in this manual. For more information on KBasic refer to the online documentation or register with the KBasic Community Forum. (www.kbasic.com)

New Set of Manuals About KBasic

KBasic was developed by KBasic Software and has aroused much interest on the Internet among businesses and developers. In response, we documented this exciting new technology with a set of new manuals. This series of manuals covers language references, introductory volumes, API references, and advanced topics on programming in KBasic, such as databases and networks.

Conventions Used in This Manual

In this manual, normal text appears in a regular font. Here is an example: "This is normal text".

Syntax and source code appear in Courier New with a grey background. For example:

```
Dim i As Integer
```

Important references and keywords are shown in italics. For example: *Arguments*

Sources for Further Information

Visit the KBasic website at http://www.kbasic.com/ and look under *Manuals*. Also, search the Internet using Google at http://www.google.com/ to find more information covering programming in BASIC, including Visual Basic 6.

Introduction to Programming

If you are not a newcomer to programming computers, you may want to skip this section and go straight to the introduction on the KBasic programming language, instead. If you are new to programming or you just want to read along, let's get started.

Have you ever wanted to know how a computer program works? If so, there might be a computer programmer somewhere inside you waiting to get out. As a beginner, you may have found computer programming somewhat intimidating and possibly downright frightening. If you've ever felt this way, this manual is here to prove that programming computers can be fun, rewarding, and not too difficult.

You have probably heard a lot about KBasic. In fact, you may have already downloaded it and played around with the IDE and some of the sample programs, and have your own ideas about what a program is and how it works. Now, you want to create your own KBasic programs. But, before we get started, you ought to have a solid understanding of what programming is all about. So, we encourage you to read this introduction thoroughly, to better aquaint yourself with the fundamental concepts involved in programming computers. It will be worth it!

The Big Secret

The computer programming world has a well-kept secret. You will probably find this secret hard to believe. Computers are stupid! A computer can do absolutely nothing on its own. Computers without programmers are useless. Computers can do only what they are told to do. This means computers can only do tasks that humans already know how to do.

So, why are computers so great? Computers may not be smart, but they can perform endless calculations in seconds and easily do repetitive tasks over and over without making any mistakes. Programmers (also known as developers), are the people who tell computers what to do. For example, when you sit yourself down in front of a word processor and write a letter, you are indirectly giving the computer commands to execute or act upon. You are using the commands contained within the word processing program which were written by a programmer. The computer program actually tells the computer what to do.

The bottom line is this: if you want to give commands directly to your computer yourself, you must learn to write programs.

Why would you want to learn to program? There are many reasons for doing this. For example, you may want to:

- be able to write the programs you really need
- learn more about how computers work
- write programs for your friends, family, colleagues, or even your boss
- have a fun and rewarding hobby or career

These are all good reasons, and you might even have a better one for learning to program. But, whatever your reason, once you start programming, you will find it can be fascinating, interesting, and addictive.

What is a Computer Program?

A computer program is a list of instructions or commands telling a computer what to do. The computer follows these instructions, one-by-one, until it reaches the end of the program.

Each line in a computer program is usually a single command that the computer must do. And each command normally does only one small task, such as printing a name on the screen or adding a few numbers together. When you put hundreds, thousands, or even hundreds of thousands of these commands together, your computer can do great things: calculate complex mathematical problems very quickly, print a document, draw pictures, or enable you to play computer games.

As you will see in the next paragraph, computer programs are written in a programming language. There are many different programming languages to choose from. For our purposes, we will be concentrating on the BASIC language, from which KBasic has evolved.

Programming Languages

Computers do not understand German, English or any other human language. They cannot even understand BASIC (the computer language which KBasic is based upon). Computers understand one thing only, machine language or machine code, which is entirely composed of ones and zeroes from the binary numeral system. Programming languages like BASIC allow people to write programs in an English-like language, which is then translated into machine language by another program called an interpreter or compiler. This is done so that the computer can understand the English-like program in its own language – much like someone translating a foreign language for you so you can understand what the other person is saying to you in your own language.

KBasic programs are a dialect of the BASIC computer language that was developed to help make it easier for people to program computers. KBasic uses words and symbols which are used to represent the binary instructions used in machine language. These words and symbols are much simpler for programmers to understand and remember. As a programmer, it is so much easier to use a command like `Print` instead of a binary-coded instruction like `10001000111011010001111011100010`, for example, which is long and difficult to remember.

Moreover, KBasic enables a programmer to visually assemble a program's appearance from parts in a toolbox, which makes it easier for you to communicate with your computer as a software developer. Instead of thinking like a machine in long binary numbers, you can think like a human being using words and symbols to construct your program.

History of the BASIC Name

Sometimes you see the name of the BASIC programming language spelt with lowercase letters like this: Basic. Even KBasic uses this spelling. However, the word BASIC actually started out as an acronym, which is why you also see the name spelled in all capital letters. The BASIC acronym stands for Beginner's All-purpose Symbolic Instruction Code. A BASIC program uses symbols and words (and a few numbers) that people can understand. How is it possible that the computer can understand and run a BASIC program?

When you run KBasic, you are also loading a compiler. A compiler is a special program that takes the words and symbols from a KBasic program and translates them into machine language which the computer can understand. Your computer would not have any idea of what to do with your program without the compiler's help. There are many computer languages, including Java, C++, and BASIC. However, all computer languages have one thing in common: they can be read by humans and therefore must be converted to machine language before the computer can understand them.

All Kinds of BASIC

The first BASIC programming language was designed in the early 60's by John Kemeny and Thomas Kurtz to help non-science students program computers. Since that time, numerous versions or dialects of this language have been written, of which KBasic is only one of them. The old DOS operating system, for example, came with the QBasic language. Older versions of Microsoft's Windows operating system came with Visual Basic and their Office Suite with VBA (Visual Basic for Applications). All these software packages allow you to create computer programs in BASIC, but they all implement the BASIC language somewhat differently.

However, of the versions of BASIC mentioned here, KBasic enables you to write cross-platform applications for Linux, Mac OS X and Windows.

Compilers and Interpreters

Some computer languages, including some types of BASIC, convert a program to machine language one line at a time as the program is running. And other languages, such as C and C++, use a compiler to convert the entire program all at once before the program is run. In either case, all programming languages must be converted to machine language in order for the computer to understand the program.

A compiler changes your program into an executable file that can be run directly by a computer. An executable program is actually a machine language program that is ready for your computer to read and understand. With few exceptions, most computer programming languages make use of a compiler. However, in some cases, hybrid compiler-interpreters are used by some languages, such as Java and KBasic, which are a mix of compiler and interpreter.

The Programming Process

Now that you know something about computer programs, how do you go about writing one? Creating a computer program is fairly easy, although it can be a long process when writing lengthy programs. Writing a KBasic program requires development steps similar to those you use when writing a report. The following list outlines possible steps you can take to write your own programs:

1. Come up with an idea for a program, e.g. a data capture program, and sketch out on paper how it might look on the screen.
2. Create the program using KBasic's toolbox and source code editor.

3. Save the program to disk.
4. Run the program and test it to see how it works.
5. Fix any programming errors (also known as bugs).
6. Repeat steps 3 to 5 until there are no more errors.

Most of the steps in the programming process are repeated over and over as errors are discovered and corrected. Even experienced programmers cannot write error free programs unless the program is extremely short. Programmers often spend more time fine-tuning their programs than they do writing them initially. It is important to do the fine-tuning, because we are not as logical as we like to think we are.

Moreover, our minds are incapable of remembering every detail required to make a program run perfectly. Only when a program crashes (doesn't work) or does something unexpected can we hope to find errors hiding within code. Computer experts say there is no such thing as a program without bugs. After you start writing full-length programs, you will see how true this statement is.

Attack of the Bugs

Programming or software bugs are errors that may stop your program from running correctly. Many programming bugs are merely annoying or troublesome (minor bugs), but some can have serious consequences (major bugs). Therefore, before a programmer can release their program to the public, they must correct as many errors as possible, especially the major ones.

Is programming difficult?

Well, yes and no.

It is easy to learn to write small programs with KBasic. The language is logical, English-like, and easy to understand. With only minimal practice, you can write many useful and fun programs. Actually, what you will learn in this manual is enough programming for just about anyone who is not planning to become a professional programmer.

However, if you want to make programming a career, you have much to learn that is not covered in this manual. For example, consider a word-processing program such as the one found in OpenOffice, which took dozens of programmers many years to write. To write such complex software, you must have a good knowledge of how a computer works. Additionally, you must have spent many years learning the skills of professional computer programming.

Still, there is a lot you can do with KBasic, whether you are interested in writing utilities, small applications, or even computer games. And, step-by-step, you will discover programming in KBasic is not as difficult as you might have thought.

How Much KBasic Do I Need to Know to Use It?

It is possible to develop KBasic programs without having to write any lines of source code. There are many places to obtain BASIC programs, such as books, the Internet, and even from your friends and colleagues. You can drop these programs into KBasic and have an application ready to use. However, sometimes these programs need modifications and this is where the real challenges to your programming expertise occurs.

Something You Need to Know

Before you start writing your own programs with KBasic, it is important to know what computer programs are and how they work. Such background knowledge will help you understand why KBasic does some of the things it does, as well as make it easier to locate errors in your programs.

You should remember the following points:

- A computer can do only what a human instructs it to do.
- There are many computer languages and the language you learn in this manual is a dialect of BASIC.
- A computer program is a list of commands the computer follows from beginning to end.
- A BASIC program must be converted to machine language before the computer can understand it. KBasic's compiler does this conversion for you.

- Writing a program is like writing a text document. You must write several drafts before the program is completed and ready to use.
- Programming with KBasic is as easy or as difficult as you want it to be.
- You should also have a fundamental understanding of object-oriented programming, such as C++ or Java. Many of the principles and concepts of KBasic are based upon those found in these two languages.
- There are also some differences between KBasic and C++ that you should know about as well.
- If you are using databases, a good understanding of SQL (structured query language) and client-server models is necessary to build more complex database-aware KBasic programs.

Introduction to the KBasic Programming Language

Welcome to the KBasic programming language. This chapter is concerned with what exactly KBasic is, why you should learn KBasic, and what makes KBasic different from other programming languages. We also cover the fundamentals of KBasic, explain which background knowledge is needed, and show you how to write your first KBasic program.

What is KBasic?

KBasic is a powerful programming language designed to be intuitive and easy to learn. KBasic also represents a further bridge between Linux, Mac OS X, and Windows. KBasic is a new programming language, a further BASIC dialect related to Visual Basic 6 and Java. More precisely, KBasic is an object-oriented and event-driven programming language, developed by KBasic Software (www.kbasic.com), and is designed particularly for the needs of GUI-developers to strengthen the development process. C/C++ developers feel that BASIC is a beginner's language, but with KBasic you can write many applications that may otherwise have been written in the more difficult C/C++ languages were it not for KBasic. KBasic is an easy, usable, object-oriented, event-driven, interpreted, stable, platform-independent, modern programming language.

Warning

In order to ensure the compatibility of your programs with future versions of KBasic, you should avoid using old elements of the KBasic language such as 'GoSub', 'DefInt', 'On Error GoTo', etc., which are only supported for historical reasons to help people migrate to KBasic – for more information, please read the migration from VB 6 references at the end of this manual. Therefore, please use the modern elements of the KBasic language which are described in this manual.

KBasic Examples

The examples in this manual are in English. KBasic was originally developed as open source software and will continue to be open in the future. The project was started by Bernd Noetscher in the summer of 2000. After many disappointments (due to lack of volunteer support), he decided to develop the programming language as a commercial product for Linux, Mac OS X and Windows. At present, KBasic version 2.0 is under construction. Version 2 will be more user-friendly, more efficient, and have additional programming libraries with new object-oriented features.

KBasic is Easy to Learn

KBasic is a simple but powerful language to use and was designed for rapid development (RAD) with easy error tracing. KBasic is modelled after standard BASIC, is syntax compatible to Visual Basic 6 (apart from its components), and is therefore very similar. KBasic contains object-oriented ideas from Java and C++, with the confusing elements of those languages being omitted. Like Java, KBasic uses references instead of pointers to objects. In addition, the KBasic garbage collector automatically manages the allocation and deallocation of memory for the programmer. You do not have invalid pointers and memory leaks. Instead, you can spend your time productively in the development of your application. KBasic is actually a complete, elegant and powerful programming language.

KBasic is Object-Oriented and Event-Driven

KBasic is object-oriented and event-driven. If you are familiar with procedural programming in BASIC, you probably think that if you use KBasic you will have to change the way you write your programs and make them all object-oriented. But, this is not true. You can still continue to program with procedures only, if you prefer.

However, if you get to know how powerful this new object-oriented paradigm is, you will see how easy it is to develop re-useable, complex application code clearly and in a modular way. KBasic is object-oriented regarding the binding to C++ programming libraries, or built-in KBasic classes, with true inheritance. Unlike many other programming languages, KBasic was designed from the beginning as an object-oriented programming language. So, most things in KBasic are objects. However, simple data types (like numeric, strings and boolean values), are the only exceptions. Although KBasic has been designed in such a way that it looks similar to Visual Basic, you'll find that KBasic avoids many problems associated with VB6 in an elegant way.

KBasic is Interpreted

KBasic produces pseudo code instead of machine code. In order to run a KBasic program you can use the KBasic development environment (IDE) or the menu entry 'Make Runtime' (only available in the Professional version). KBasic is thus an interpreted programming language. KBasic pseudo code is an architecture-independent format, which was designed to run programs efficiently on different platforms.

KBasic is Stable

KBasic was designed to facilitate the writing of very reliable and durable software. Of course, KBasic does not eliminate the need for quality control. It is still possible to write unreliable software by using bad coding practices. However, by eliminating certain kinds of programming errors, KBasic makes it easier to write reliable software.

KBasic is a typed language. This means that KBasic can do very detailed examinations of your program for potential type inconsistencies. KBasic is more strictly typed than standard BASIC. One of the most important characteristics regarding reliability is correct memory management. By not supporting pointers, KBasic excludes the danger of accidently overwriting data in memory while a program is running. Similarly, KBasic's garbage collector prevents memory leaks and other malicious errors with dynamic memory allocation and deallocation. In addition, the KBasic interpreter examines different things at run time, e.g. whether arrays and strings are still within their bounds. Exception handling is another new feature. An exception is a signal for a exceptional program condition, which occurs like an error. By using the `try`/`catch`/`finally` statements, you can place all your error processing routines in one place, making it easier to catch and fix errors.

KBasic is Fast

KBasic is an interpreted programming language. Therefore, it will never be as fast as a compiled machine code language, like C. Actually, KBasic is roughly 20 times slower than C. Now, before you run away, bear in mind that this speed is fast enough for nearly all kinds of GUI applications, which often wait for the user to input something.

When analyzing the power of KBasic, you need to take into account the placement of KBasic with respect to other programming languages. On the one end are high-level, fully-interpreted languages like PHP and the UNIX-shell. These languages are very useful for creating a draft, but they are executed slowly. On the other end are the fully compiled machine code programming languages. These languages provide extremely fast programs but are not very portable or reliable.

KBasic falls in the middle of these languages. The performance of KBasic is much better than high-level programming languages, but have the ease and portability of those languages. Although KBasic is not as fast as a compiled C program, it is platform-independent, providing the ability to write stable programs for Windows, Mac OS X, and Linux.

Why KBasic Succeeds

Traditionally, new programming languages have suffered from the disadvantage of forcing you to abandon everything you know and make you start from scratch with a new set of concepts and syntax – the rationale being that it is better in the long run to lose all the old baggage and start afresh. This may be true in the long run. However, in the short term, much of that baggage is valuable. The most valuable elements may not be the existing code base, but instead the existing mind base. If you are a competent VB6 programmer and must drop everything you know about VB6 in order to adopt a new programming language, you immediately become much less productive for many months, until your mind fits around the new paradigm. Whereas, if you can leverage your existing VB6 knowledge and expand upon it, you can continue to be productive with what you already know while moving into the world of object-oriented programming.

The problem with learning a new language from scratch is loss of productivity. No company can afford to suddenly lose the productivity of a software engineer because they are learning a new language. For this reason, KBasic is

very similar to VB6. It allows you to continue creating useful code, and applying KBasic's new features as you gradually learn and understand them. This may be one of the most important reasons for the success of KBasic. In addition, most of your existing VB6 code is still usable in KBasic.

A Simple Example

Enough! Now that you have an insight into what KBasic is all about, let us stop discussing abstract phrases and instead study a simple KBasic program. Here is the much loved and favourite "Hello World" example:

```
' program beginning

CLS
Print "Hello World!"
Print
Print
Print "                 / `._      .        .       _.' \"
Print "               '.@ = `.     \      /     .' = @.'"
Print "                \ @`.@ `.    \    /    .' @.'@ / "
Print "                 \;`@`.@ `.   \ /   .' @.'@`;/  "
Print "                  \`.@ `.@ `'.(*).'` @.' @.'/   "
Print "                   \ '=._`. @ :=: @ .'_.=' /    "
Print "                    \ @  '.'..'='..'.'  @ /     "
Print "                     \_@_.==.: = :.==._@_/      "
Print "                     /  @ @_.: = :._@ @  \      "
Print "                    /@ _.-'  : = :  '-._ @\     "
Print "                   /`'@ @ .-': = :'-.@ @`'`\    "
Print "                   \.@_.=` .-: = :-. `=._@./    "
Print "                     \._.-'   '.'   '-._./      "
Print
Print "... you just ran your first KBasic program!"
```

Input these lines into a new KBasic code window. After running it in KBasic you will see a nice butterfly on your screen. Now let's go on to the next chapter.

Rapid Application Development

You can use KBasic's visual programming features from within the IDE to quickly develop KBasic applications. In the Form designer, you can:

- Design the user interface for your program.
- Specify the behavior of the user interface elements.
- Define the relationship between the user interface and the rest of your program.

In addition to its visual programming features, KBasic gives you the ability to quickly complete many tasks, including the ability to create new program elements. You can also import Visual Basic 6 projects and files from the file system as needed, and export textual data from the IDE to the file system as a CSV file.

In KBasic, a program element is one of the following:

- **Project**: the top-level program element. A project contains classes, modules, and forms.
- **Form**: a KBasic visual programming feature.
- **Class/module**: KBasic language constructs. Classes contain methods and other statements.
- **Subroutine/function/method**: KBasic language constructs.

KBasic gives you the software tools you need to develop industrial-strength code. Specifically, you can use the integrated visual debugger to examine code while it is running and browse your code at the level of project, class, or method.

How Can I Get KBasic Professional Edition?

You can get KBasic Professional Edition directly from KBasic Software. If you would like to order KBasic, make an enquiry, or have any questions, please send an e-mail to: sales@kbasic.com. The current price for KBasic Professional Edition is €24.95 (inclusive of shipping costs), or roughly US$35. You will receives three program versions of KBasic, one for each of the following platforms: Windows, Mac OS X, and Linux. Furthermore, you will also receive a licence for each version of KBasic, granting you the legal right to develop private or commercial

programs without having to pay any additional fees or royalties.

Installation of KBasic / Installation description / System needs

KBasic is easy to install and needs about 200 MB of space on your hard disk.

Officially supported are:

- Windows XP/2000/Vista
- Linux
- Mac OS X 10.3 and above

KBasic will not run on the following systems in the future: Windows 95/98/Me

You need at least 64 MB RAM and your CPU should run at least at 400 Mhz. We recommend a screen resolution of 1024 x 768, 32bit, True Color

Starting KBasic

You can start KBasic by doing one of the following:

- For Linux, select KBasic on the desktop
- For Windows, select KBasic in the program menu or Desktop Shortcut
- For Mac OS X, select KBasic from your Applications Folder (*Hint - create a Dock Shortcut)

Now that KBasic is up and running, the main window of KBasic appears. Use this window to access other windows, create program elements, and view the contents of program elements.

FAQ

What is the difference between KBasic and Visual Basic 6?

KBasic is very similar to VB6, making it easy for VB6 developers to learn. There are also some important differences between KBasic and VB6, such as the pre-processor and the exception handling. One of the main differences between KBasic and VB6 is that KBasic is a true object-oriented programming language with the possibility to define classes, to inherit from them, and to create objects from classes. KBasic contains all syntax elements of VB6 and supports controls and other components similarly to VB6. Visual Basic 6 is fully supported as its syntax is fully supported. Furthermore, KBasic provides additional object oriented features and extensions.

A KBasic program consists of one or more classes, modules, forms or only functions or subs. As in VB6 it is possible to have a global namespace, global functions or subs, and global variables. KBasic does not support a pre-processor, but this is planned for the future. (e.g. '#ifdef' and '#if'). Theoretically it is not needed because KBasic is a platform-independent programming language and therefore there does not have platform-related dependence. Furthermore, the simple data types are the same except their size might change. Changed sizes are: 'Boolean' is 1 Byte, 'Integer' is 32bit and 'Long' is 64bit in KBasic.

Can I import existing Visual Basic 6 programs automatically?

KBasic will include a VB6-converter in the next major release of KBasic. Meanwhile you can manually copy your files and open them in KBasic.

On which platform is KBasic available?

At this time, there exists versions of KBasic for Linux, Mac OS X, and Windows.

Where can I find more information about KBasic?

Examples, documentation, and the KBasic web site. First read this manual. You can also get news and new

information on www.kbasic.com. Many programming examples are posted on the Internet. Visual Basic 6 websites or BASIC archives list useful program examples that can be used inside KBasic with only minor changes.

KBasic-Syntax

The syntax of sub, function, or statement in the KBasic help entry shows all elements needed to correctly use the sub, function, or statement. Example: Syntax of the MsgBox-Function

MsgBox(prompt[, buttons] [, title] [, helpfile, context])

Arguments listed inside brackets [ ] are optional. (Do not write these [ ] in your KBasic code). The only argument, what you have to give the MsgBox-Function, is the one for the showing the text: 'prompt.' Arguments for functions or subs can be used with the help of their position or their name. In order to use the arguments defined by their position, you cannot ignore the position written in the syntax. You must write them exactly in the same order they occur in the syntax. All arguments must be separated by a comma. Example:

```
MsgBox("The answer is right!", 0, "window with answer")
```

To use an argument with its name, use the name of the argument and colon and equals sign := and the value of the argument. You can write these named arguments in any order you wish. Example:

```
MsgBox(title:="window with answer", prompt:="The answer is right!")
```

Some arguments are written inside {} in the syntax of functions or subs.

Option Compare {Binary | Text}

In the syntax of the 'Option Compare'-statement: { } together with | means that one of the elements must be written.(Do not write these { } in your KBasic code). The following statement defines that text will be compared and sorted while ignoring case.

```
Option Compare Text
```

Syntax of the 'Dim'-Statement

Dim VarName[([Indexes])] [As Type] [, VarName[([Indexes])] [As Type]] …

Dim VarName[([Indexes])] [As Type] [, VarName[([Indexes])] [As Type]] … 'Dim' is a keyword in the syntax of the 'Dim'-Statement. The only needed element is VarName (the name of the variable). The following statement creates three variables: myVar, nextVar and thirdVar. These variables are declared as 'Variant'-variables automatically (or 'Double' in 'VeryOldBasic Mode').

```
Dim myVar, nextVar, thirdVar
```

The following example declares a variable of type 'String'. If you have declared the data type of the variable explicitly, it will help KBasic to optimize the RAM-usage and will help you to find errors in your code.

```
Dim myAns As String
```

If you want to declare many variables in one line, you should declare every data type of each variable explicitly. Variables without declared data type get the default data type, which is 'Variant'. Dim x As Integer, y As Integer, z As Integer

X and y get in the data type 'Variant' in the following statement. Only z has the 'Integer' data type.

```
Dim x, y, z As Integer
```

You have to put ( ) or [] (modern style), if you want to declare an array variable. The indexes of the array are optional. The following statement declares a dynamic array named myArray.

```
Dim myArray(100)
Dim myArrayModernStyle[200]
```

KBasic is not one programming language, but three

Using one of the following commands you can switch KBasic modes:

If you want to use KBasic's newest features (default), use

- Option KBasic

If you want to use old VB6 code, use

- Option OldBasic

For very old BASIC code, like QBasic, use:

- Option VeryOldBasic

It is possible to use all three modes in your programs, e.g. one module uses one mode while a second module uses another mode. Just place one of these lines in top of your module. Default is 'Option KBasic.'

Software development with KBasic

A typical KBasic application consists of forms, modules, classes, and other objects, which together are your application. Forms, their controls, changing values in controls, or having the user click on buttons generates events. You can react by assigning KBasic-code to such events.

Event controlled programming vs. traditional programming

When a traditional, procedure-built program runs, the application controls the executable parts of the program, ignoring the events. The application starts with the first code of line and goes through its source codes as the developer has defined. If needed some procedures are called.

In event controlled applications, user action or a system event triggers the execution of the event procedure. The order in which the code is executed depends upon the order of the events, which occur based upon user actions. This is the principle of graphical user interfaces and event controlled programming. The user performs an action and your program reacts.

How does an event controlled application run?

An event is an action recognized by your forms or controls. Objects in KBasic know some predefined events and react to them automatically. To get a control to react to an event, write an event procedure for the event.

A typical application runs as follows:

- 1.The user starts the application.
- 2.The form or form control receives an event. The event can be triggered by a user-action (e.g. key pressed) or by your code (e.g. event 'Open' if your code opens a form).
- 3.If there is a event-procedure for the event, the desired event code executes
- 4.The application waits for the next event

Some events automatically raise other events. More information is provided in the help of KBasic Software IDE.

Three Programming Steps

To create a simple KBasic program, you need only complete three steps, after which you will have a program that can be run outside of KBasic's programming environment just like any other application you may have installed on

your computer. The three steps are as follows:

- 1.Create the program's user interface
- 2.Write the program source code, which makes the program do what it is supposed to do
- 3.Compile the program into an executable file to run as a standalone application (able to run without being loaded into KBasic – only Professional Version of KBasic). This is done by the KBasic Software Atelier Binary Compiler.

Of course, there are many details involved in each of these three steps, especially if you are writing a lengthy program. As you work, complete these steps in the order listed above. You should frequently alternate between steps 2 and 1 to fine-tune your user interface. You might even return to step 2 from step 3 as you discover problems after compiling your program.

Object oriented programming with KBasic

Object oriented programming is one of the most important concepts of recent years and has become a commonly used technique. Object oriented programming covers issues like: What are classes and objects and how they relate to each other The behavior and attributes of classes and objects Inheritance of classes and how the design of programs are affected.

Objects and classes

KBasic is an object-oriented programming language. Object-oriented, in KBasic terms, contains the following ideas:

- Classes and objects in KBasic
- Creating objects
- Garbage collection to release unused objects
- The difference between class-variables and instance-variables and the difference between class-methods and instance-methods
- The extension of classes in order to create a sub-class (child-class)
- The override of class methods and polymorphism when it comes to calling methods
- Abstract classes

If you are a Java-developer or a experienced developer in another object-oriented programming language, you will know many of the concepts of KBasic.

Object-oriented programming means every object is part of another object. For example, in the real world, cars consist of many objects like windows, steering wheels, and so on.

There is only one class (building plan) for a window or steering wheel. Therefore there exists a class window and a class steering wheel, but many objects window or steering wheel, which are the instances of a class. An instance of a class is also called an object of a class. Classes exist when you write your code. Objects are created when KBasic runs your code using your classes!

Every class has the following elements:

- 1. Parent-class
- 2. Attributes (variables, constants…)
- 3. Methods

Parent-class means all attributes and methods of the parent-class apply to the new class in addition to attributes and methods you declare in the new class.

Attributes can be properties, constants, or variables pertaining to a class.

Methods are procedures and functions of a class. Methods are arguably the most important part of any object-oriented programming language. Whereas classes and objects provide the framework and class and instance variables provide a way of holding that class' or object's attributes, the methods actually provide an object's behavior and define how the object interacts with other objects in the system.

Variables and methods can be related to a class (one time in memory) or can be instance-related (as many as

instances of that class exist) in the declaration of a class (outside a method):

- Instance-variable
- Dim instanceVar
- Instance-method
- Sub instanceSub()
- Class-variable
- Static staticVar
- Class-method
- Static Sub staticSub()

Class-method or class-variable can be used without any created object. Instance-method or instance-variable can only be used together with an object. Instance-variables are like local variables of a procedure. Local variables only exist after calling the procedure, instance-variables only exist within the lifetime of an object. Class-variables are global and available at any time. Class variables are good for communicating between different objects with the same class or for tracking global states among a set of objects.

Classes can contain constants as well, like declaring variables. Constants are declared in the declaration area of a class.

Classes can give all its elements to its child, which is called inheritance.

Inheritance of classes

Inheritance is an important part of object-oriented programming. KBasic, like Java, supports single inheritance, meaning every class inherits one parent class. It is not possible to inherit from many classes. In the far future, it will be possible to use an interface to do many things you could do with multi-inheritance. As in Java, all objects in KBasic are created with 'New.' A class is a collection of data and methods working on those data. Data and methods describe the content and the behavior of an object.

Objects are instances of a class. You cannot do much with a single class but can do much more with an instance of a class, a single object containing variables and methods. Object-oriented means the objects are the centerpiece, not procedures and variables.

KBasic-Syntax

To read and write programs quickly you must know the syntax of KBasic. The syntax defines how to write programs. It defines how to write each language element, how they work together, and how to use them. The following pages cover these language elements.

Statements and expressions

Statements are a complete action in KBasic. Statements can contain keywords, operators, variables, constants, and expressions. Every statement can be categorized by one of the follwoing:

Statements as declaration or definition for class, module, variable, constant, procedure, function, method, property Assignment statements, assigning a value or expression to a variable or constant, or property Executable statements performing an action. These statements can execute a method, function, procedure, loop, or condition. Executable statements often contain conditional or mathematical operators (even complex expressions).

Multi-Line statements

Normally every statement fits into one line, but sometimes it's more readable to split one statement into several lines. If so, use the line continuation underscore (_). In the following example the (_) is used:

```
Sub field()
          Dim myVar As String
          myVar = "Bernd"
          MsgBox Prompt:="Hello " & myVar, _
                    Title:="Hello"
End Sub
```

You can write many statements in one line using a colon (:) to separate the statements. For example: Print „Hi": Print „Ho"

Variables and DataTypes

While computing expressions you may need to store values for a short time. For example, you would like to calculate different values, compare these values, and depending on those values, perform different operations. In order to compare values you need to store them. Storing does not save the values on disk but saves them in memory because you need the values only when running your program.

KBasic uses variables to store values at runtime of your program. After stopping your program, all variables (even objects) are deleted. A variable can change its value at any time. You can change its value by assigning an expression, another variable, or a constant to it. A variable is a space in memory used by KBasic to store values so variables are not stored in a file. Like a file, a variable has a name (used to access the variable and to identify it) and also a data type to specify the type of information a variable holds.

KBasic knows instance-variables, class-variables, global variables, and local variables.

Instance-variables are part of an object, class-variables are part of a class, global variables can be used everywhere, and local variables are part of a procedure, function, or sub (also called method in classes).

Declaration of Variables / Explicit declaration

Before using variables, you must declare them. You must define the name and the data type of a variable. The 'Dim'-statement declares a variable. 'Dim'-statements are one of many variable declaration statements, which slightly differs from the others. Other declaration statements are 'Redim,' 'Static,' 'Public,' 'Private,' 'Protected,' and 'Const'.

```
Dim myName As String
```

Variables can be declared everywhere in procedures, functions, and subs, not only at the top of procedures, functions, and subs. However, most or the time they are at the top.

```
Sub doEvent()
  Dim myName As Integer
End Sub
```

Every line of your source code can contain many declaration statements for many variables. Which data type the variables have depends upon the way you declare the variables. If you give every variable an explicit data type specifier, the default data type for variables will not be used. The default data type depends upon the mode chosen for your program. In 'KBasic Mode' and 'OldBasic Mode' the default data type is 'Variant,' in 'VeryOldBasic' it is 'Double.'

```
Dim firstname, sirname, lastname As String
```

name is type 'Variant' sirname is type 'Variant' lastname is type 'String'

Or explicitly with different types:

```
Dim myName As String, age As Integer
```

To use a data type you have to write it for every variable, otherwise the default data type would be used. In the following statement all variables are declared as 'Integer.'

```
Dim intX As Integer, intY As Integer, intZ As Integer
```

In the following statement intX and intY are declared as 'Variant' and intZ as 'Integer.'

```
Dim intX, intY, intZ As Integer
```

Variables can have a start value for initial. In order to do so, you have two ways:

Either

```
Dim name = "sunny sun" As String
```

or

```
Dim name As String = "sunny sun"
```

If many variables are declared in one line, only lastname has the value "sunny sun."

```
Dim name, sirname, lastname = "sunny sun" As String
```

If no value is assigned by declaration, the default value of 0 is used if it is a numeric variable. If it is a string, a nullstring is used with length 0 (""). Object variables are set to 'Null.' Variant is 'Empty.'

Names of variables must not be longer than 128 characters and must not use all characters possible. Do not use periods, commas and other non-writable characters. Only the underscore (_) is allowed.

Important! Normally, KBasic does not differentiate between a variable name written in lowercase or uppercase. In fact you can write it differently in every line. However, to be compatible with future versions of KBasic, always write your variable names consistently. Some exceptions exist. If you use the bindings, you always write the name of methods the way it is mentioned in the documentation. So Named, named, naMED are the same variable.

Use the following rules to name procedures, functions, subs, constants, variables, or arguments in KBasic:

- Use a letter (A-Z, a- z) or underscore (_) as your first character
- Never use whitespace ( ), period (.), exclamation mark (!), or the following characters : @, &, $, #, „ in your names.
- The name may contain numbers but must not start with a number
- Do not use names already in use by predefined elements of the KBasic language, such as keywords (e.g. 'If'), built-in functions (e.g. 'Print'), classes etc. This would lead to a name collision in which the original language element would be expected. Your program will not compile.
- You cannot use variables with the same name in the same scope. e.g. you cannot declare variable 'age' two times in a procedure, but you can declare 'age' in a procedure and in a module at the same time because they are declared in different scopes (at different places).

Declarations of variables in different scopes

You can place a declaration statement inside a procedure, function, sub, or method to declare a variable in local scope. Additionally, you can place a declaration to declare global-variable, class-variable, instance-variable in the declaration section of a class or module. The place of the declaration determines the scope of the variable. The scope of a variable cannot change at runtime of your program, but you can have different variables with the same name at different places. For example, if you have the declaration of a variable named 'price' in a procedure, and the same declaration in a module, all uses of this variable in the procedure are related to the local variable 'price,' all uses outside the procedure are related to the global variable 'price.'

A variable named sName with type of 'String' is declared in the following example:

```
Dim sName As String
```

If this statement is part of a procedure, then it is possible to use this variable in the same procedure only. Is this statement part of a module declaration section, you can use it in all procedures of the module, but not outside the module (in fact it is declared as 'Private' implicitly). To be able to use it everywhere, you can use the keyword 'Public'. Example:

```
Public sName As String
```

Use of the 'Public'-Statement: You can use the 'Public'-statement to declare public variables in module scope or class

scope, making the variable accessible from everywhere.

```
Public sName As String
```

Use of the 'Private'-Statement: Use the 'Private'-statement to declare private variables in module scope or class scope, making the variable accessible only from the same scope (module scope, all module procedures, class scope, all class methods).

Private myName As String

If the 'Dim'-statement is used in module scope or class scope, it is treated as a 'Private'-statement. Use the 'Private'-Statement to have cleaner, more readable code.

Use of the 'Protected'-Statement: Use the 'Protected'-statement to declare protected variables in class scope, making the variable accessible from within the same scope (class scope, all class methods, sub-classes, all sub-classes methods). This allows you to underline the inheritance hierarchy of your classes.

```
Protected myName As String
```

Use of the 'Static'-Statement: Static has three different meanings, depending upon the context used.

- Static inside a class, but outside a method. If you use a 'Static'-statement instead of a 'Dim'-statement, the variable is declared as class variable, meaning it can be used without an instance (object) of this class. A class-variable exists only one time at runtime.
- Static outside a class, but inside a procedure (sub or function) or method. If you use a 'Static'-statement instead of a 'Dim'-statement, the variable is declared as local static variable. The variable, once it has been declared, is not destroyed by leaving the procedure. The next time the procedure is entered, the value of the variable still exists. Therefore, a local static variable is only one time declared when using recursive calls of a procedure.
- Static outside a class, but before a procedure (sub or function) or method. If you use a 'Static'-statement before the keyword 'Sub' or 'Function' all local declared variables are declared as 'Static' variables, which means that a variable, once it has been declared, it is not destroyed after leaving the procedure. The next time the procedure is entered, the value of the variable still exists. Therefore, a local static variable is only one-time declared when using recursive calls of a procedure.

You can add the keyword 'Static' to other keywords ('Public', 'Protected', 'Private').

Declaration of variables automatically / implicit declaration

For historical reasons it is possible to use variables without declaration. It is supported within a special mode in which all variables are declared automatically when using a variable name that has not been used before. Important! You should not use this mode unless you want to use old BASIC code without changing it. Good programming style dictates declaring all variables with their types. This also makes it easier for others to understand your code and minimizes typing errors.

In order to activate implicit declaration, write the following line in top of your program:

```
Option OldBasic
Option Explicit Off
```

Use of the 'Option Explicit'-statement (supported within 'Mode OldBasic' and 'Mode VeryOldBasic' only) As mentioned previously, you can implicitly declare a variable in KBasic by using an assignment statement or expression with the variable name. All implicitly declared variables have the data type 'Variant' ('Option OldBasic') or 'Double' ('Option VeryOldBasic'). Variables of type 'Variant' need more space in memory than most of the other data types. Your program is faster when you explicitly declare your variables with the smallest data type possible. Another advantage is that using declaration avoids name collisions or typing errors.

To explicitly declare variables within 'Option OldBasic' or 'Option VeryOldBasic,' write on top of all module and class files 'Option Explicit On.' This creates a compilation error when a variable name is not declared. If you are using the 'KBasic Mode' you must declare all variables. It is like 'Option Explicit On.' By the way, the 'KBasic Mode' is set to default in all programs.

Important! You must declare dynamic arrays or fixed arrays at all times. Furthermore, you can mix the modes in your program. One file is set 'Option OldBasic,' another file is set 'Option VeryOldBasic.' You are free to set 'Option Explicit Off' for the old modes.

Local variables

The value of a local variable is available inside the procedure only. You cannot access the value from outside the procedure. This makes it possible to have the same variable name for different variables in different procedures without name collision.

```
Example:

Sub test1()
  Dim i As Integer

  i = 1
End Sub

Sub test2()
  Dim i As Integer

  i = 19
End Sub
```

Assignment-statement

Assignment-statements give or assign a variable a value or expression with variables, constants, numbers, etc. An assignment always includes a sign (=). The following example shows an assignment.

```
Dim yourName As String
yourName = InputBox("What is your name?")
MsgBox "Your name is " & yourName
```

You do not need the 'Let' keyword. It is outdated and optional. An example shows how to 'Let':

```
Let yourName = InputBox("What is your name?")
```

Use the 'Set' statement VB6 to assign an object to an object-variable. 'Set' is not needed in KBasic and must not be used. After the declaration of a variable, you can use it, e.g. make an assignment (=).

```
Dim Cool As Boolean
Dim myName As String

Cool = True
Name = "Julie"
```

Cool contains 'True', name contains 'Julie' after the assignment.

Lifetime of variables

The lifetime of variables is the time in which the variable exists or has a value. The value of a variable can be changed during its lifetime. If a variable is outside its scope, it has no value. If a procedure is executed, a variable exists after its 'Dim'-statement. Local variables exists until the procedure is completed; after reentering the procedure all variables are created again after their respective 'Dim'-statements.

If you do not change the value of a variable during its lifetime in a program, it contains the initialized value at program start. A variable declared in a sub contains an assigned value until the sub is left. If the same sub is called again, the value is reset to the initialized value.

When using static-declared variables in procedures, the variables only exist once in memory, unless the procedure is called multiple times using recursion. If you use global-declared module variables or class-variables, it is the same. The variables exist once in memory at runtime. If you use instance-variables you must consider that they only exist together with the instance (object) to which they belong.

Place of declaration

Declare every variable and every procedure inside a class or at least inside a module.

```
Syntax:

Dim Name[([Index])] [As Type] [, Name[([Index])] [As Type]] ...

Dim Name [= Expression] [As Type]

Dim Name [As Type] [= Expression]

[Public | Protected | Private | Dim | Static] Name [= Expression] [As Type]
```

Data Types

Data types describe the type of data stored inside a variable. If you declare a variable you must also give it a data type. KBasic supports all VB6 data types and much more. You can use one of the following data types:

- One of the simple data types (e.g. 'Integer')
- Name of built-in class (KBasic class, binding-class)
- User defined class
- User defined data type
- User defined enumeration

Simple Data Types

Simple data types store numbers and text. Simple data types are simple because they are built in within KBasic and are not complex like classes. Every simple data type has borders determining the size of the stored number. If a value is too big it loses precision, meaning it loses information! The following table contains all KBasic supported datatypes with their needed space in memory.

- Data Type / Size / Value
- Boolean / 1 / 'True' or 'False' - stored as 8-bit numbers (1 Byte) and can only be 'True' or 'False',
- Byte / 1 / 0..255 - stored as 8-bit numbers (1 Byte) without a sign and can be >= 0 and F 255
- Short / 2 / -2^15 to 2^15-1 - stored as 16-bit numbers (2 Bytes). Values can be greater than or equal to -32.768 and less then or equal to 32.767.
- Integer / 4 / -2^31 to 2^31 –1 - stored as 32-bit numbers (4 Bytes). Values can be -2^31 >= and F +2^32. You can use integer variables to simulate enumerations, such as 0 = black, 1 = white and so on. You should not use it for enumerations like in VB6, better use the new approach with the 'Enum'-keyword. Type suffix is (%).
- Long / 8 /-2^63 to 2^63 –1 - stored as 64-bit numbers (8 Bytes). Values can be -2^63 >= and F +2^64. Type suffix is (&).
- String / size depending on string length / unlimited - There are two types of strings, strings with fixed length and strongs with dynamic length. Dynamic strings can be up to 2^31 characters long. Normally, strings or string arguments are dynamic strings. The dynamic string changes size when assigning new values. Fixed strings are used together with 'Type'-keyword, only. Type suffix is ($).
- String / fixed length / length of string - Fixed strings can be up to 2^31 characters long. Fixed strings are used together with 'Type'-keyword, only. Type suffix is ($). myStringVar = "1234 Names symbols .!/& keywords IF ? - "
- Character / 2 / - reserved for future.
- Single / 4 / - stored as floating numbers (4 Bytes) with single precision. Values can be between -3.402823^38 or -1.401298^-45 or between 1.401298^-45 until 3.402823^38. Type suffix is (!).
- Double / 8 / - they are stored as floating numbers (8 Bytes) with double precision. Values can be between -1.79769313486232^308 bis -4.94065645841247^-324 or between 4.94065645841247^-324 till 1.79769313486232^308

Type suffix is (#).

- Object / 4 / + size of the object - stored as 32-bit references (4 Bytes).
- DateTime or Date (Date/Time) / 8 - stored as numbers (8 Bytes). A date literal must be #yyyy-mm-dd# . A time literal must be #hh:mm:ss# .
- Decimal or Currency / 8 - stored as numbers (8 Bytes). Values can be between -922,337,203,685,477.5808 or

922,337,203,685,477.5807. Type suffix is (@).

- Variant / 40 - A 'Variant' variable is always 40 bytes big.
- Classes / - see Object
- User defined data type / see next chapter - size depends on its elements

User defined types

User defined types have been heavily used in the 1980s and 1990s, but today they step behind ordinary objects. User defined types are a group of variables with different types stored in one variable. These types can be simple data types or even user defined types.

```
Syntax:

Type Name
 Name [(Index)] As Type
  ...
End Type
```

Class types/Objects

Variables can store objects (actually, references to objects). These variables are object variables of data type 'Object' and are 4 bytes. Unlike in VB6, assignment of objects to an object variable do not need the 'Set'-keyword. The '='-statement is recognized by KBasic correctly.

Though an object variable can contain any object (actually, the reference to that object), the binding is done at runtime (late binding). If you would like to have early binding, use a class that inherits the object class, like TextBox. Force early binding using a cast statement, too.

Type Variant

The data type 'Variant' is automatically used if you do not specifiy a data type for an argument, constant, procedure or variable. An exception, is using a variable in 'VeryOldBasic'-Mode. It has no 'Variant' but 'Double'-data type. Variables of type 'Variant' can contain strings, dates, time values, boolean values or even numerical values. The size of 'Variant' is 40 bytes. Accessing variant data types is slower. The following example creates two variables of type 'Variant':

```
hisVar = 3

Dim myVar

Dim yourVar As Variant
```

The first statement implicitly declares a variable (automatically).

Additionally, a 'Variant' can store the following values:

- 'Null'
- 'Empty' (old style)

Suffix of variables

Names of variables can determine their data types. Historically, you can append a special symbol showing the variable's data type. KBasic supports this old style of programming for historically reasons. Example:

```
Dim i% ' Integer variable
Dim l& ' Long variable
Dim s$ ' String variable
```

The following type symbols are supported.

- Integer - type sign for 'Integer' is (%).
- Long - type sign for 'Long' is (&).

- String - type sign for 'String' is ($).
- Single - type sign for 'Single' is (!).
- Double - type sign for 'Double' is (#).
- Decimal (Currency) - type sign for 'Currency' is (@).

Comments

The comment symbol (') is used in many code lines in this manual. Comments can explain a procedure or a statement. KBasic ignores all comments while compiling and running your program. To write a comment, use the symbol (') followed by the text of the comment. Comments are printed in green on screen. KBasic recognizes four ways to write comments, as shown below.

```
REM this is a comment

' this is a comment, too


and like in Java


/* comment begin and comment end */

/** docu comment begin and docu comment end */
```

Comments are extremely helpful when it comes to explaining your code to other programmers. So comments, normally, describe how your program works.

Way of naming

When coding in KBasic, declare and name elements, like procedures (functions and subs), variables, and constants and so on. All names

- must start with a letter;
- must contain letters, numbers, or the sign (_); periods and commas are forbidden;
- must not be longer than a defined length; and
- must not contain reserved words

A reserved word is a part of KBasic and has a predefined meaning. These include keywords (e.g. 'if' or 'then'), built-in functions (e.g. 'mid') and operators (e.g. 'mod'). A complete list of reserved words is at the end of this manual.

Literals

Besides keywords, symbols, and names, a KBasic program contains literals. A literal is a number or string representing a value. There are different numerical literals.

- Byte, Short, Integer, Long - 1, 2, -44, 4453
- Hex - &HAA43
- Binary - &B11110001
- Octal - &O1234
- Single (Decimal), always English formatted - 21.32, 0.344, -435.235421.21
- Double (Decimal), always English formatted - 212.23
- Currency, always English formatted - 45.3@
- Date / Time, always English formatted - #1993-12-31#
- String - Is simply text, but it must start with a (") and end with a (") so that KBasic can recognize it as string. A double (") inside a string is interpreted as single ("). There are no escape sequences like in C++. They can be used with built-in functions.
- String (multi-line) - Is simply text, but it must start with a (<?) and end with a (?>) so that KBasic can recognize it as string.
- Boolean - When you cast a numerical value to Boolean. Values with 0 are 'False'. Values with -1 are 'True'.

Expressions

Expressions stand for values. They can contain keywords, operator, variables, constants or even other expressions. Examples for expressions are:

- 1 + 9 (number operator number)
- myVar (variable)
- myVar TypeOf Object (variable keyword classname)

Expressions can return a value or not. If an expressions is to the right of an assignment statement (=), the expression gives or must give a value back.

```
myVar = 1 + 9
```

1 + 9 is 10 and this expression value (10) is stored in myVar. It is exactly the same when the expression is used as a parameter in a function call.

```
MyProcedure(1 + 9)
```

Expressions are calculated at runtime and represents a value. The result can be assigned to a variable or to other expressions listed above.

Constants

Constants are similar to variables but they cannot change values. When you declare a constant you assign a value to it that cannot be altered during lifetime of your program. Example:

```
Const border As Integer = 377
```

This 'Const'-Statement declares a constant border with data type 'Integer' and a value of 377. Use the same rules for declaring constants as for variables. A 'Const'-statement has the same scope as the variable statement. To declare a constant inside a procedure, place the 'Const'-statement inside this procedure (normally one of the first statements). To declare a constant as accessible for all procedures, place it within the module (or class) where the procedures are.

This 'Const'-Statement declares a constant conAge as public with data type 'Integer' and a value of 34.

```
Public Const conAge As Integer = 34
```

Constants can contain all simple data types: Boolean, Byte, Short, Integer, Long, Currency, Single, Double, Date, String or Variant. You can also declare many constants in one line.

```
Const conAge As Integer = 39, conSalary As Currency = 3500
```

This 'Const'-Statement declares constant conAlter and conSalary. Syntax:

```
Const Name = Expression

Const Name [As Type] = Expression [, Name [As Type] = Expression] ...

[Public | Protected | Private] Const Name [As Type] = Expression
```

Operators and Operands

KBasic supports all VB6 operators. Operators are '+' or '-' for example. Operands are numbers, strings, or variables (or expressions). KBasic supports operator overloading for the bindings. In the future, it will be possible to use operator overloading inside KBasic classes like it is possible in C++.

There are different kind of operators.

Operators for calculating

They are used for calculating of two operands. Each of those operators needs two operators. The minus (-) can be used to negate values, like -9 means negative 9, +9 means positive 9, 9 means positive 9

Calculating operators are:

- + Addition – Addition adds two numbers or strings. If you add a number and a str`ing KBasic cannot be sure if the * result should be a number or string. In this case, use the '&' Operator instead of '+' for strings. e.g. 3 + 4
- - Subtraction – Subtraction subtracts a number from another number. e.g. 5 – 7
- * Multiplication - Use to multiply two numbers e.g. 33 * 7
- / Division - Use to divide two numbers. The result is floating point. e.g. 2 / 5
- \ Integer-Division - The result has data type 'Integer'. e.g. 29 / 5
- Mod Modulus, also known as remainder of Integer-Division – Returns the remainder of a result of an Integer-Division e.g. 45 Mod 10

Integer-Division results in an 'Integer' value. Example: x is 10, y is 4

```
Print x + y ' is 14
Print x – y ' is 6
Print x * y ' is 40
Print x / y ' is 2.5
Print x \ y ' is 2
Print x Mod y ' is 5
```

Normally, the result has the data type of the operand (= expression) with more precision. Assignment-operators

At this time, KBasic supports more than the normal assignment operator (=). Like C++, KBasic supports the extended assignment operators.

- += Add-Assignment (e.g. var += 1 is like var = var + 1)
- -= Subtract-Assignment (e.g . var -= 1 is like var = var - 1)
- /= Divide-Assignment (e.g . var /= 1 is like var = var / 1)
- *= Multiply-Assignment (e.g . var *= 1 is like var = var * 1)

If you use the bindings, you can additionally use further C/C++-assignment operators like |= Or-Assignment. See binding documentation for more details.

Increment and Decrement

Currently, C++ operators, '++' and '- -' are supported for the bindings. To increment or decrement, use var = var + 1 or var = var – 1. You may also use INC(var) (increment, +1) or DEC(var) (decrement, -1) in the future.

Comparision

KBasic has different operators to compare expressions: These operators compare between two operands (expressions) and the result is 'True' or 'False':

- = Equal - e.g. x = 3
- <> Unequal - e.g. x <> y
- < Smaller - e.g. x < 3
- > Bigger - e.g. x > 4
- F Smaller or equal - e.g. x F 4
- >= Bigger or equal - e.g. x >= 3

If you use the bindings, you can use further C/C++-comparision operators like ==-Assignment. See binding documentation for more details.

If you use the bindings, you can additionally use further C/C++-comparision operators like ==-Assignment. See Binding documentation for more details.

Logical operators (Boolean Operators)

Use logical operators for performing bit operations or combining bits together. The result is ‘True’ or ‘False’. Unlike VB6, KBasic supports true logical operators for decisions (‘AndAlso’, ‘OrElse’).

- AndAlso - Both operands (expressions) must be ‘True’
- OrElse - One of the operands (expressions) must be ‘True’
- You can also use bit-operators instead of ‘AndAlso’ or ‘OrElse’.
- Bit Operators
- And and - e.g. 4 And 6 - Useful for doing logical conjunctions of two expressions. The result is ‘True’ if both expressions are ‘True.'
- Or or - e.g. 33 Or 8 - Useful for doing logical disjunctions of two expressions. The result is ‘True’ if one of both expressions is ‘True.'
- Xor exclusive or - e.g. 77 Xor 3 - The result is ‘False’ if both expressions are ‘True.' Or the result is ‘True’ if one of either expressions is ‘True.'
- Not not - e.g. Not 5 -Useful if you would like to negate expressions. Furthermore, ‘Not’ inverts a bit of byte. BitAnd, BitOr, BitXor, BitNot are the same as ‘And’, ‘Or’, ‘Xor’ and ‘Not’ and are supported for compatibility reasons.

A result is -1 (‘True’) when all bits are set (binary 1111 1111) or 0 when all bits are not set (binary 0000 0000). When using the bindings, you can use further C/C++-shift operators like « or ». See binding documentation for more details. You can use SHL(number of digits) or use SHR(number of digits) in the future.

Other Operators

- ^ Power - Useful if you want to use power. If they are many powers in one expressions they are processed from left to right.
- Like - Useful to compare two strings (using a pattern). If string and pattern match, the result is ‘True’.
- New - creates a new object using a class
- ( ) or [ ] - Accesses indexes of arrays or argument holders for procedures
- . dot operator - Needed for calling methods or elements in types or enumerations
- & String concatenating - Appends one string to another
- Eqv - means x Eqv y = Not (x Xor y)
- Imp - means x Imp y = Not (x And Not y)
- Is – Determines if two objects are based on the same class.

If you use the bindings, you can use the following C/C++-operators. See binding documentation for more details.

- ++ increment
- – decrement
- [ ] index access
- |= or-assignment
- &= and-assignment
- | or
- ! not
- == is equal
- != unequal
- ^= power-assignment
- « shift left
- » shift right

String concatenating

It is possible to add two strings, or append one string to another. The result is data type ‘String’. To append, there exist two operators: '+' and '&'. The '+' works incorrectly when one operand is a different type than ‘String’. To be sure of the correct types, use '&' for string concatenating.

- '+'
- '&'

Example: Name + "is a" + color + "bird" If the first operand is a string, using '+' forces all other operands to be treated as strings.

Operator order

KBasic supports die operator order of VB6. Normally, an expression is executed from left to right following standard mathematical rules.

e.g: x = 10 + 10 / 2 results in 15 and not 10, because / is calculated before +. Here is the overview about operator order/priority from top to bottom, which means '*' is executed before 'And.' The bindings operators are included in braces ():

- 1. . ( ) [ ]
- 2. Not, BitNot, !, ++, –, (unary +), (unary -)
- 3. New
- 4. ^
- 5. * / \ Mod
- 6. + - &
- 7. Imp Eqv
- 8. < > F >= = Is Like == !=
- 9. And BitAnd (& C/C++ And)
- 10. Or Xor BitOr BitXor (| C/C++ Or) (^ C/C++ Xor)
- 11. AndAlso
- 12. OrElse

Use parenthesis () to change the order. e.g. X = (10 + 10) / 2 results in 10 and not 15, because + is calculated before / thanks to the braces.

Avoiding name collision

A name collision occurs when you use a name that was already defined in KBasic (or even by KBasic itself = keywords). Avoid name collisions by knowing the concept of scopes. There are different types of scopes in KBasic: procedure scope, global scope, class scope, module scope, and the scope modifiers ('Public', 'Private', 'Protected'). You can have name collisions in the following situations:

- If an identifer can be accessed from different scopes
- If an identifier has different meanings in the same scope

Most name collisions can be avoided by using fully qualified names of identifiers. Example:

```
aModule.aSub(Module1.Var1)
```

Editing source code

Editing source code is no different than editing text in a word processor. You have a cursor showing the current position you can type in. The find and replace commands within the editor work just as in your word processor, as well as many other common commands.

Working with objects

Because KBasic is an object-oriented programming language, you probably would like to know how to use these features. The following issues need to be discussed:

- How can I create a class?

- How can I create an object (instance) of a class?
- How can I use class-variables, class-methods, instance-variables and instance-methods?
- How can I convert an object into another object?

Create new objects

Either you create an object based on a built-in class of KBasic, or you create it from your own defined class.

Use of New

A variable, which contains the object and is declared, does not create the object at once. This variable only has the data type of an object and represents only a reference to an object still to be created. Create objects using the special keyword 'New.' This creates an object by giving it the desired class name and returns a reference to the new object. The reference is stored in a variable of the class type. In this way, creating objects in KBasic is the same as for Java or C++. Example:

```
s = New Control()
```

You do not need to use the braces (), but when creating an object you may have arguments that must be written inside braces. 'New' is followed by the class name of the new object. The class has a special procedure called constructor, called by 'New'.

This constructor does some initial work for an object and returns a reference to the newly created object. Unlike regular procedures, you cannot call a constructor by calling it directly. Instead constructors are called by KBasic automatically when you create a new object with the 'New' command. Yet another example:

```
s = New Timer(begin, end)
```

What happens when using 'New'?

Using 'New' creates a new instance (an object) of a class. The matching constructor of that class is executed. When using or declaring constructors, consider two important things first: The name of the constructor matches the name of the class it means. The return value is always an instance of that class (implicitly). There is no return type declaration necessary, nor is the keyword 'Sub' or 'Function' used. A constructor must not explicitly return values using the keyword 'Return'.

Syntax:

```
objectVariable = New ClassName[(Arguments)]

objectVariable = New ClassName()

objectVariable = New ClassName
```

Several Constructors

Sometimes, you may want to use an object differently or perform the initialization differently. You can do so by using different constructors for the same class. Every class can have different constructors with different arguments.

Example:

```
Class test1

  Constructor test1()
  End Constructor

  Constructor test1(i As Integer)
  End Constructor

  Constructor test1(i As Integer, m As String)
  End Constructor

End Class
```

'Null' (or 'Nothing')

The default value for object variables is 'Null.' 'Null' (or 'Nothing') is a reserved word. The object variable does not yet have a reference. 'Null' can only be assigned to object variables.

Releasing objects automatically

When you no longer need a object variable (or object), you do not need to explicitly delete or free the object. KBasic mechanisms do that automatically. This is called garbage collection.

Releasing objects manually

You can also explicitly free or release objects using the 'Null' keyword. Just assign 'Null' to the desired object variable.

```
myVar = Null
```

Objects that are no longer referenced are automatically deleted by KBasic. This happens when no object variable refers to the object because the object variable has been deleted or the value has been explicitly set to 'Null.' For example:

```
objectVar = Null
objectVar = Nothing ' old style
```

You do not have to release objects, you can just forget about them. This is much different from C++ and other programming languages.

Create a class

Normally every class is a child of the special class 'Object' directly or indirectly through another class. A class can inherit from another class (be a child of that class). The declaration of a class consists of a class name and the parent class name. Furthermore, you have constructors, sometimes destructors, some variables (class-variables and instance-variables), and procedures (methods, class-methods, instance-methods) and, of course, some properties and constants.

```
Class oak Inherits tree

Parts are:
Variables/ Constants / Properties / Types / Enumerations
Constructors
Destructors
Functions
Subs

End Class
```

A class can only be declared inside a module-file (not recommended) or inside a class file. To declare a class inside a form-module-file is not possible, though it actually contains a class that inherits from class 'Form' implicitly. It is possible to declare many classes in one class-file.

Classes are not executed, but methods are

A newly created class only contains the declaration part (class name and parent name), but no methods. These must be created by the programmer. KBasic does not run any class. It runs the methods inside the class.

Accessing objects

Accessing objects is done by using the dot operator (.).

Example:

```
className.variable
```

This syntax is quite the same as accessing elements of user defined types (‘Type’ … ‘End Type’).

Accessing class-variables and instance-variables

Accessing variables is done by using the dot operator (.). Example:

```
myObject.variable
```

When accessing class-variables, use the class name. When accessing instance-variables, use the name of the object (instance) variable:

```
className.classVariable
objectName.instanceVariable
```

Important! Class-variables only exist one time, so it is not important how many objects of that class exists. The class itself exists only one time, so do the class variables. You can use the class-variables without having created an object of that class beforehand. The class is available while running your program at any time. It is like a global variable

However, instance-variables are part of an object of a class, so instance-variables exist whenever objects exists of that class and only during the lifetime of the objects. If you declare a variable inside a class, you can mark a variable with the keyword ‘Static’ if it should be a class-variable. If you do not, it is by default an instance-variable. Example:

```
Static myClassVariable As Integer
Private myInstanceVariable As String
```

Another example:

```
myClass.classVariable = 23

Dim o As myClass
```

Accessing inside a method:

```
o.instanceVariable = 99
```

Class-methods and instance-methods

Like variables, methods can also be class related or object related. Class-methods are always present, instance-methods are only present when its object is present. Class-methods are like global functions. You cannot use ‘Me’ or ‘Parent’ inside class-methods because there is no object.

When you define a method, use the keyword ‘Static’ to define a class-method. If you leave it out, the method is by default an instance-method.

Example:

```
Static Sub myClassMethod()
...
End Sub

Sub myInstanceMethod
...
End Sub
```

Calling methods

Again, use the dot opertator (.) to access the methods.

```
myObject.myMethod()
```

References to objects

Using the ' = '-Assignment you can assign a reference of an object variable to another object variable. Both variables then reference the same object.

Example:

```
Dim i As tree
Dim s As tree

i = New tree
s = i
```

Both variables then reference the same object. Important! 'Set' known from VB6 is not allowed and not needed to assign objects in KBasic, because it is obsolete.

Copying objects

Because objects are not passed by values in procedures, an assignment of an object does not copy an object. Instead you must provide proper methods in the classes that can copy an object. In the future, KBasic will provide objects with a clone method to copy objects.

Comparision of objects

IMPORTANT! The ' = '-comparison operator tests if two object variables reference the same object,not if both variables objects have the same content. Instead you have to provide proper methods in the classes to compare objects. In the future, KBasic will provide objects with a compare method that will be able to compare objects. This will be the equals method.

Summary: Objects, properties, methods and events

An object is an element of an application, such as a table, a form, or report. To use such an element, use the collections provided by KBasic. A collection contains different objects of the same object type, i.e. - a collection for forms contains all forms of your application. Elements of a collection can be accessed by an index or name.

Accessing by index:

```
Sub myClose()
          Forms(1).Close
End Sub
```

Accessing by name:

```
Sub closeForm()
          Forms("myForm2").Close

End Sub
```

If the collection contains methods usable for all elements of that collection, you can access it using the collection name and the desired method.

```
Sub closeAll()
          Forms.Close
End Sub
```

A method is part of an object and contains executable code, which can be called at any time.

```
Sub addNewEntry(newEntry As String)
          Combo1.Add(newEntry)
End Sub
```

A property is part of an object that stores a value like size, color, or position of an object. You can change the value of a property by writing the object name, a dot (.), property name, equal sign (=), and the new property value.

```
Sub changeName(newTitle)
          myForm.Caption =  newTitle
End Sub
```

Some properties are only readable. See the proper help entry of a property for more details. Getting a property value:

An event is an action recognized by an object, e.g. if someone clicked with the mouse or pressed a key then an event procedure is executed. Event procedures can be called by system events or manually by your program, like normal procedures.

```
Sub CheckBox1_Click(.....)
    Print "Hi"
End Sub
```

Creating object variables

Think of an object as a variable, like it would be the object itself. You can access the object through an object variable, call an object method, or property.

How to create an object variable?

- 1. Declare a object variable
- 2. Assign an object to the object variable using the 'New' keyword to create a new object or assign an existing object

Declaration of an object variable

To declare an object variable, use the 'Dim'-statement or another declaration statement ('Public', 'Private', 'Protected' or 'Static'). A variable, which references an object, must be of type 'Variant', or 'Object', or specific object type (like 'Form', 'Font'…) Some declaration of objects examples:

```
' Object1 as Variant
Dim Object1

' Object1 as general Object
Dim Object1 As Object

' Object1 as Font-Object
Dim Object1 As Font
```

When using an object variable without a declaration, the object variable has the default data type 'Variant' (IMPORTANT! using without declaration is possible only in OldBasic Mode!).

The declaration of an object variable with data type object is useful when you do not know which object type you would like to have in a generic sub.

If you know the specific object, declare the object variable with the same data type as the object. Example with normal object type and specific object type:

```
Dim Objekt1 As Object                    ' Object

Dim Objekt1 As Sample                    ' Sample
```

The declaration of specific object types has some advantages, such as type checking, more readable , and faster code execution.

Assignment of objects to an object-variable

Using the '=' statement lets you assign objects to an object variable. An object variable can have the value of 'Null' (or 'Nothing') or an object reference.

Some examples:

```
Object1 = Object2   ' object reference
Object1 = Null                ' set no object reference
```

Combine the declaration of an object variable with the creation of an object when you use the keyword ‘New’ and the '=' statement.

Example:

```
Dim Object1 As Object = New Object      ' creation and assignment
```

When assigning the value ‘Null’ to an object variable that has previously contained an object reference, use the object variable to avoid accidentally changing the object. Example:

```
If Not Object1 Is Null Then
  ' object variable references an object
  ...
End If
```

Use current instance or object / Me / Parent

The keyword ‘Me’ references the current instance (or object) in which the code is currently executed. ‘Parent’ refers to the current parent object. Normally it is the current class (user defined class or form class).

Example:

```
Sub changeObjectColor(Object1 As Object)
          Object1.BackColor = RGB(Rnd * 256, Rnd * 256, Rnd * 256)
End Sub

changeObjectColor(Me) ' statement inside the object
```

Second time use of 'Parent'

There is another meaning of ‘Parent’. If a parent class contains several constructors, you can explicitly determine the parent constructor to be called by using the ‘Parent’ keyword with the desired constructor of the parent class. This statement must be the first statement in a constructor of a child class because the constructor of a parent class must be called before any action of a child can happen. Normally, the right constructor is automatically used to call parent if you do not explicitly include it.

Example:

```
Class movies

  Protected sMovieName As String

  Sub printName
    print sMovieName
  End Sub

  Constructor movies(s As String)
    sMovieName = s
  End Constructor

End Class


Class movies2 Inherits movies

  Constructor movies2(ByRef s As String)
    Parent.movies(s + "2")
  End Constructor

End Class


Dim k As Integer = 9


Dim m As New movies2("final fantasy")

m.printName()
```

Casting of objects and simple data types

There exists some functions for casting simple data types like 'Cint.' While using objects, there is a possibility to cast objects in a parent or child class. While casting an object, it is either reduced (upcasting) or extended (downcasting). The inheritance hierarchy shows which object is a child object of another object. At the top is the parent of all objects, the class 'Object,' to bottom follow the child classes.

Upcasting and downcasting implicitly

Downcasting, the object 'o' contains more methods, variables than object 'm' and is therefore extended:

```
Dim m As Control
Dim o As CommandButton ' control is parent class
o = m
```

Upcasting, the object 'h' contains lesser methods, variables than 'm' and is therefore reduced:

```
Dim h As Control
Dim k As CommandButton ' control is parent class
h = k
```

Comparing objects

When using '=' with object variables in an expression, the expression becomes 'True' if both variables points to the same object. To compare both objects' variables or contexts, you must provide proper methods inside the classes used to compare these objects.

Asking about objects and classes

Determining types of objects at runtime:

```
If TypeOf myObject Is myClass Then
```

Returns 'True,' if the class name is the seeked name, or 'False' if it is different Syntax:

```
TypeOf objectVariable Is ClassName
```

Subclassing and inheritance

There is another possibility to extend your own classes or KBasic classes. This is called inheritance, meaning you use a parent class as a template for a new class in which you add new methods, variables, etc. All elements, like methods and variables of the parent class, are automatically inside the new class (hidden) and can be accessed and used depending upon the declaration in the parent class. To change the behavior, override a method of the parent class (have the same name and arguments), so that the algorithm of the parent class uses the new method in the new class instead of the original method in the parent class.

Inheritance is a very powerful and an important feature of object-oriented programming languages. Using the keyword 'Inherits' tells KBasic which parent class it uses for the new class:

```
Class movies

  Protected sMovieName As String

  Sub printName
    print sMovieName
  End Sub

  Constructor movies(s As String)
    sMovieName = s
  End Constructor

End Class
```

```
Class movies2 Inherits movies
...
End Class
```

Because, every class has a parent class, all classes forms together a class hierarchy (or inheritance hierarchy). The inheritance hierarchy like a tree, with a root (parent class 'Object') and many twigs (many child classes).

Constructor chaining

When declaring a class, KBasic guarantees that the class can be used to create an object because it automatically inserts a hidden default constructor with no arguments that is called when using this class without arguments. If you create a constructor, KBasic looks after the first statement in this constructor and inserts an automatic constructor call for the parent class if there is no explicit call for the parent constructor. The hidden default constructor can always be overridden.

The call of constructors is like a chain. All constructors are connected together. Every time you construct an object, it follows all constructors of all parent classes up to the parent class 'Object.' The first constructor called is the constructor of 'Object,' then the child constructors of the child classes.

The default constructor

If a class does not call a parent constructor explicitly, KBasic does it implicitly hidden. If a class does not have a constructor, KBasic implicitly inserts a standard constructor, which calls the parent constructor automatically.

Hidden Variables

An existing variable in a parent class can be overloaded by a variable in the child class through inheritance, though it is possible to access both variables with the same name. It is called hiding, because it is not automatically visible anymore. To access the parent variable, enter 'Parent' dot (.) and the variable name:

```
Parent.myVariable  ' access parent variable
myVariable         ' access me variable
```

Hidden Methods

An existing method, like a variable, in a parent class can be overloaded by a method in the child class through inheritance, though it is possible to access both methods with the same name. This is called hiding, because it is not automatically visible anymore. Hidden methods are also called overridden methods. Overriding takes place if the child class has a method with the same methods' signature (name, return type, same arguments) as the parent class. The parent method is now overridden. If you would like to access the parent method, enter 'Parent' dot (.) and the method name:

```
Parent.myMethod()    ' access parent method
myMethod()               ' access me method
```

Overriding methods

Overriding occurs if the child class has a method with the same method signature (name, return type, same arguments) as the parent class. The parent method is now overridden. If this method is called within the child class, the new defined method is used, even by the parent methods (included by the child class).

Overriding methods is a very important technique in object-oriented programming. However, do not confuse overriding with overloading. Overloading means having many methods with the same name in a class but with different method signatures (different return type or arguments).

KBasic treats variables and methods equally within a class. It is no different when it comes to overriding methods and hiding variables. You can easily access hidden variables by casting its type to the parent type or using the keyword 'Parent'. The difference is an internal meaning because variables are different when used in child and parent class (they actually use their own variable), while methods are used by both (both parent and child class use the same method!).

Dynamic-searching of methods

While compiling, KBasic cannot know which methods in which class should be called because it could be coded to dynamically search at runtime. This is called late binding, meaning it is not as fast as early binding that takes place when you call functions or procedures of a module.

Calling an overrided method

Access overridden methods by casting its object variable to the parent type or use the keyword 'Parent'.

Example:

```
Class A
  Dim i As Integer

  Function f()
    Return i
  End Functon

End Class

Class B Inherits A
  Dim i As Integer    ' this variable hides i in A

  Functon f()        ' this method overrides f() in A
    i = Parent.i + 1       ' accessing A.i
    Return Parent.f() + i  ' accessing A.F()
  End Functon
End Class
```

If you override a method, you have two methods: the old method in the parent class and the new method in the child class. Access the old method explicitly using 'Parent.' Access the new method using 'Me' or the standard call of a method. A further example:

```
' class example
Class being

  Constructor being()
    Print "being.Constructor!!!!"
  End Constructor

  Sub cry()
    Print "being.cry"
  End Sub

End Class


Class body Inherits being

  Constructor body()
    Print "body.Constructor!!!!"
  End Constructor

  Sub cry()
    Print "body.cry"
  End Sub

End Class


Class face Inherits being

  Constructor face()
    Print "face.Constructor!!!!"
  End Constructor

  Sub cry()
    Print "face.cry"
  End Sub

End Class

Dim l[10] As being

l[3] = New being
l[4] = New face
l[5] = New body

' polymorphism
```

```
l[3].cry()
l[4].cry()
l[5].cry()
```

Hiding data

Hiding data (variables and constants…) is a very important technique in object-oriented programming. Hiding data means not all data in a class can be accessed by other modules or classes, but through methods of the same class in which the data is present so that it is only visible within this class. This helps reduce mistakes when using internal data of a class by other classes or modules. You have public methods and variables that are accessible by all others and you have private methods and variables that should only be used by the class itself and those public methods. Further reasons for hiding data are:

Internal variables, which are externally visible, mixing up the API for other coders etc. Hiding them leads to small, smart classes with a few variables and methods visible to others. If a variable is visible you must document it. Hiding variables reduces documentation time.

Scope modifiers

To hide variables or methods you must declare them as 'Private':

```
Class plane

  Private wings As Integer    ' only visible within this class

  Private Funcion countWings()' only visible within this class
    ...
  End Function

End Class
```

A private variable is visible for the class in which it has been declared. This means the methods of this class can only use this private variable. Private methods and variables are not visible to child classes of the class in which they are declared (in fact they are present in the memory of that object, but not visible). Non-private methods and variables are always visible to child classes.

Besides 'Private,' there exists 'Public' and 'Protected' as scope modifiers. Protected is useful if the variable or method should be visible only to child classes and the class in which they are declared. Public means that every part of your program (any other class or module) can access and use the public declared part of your class (variable or method). The default modifier is 'Public.' If you do not write 'Public' in front of your variable or method, it is automatically declared as public.

Some tips for using scope modifiers

Use 'Private' for methods and variables used only in the class, but not outside. Use 'Public' for methods and variables that are used and accessible from everywhere.

Abstract Classes and Methods

It is possible in KBasic to define methods that have no code, but can be used as an template for child classes. This technique is called 'abstract declaration.' Abstract methods are useful when you do not have code in a method of a parent class but must be present in the parent class for design reasons. Child classes now are forced to implement this method with some code in it. It is also possible to mark a complete class as abstract; you cannot create an object of that class because this class contains declarations but no code. The code should be included in the child classes, which can that be used after the 'New'-keyword.

Normally, you do not use abstract methods or classes, but it could be useful. Some rules when dealing with abstract: Every class containing an abstract method is automatically declared as abstract. An abstract class must contain at least one abstract method.

An abstract class cannot be used to create an object. A child class of an abstract class can be used to create an object if all abstract methods have been overridden with a code implementation. It is non-abstract now. If a child class of an abstract class does not override all abstract methods it become an abstract class.

Arrays

If you have worked with other programming languages, you understand the concept of arrays. Arrays are variables that look like a chain. All elements of that chain have the same data type and can be accessed by an index. These variables have one name but a different index and are placed as one block in memory. You can use the array as one block or as one element of the array. Arrays are declared the same as other variables are declared (use 'Dim', 'Static', 'Private', 'Public' or 'Protected').

Using arrays leads to cleaner, more flexible code because you can use loops to use or access thousands of variables (the arrays). Arrays have an upper and a lower boundary and elements of that array are between those boundaries. If an array has the data type 'Variant,' each element of the array can contain a different data type.

Different kinds of arrays

There are two types of arrays that differ in one issue: one type has a fixed size while compiling and the other type changes its size dynamically at runtime.

- Dynamic array: changes the number of indexes at runtime
- Static array: fixed number of indexes

It is possible not to know the length of an array while coding or you would like to change its size at runtime. Another advantage to dynamic arrays is that you can free memory when using dynamic arrays. Static arrays are always in memory.

Declaration

Declare an array using 'Dim' and length declaration in ( ) or [ ]:

Syntax:

```
Dim variableName(Index) As Type

Dim variableName[Index] As Type

Dim variableName[Index, Index, ...] As Type

Dim variableName[Index To Index] As Type

Dim variableName[Index To Index, Index To Index, ...] As Type
```

Example:

```
Dim i(100) As Integer
Creates an array with 100 'Integer'-variables.
Accessing array elements
Access an element of an array when you write an integer expression in braces () or [] after the name of an array:
i(3) = 10
```

Access with position. The position is 1 in the example:

```
i(1) = 0
```

The index is always checked. If it is too small or too big you run into an error. This is a further feature of KBasic to avoid program crashes. The expression, which stands for the index, can be calculated at runtime. Any numerical expression is allowed:

```
n = 100
Dim i(n) As Integer
```

Getting lower and upper bound

UBound is used for the upper bound:

```
UBound (arrayVar [,(Dimension])
```

LBound is used for the lower bound:

```
LBound (arrayVar [,(Dimension])
```

Default: An array with declared 10 elements is counted from 0..10 (including 10). Therefore, 0 and 10 are usable indexes and there are 11 elements declared. In fact, an array contains always one more element than declared when you do not specify lower and upper bound.

Explicitly declaring the lower bound

```
Dim i(50 To 100) As Integer
```

This creates an array with a lower bound of 50. Example:

```
Dim curCosts (364) As Currency

Sub createArray ()
          Dim curCosts (364) As Currency
          Dim intI As Integer

          For intI = 0 To 364
                    curCosts (intI) = 20
          Next
End Sub
```

Changing the lower bound

Use the 'Option Base'-statement on top of an class or module file to change the default index of an array element from 0 to 1. In the following example, the first accessible index will be 1:

```
Option Base 1

Dim curCosts(365) As Currency
```

Change the upper bound of an array using the 'To'-keyword. Example:

```
Dim curCosts(1 To 365) As Currency
Dim sWeekday(7 To 13) As String
```

Use of multidimensional arrays

KBasic supports multidimensional arrays as well. The number of dimensions is limited to 32. To create a multidimensional array:

```
Dim i(100, 50, 400)
```

The following statement declares a two-dimension array:

```
Dim sngMulti(1 To 5, 1 To 10) As Single
```

An array with two dimensions is like a matrix. The first argument can be considered the row and the second the column. If you use arrays with two dimensions or more, you can easily use the 'For-Next' loop to iterate and access the elements of that array.

```
Sub changeArray ()
          Dim intI As Integer, intJ As Integer
          Dim sngMulti(1 To 5, 1 To 10) As Single

          ' set values of array
          For intI = 1 To 5
                    For intJ = 1 To 10
                              sngMulti(intI, intJ) = intI * intJ
```

```
                            Print sngMulti(intI, intJ)
                    Next intJ
            Next intI
End Sub
```

Save 'Variant'-values in arrays

There are two ways to create arrays with elements of type ‘Variant.' First, you can create an array with data type ‘Variant.' Example:

```
Dim varData(3) As Variant
varData(0) = "Christiane Roth"
varData(1) = "60529 Frankfurt"
varData(2) = 26
```

You can also use the ‘Array’-Function, which returns an array of type ‘Variant’ to create an array. Example:

```
Dim varDaten As Variant
varData = Array(" Christiane Roth", "60529 Frankfurt", 26)
```

Accessing the array is the same for both methods: use the index. Example:

```
Print "Data " & varDaten(0) & " has been stored."
```

Change the size of a dynamic array

Declare the array using the ‘Dim’-statement without the index expression but with braces () or [], e.g.

```
Dim a() As Integer
```

Reserve the right count of array elements with ‘Redim.' The statement ‘Redim’ can only be used inside procedures. It has the same syntax as ‘Dim.' Every ‘Redim’-statement can change the number of elements and also the lower and upper bound of that array. The number of dimensions cannot be changed with ‘Redim’.

Every time you use ‘Redim,' all values of the array are deleted if you do not use the keyword ‘Preserve.' KBasic sets the values to ‘Empty’ (when type is ‘Variant’), or value 0 (numerical types), or “” (if ‘String’). Object types will be ‘Null’. Example:

```
Function calc() As Integer
  Dim Matrix1 () As Integer

  Redim Matrix1 (22, 33)
  ...
```

You can also use variables instead of literals:

```
Redim Matrix1 (a, b)
```

Syntax:

```
Redim variableName(Index)

Redim variableName[Index]

Redim variableName[Index, Index, ...]

Redim variableName[Index To Index]

Redim variableName[Index To Index, Index To Index, ...]
```

Array-reset / array-delete

The command ‘Erase’ deletes arrays and frees the memory of arrays.

```
Erase array1[, array2]
```

Control of the program flow

The statements that make the decision and loops are known as control structures.

Decisions

The term ‘decisions’ refers to the use of conditional statements to decide what to execute in your program. Conditional statements test if a given expression is ‘True’ or ‘False.' Then, statements are executed. Normally a condition uses an expression in which a comparison operator is used to compare two values or variables. KBasic supports all VB6 conditional statements and more.

Single decision - If

A single decision is used to execute a set of statements if a condition is set (‘If’-statement). If the condition is ‘True’ then the statements after the ‘Then’ are executed and the statements after the ‘Else’ are skipped. If the condition is ‘False,' the statements after the ‘Else’ are executed. If the item after ‘Then’ is a line number, a goto is executed. If the condition is ‘True’ and there is no ‘Then’ statement on the same line, statements are executed until a line with an ‘End If’ is found. If you run in ‘KBasic Mode’ the expression must be a boolean expression, otherwise any other numerical expression is allowed. Syntax:

```
If Expression Then Statement

If Expression Then Statement : Else Statement

If Expression Then LineNo

If Expression Then LabelName:

If Expression Then
  [Statements]
End If

If Expression Then
  [Statements]
Else
  [Statements]
End If

If Expression Then
  [Statements]
ElseIf Expression
  [Statements]
Else
  [Statements]
End If

If Expression Then
  [Statements]
ElseIf Expression
  [Statements]
ElseIf Expression
  [Statements]
Else
  [Statements]
End If

If Expression Then
  [Statements]
ElseIf Expression
  [Statements]
End If
```

‘Else’ introduces a default condition in a multi-line ‘If’-statement.

‘ElseIf’ introduces a secondary condition in a multi-line ‘If’-statement.

‘End If’ ends a multi-line ‘If’-statement.

‘If’ evaluates ‘expression’ and performs the ‘Then’-statement if it is ‘True’ or (optionally) the ‘Else’-statement if it is ‘False’. KBasic allows multi-line ‘If’-statements with ‘Else’ and ‘ElseIf’ cases, ending with ‘End If’. This works as zero is interpreted to be ‘False’ and any non-zero is interpreted to be ‘True’

Example:

```
Dim i As Integer
Dim n As Integer

If i = 1 Then
  n = 11111
ElseIf i = 2 * 10 Then
  n = 22222
Else
  n = 33333
End If
```

IIf – short if

IIf returns one of two values depending on an expression. Example: testing = IIf(Test1 > 1000, "big", "small")

Syntax:

```
IIf(Expression, ThenReturnExpression, ElseReturnExpression)
```

Select Case

The 'Select Case'-statement is much more complicated than the 'If'-statement. In some situations, you may want to compare the same variable or expression with many different values and execute a different piece of code depending on which value it equals to. This is exactly what the 'Select Case'-statement is for. 'Select Case' introduces a multi-line conditional selection statement. The expression given as the argument to the 'Select Case'-statement will be evaluated by 'Case'-statements following. The 'Select Case'-statement concludes with an 'End Select'-statement. As currently implemented, 'Case'-statements may be followed by string values, in this case complex expression can be performed. The test expression can be 'True,' 'False,' a numeric, or string expression. 'Select Case' executes the statements after the 'Case'-statement matching the 'Select Case'-expression, then skips to the 'End Select'-statement. If there is no match and a 'Case Else'-statement is present, then execution defaults to the statements following the 'Case Else.' Syntax:

```
Select Case Expression
Case Expression
  [Statements]
Case Expression
  [Statements]
End Select

Select Expression ' modern style
Case Expression
  [Statements]
Case Expression
  [Statements]
End Select

Select Case Expression
Case Expression
  [Statements]
Case Expression To Expression
  [Statements]
Case Is Expression
  [Statements]
Case Else
  [Statements]
End Select
```

Example:

```
Dim i As Double
Dim n As Integer

i = 4

Select Case i
Case 0
  n = 0
Case 1, 2
  n = 1122
Case 4 TO 10
  n = 441000
Case Is = 9
  n = 9999
```

```
Case Else
  n = 999999
End Select
```

Switch – short select case

'Switch' returns a value depending on an expression. Example:

```
Dim s As String
Dim i As Integer
i = 1
s = Switch(i = 1, "Bible", i = 2, "Casanova")
Print s
```

Syntax:

```
Switch(Expression, ReturnExpression[, Expression, ReturnExpression, ... ])
```

Choose – short select case

'Choose' returns one value from a list of values depending on the index.

```
Dim s As String
s = Choose(1, "un", "deux", "troi")
Print s
```

Syntax:

```
Choose(Expression, ReturnExpression[, ReturnExpression, ... ])
```

Unconditional jumping

Most of this chapter has been dedicated to conditional branches. If you recall, however, programmers can also use unconditional branches. This type of branching can be performed using the 'GoTo'-instruction. 'GoTo' forces program execution to branch to a specific line number or label. Because line numbers in KBasic programs are now obsolete, you do not have to worry about how to use them. You may, however, want to use labels. 'GoTo' performs a unconditional jump. 'GoTo' is always executed, without a condition. Syntax:

```
GoTo {LineNo | Label:}

GoTo myExit:
GoTo nextStep:
```

With-Statement A very useful statement is the 'With'-statement. When using the 'With'-statement, you are able to group assignments or statements that reference the same object. This makes your code becomes more readable in addition to reducing redundant code.

```
Sub formSection ()
          With myClass
                    .Value = 30
                    .Bold = True
          End With
End Sub
```

You can write a 'With'-statement inside another 'With'-statement:

```
Sub setValue ()
          With j(3)
            .e.bkname = "Frankfurter Zoo"
            With .e
              .isbn ( 99 ) = 333
            End With
          End With
End Sub
```

Loop-statements

The statements that control decisions and loops in KBasic are called control structures. Normally every command is executed only one time but in many cases it may be useful to run a command several times until a defined state has been reached. Loops repeat commands depending upon a condition. Some loops repeat commands while a condition is 'True,' other loops repeat commands while a condition is 'False.' There are other loops repeating a fixed number of times and some repeat for all elements of a collection.

For Next

The For-Next loop is useful when you know how often statements should be repeated. For-Next defines a loop that runs a specified number of times.

Syntax:

```
For counter = start To stop [Step stepExpression]
  [Statements]
Next [counter]
```

'Counter' is a numerical variable used to store the current counter number while the loop is running. 'Start' is a numerical expression that determines the starting value of the loop countings. 'Stop' is a numerical expression that determines the ending value at which the loop will stop. To count in steps other than the default value of 1, use 'stepExpression' to determine a specific value for incrementing the counter. If you use 'Exit For,' this exits the 'For Next' loop immediately and continues with the line following to the 'Next' statement. This is usually used in connection with an 'If' clause (If something Then Exit For) to exit the loop before its specified end.

The 'Next' statement is the lower end of the loop and increments 'counter' by the value given by 'stepExpression' or by 1 if no 'stepExpression' incrementation is defined. The loop repeats as long as the value of 'counter' is smaller/less than or equals the value of 'stop.' The indication of 'counter' may be left out for the 'Next' statement, however this is depreciated as it can lead to confusion when nesting several ' 'For Next' loops.

Notes: The speed of 'For Next' loops depends on the variable types used. 'For Next' loops run fastest when 'counter' is an integer variable and 'start,' 'stop,' and 'stepExpression' are integer expressions. Count backwards using 'stepExpression' with a negative incrementation value. Take extra care when nesting 'For Next' loops. Remember to end the loop last initiated (see example) or an infinite loop occurs.

' Example 1:

```
Dim ctr As Integer

For ctr = 1 To 5
  Print "Z";
Next ctr

' Output:
' ZZZZZ

' Example 2:
' Here we use STEP
' The resulting string is the same
' as in example 1

Dim ctr As Integer

As Integer ctr = 1 To 50 Step 10
  Print "Z";
Next ctr

' Output:
' ZZZZZ


' Example 3:
' These are nested loops.
' The "y" loop is the INNER one
' and thus will end first:

Dim x, y As Integer

For x = 1 To 5
  Print "Line" + Str$(x)
  For y = 1 To 5
    Print "Z";
  Next y
```

```
  Print "-"
Next x

' Output:
' Line 1
' ZZZZZ-
' Line 2
' ZZZZZ-
' Line 3
' ZZZZZ-
' Line 4
' ZZZZZ-
' Line 5
' ZZZZZ-
```

Optimize For...Next-Loops

Use counter variables of type 'Integer' instead of 'Variant,' because 'Integer' is much faster.

Example:

```
Dim rabbit As Integer         ' counter of type Integer

For rabbit = 0 To 3276
Next rabbit


Dim hedgehog As Variant       ' counter of type Variant.

For hedgehog = 0 To 3276
Next hedgehog
```

The first example runs faster. The smaller the data type, the faster it runs. Variables of type 'Variant' are always slower than the directly used data type.

Other kind of loops

Use the following loops when you are not sure how often a command should be repeated: 'Do', 'While', 'Loop', 'Until' or ('Wend') . There are two different ways to use the keyword 'While' in order to test a condition within a 'Do…Loop'-statement. You can test the condition before the commands inside the loop are executed or you can test the condition after the commands of the loop have been executed at least once. If the condition is 'True' (in the following procedure 'SubBefore') the commands inside the loop execute. If you set 'myNumber' to 9 instead of 20, no commands inside the loop execute. Inside procedure 'SubAfter,' all commands execute at least once because the condition is not true.

```
Sub SubBefore()
  Counter = 0
  myNumber = 20
  Do While myNumber > 10
    myNumber = myNumber - 1
    Counter = Counter + 1
  Loop
  MsgBox "Loop has been executed " & Counter & " time(s)."
End Sub

Sub SubAfter()
  Counter = 0
  myNumber = 9
  Do
    myNumber = myNumber - 1
    Counter = Counter + 1
  Loop While myNumber > 10
  MsgBox "Loop has been executed " & Counter & " time(s)."
End Sub
```

Two possibilities allow you to use the keyword 'Until' to test a condition of a 'Do…Loop'-statement. You can test the condition before the loop (see procedure 'SubBefore' in the example), or you can test the condition after the loop has been entered (see procedure 'SubAfter' in the following example). The loop repeats as long as the condition is 'False'.

```
Sub SubBefore()
  Counter = 0
  myNumber = 20
  Do Until myNumber = 10
    myNumber = myNumber - 1
    Counter = Counter + 1
```

```
  Loop
  MsgBox "Loop has been executed " & Counter & " time(s)."
End Sub

Sub SubAfter()
  Counter = 0
  myNumber = 1
  Do
    myNumber = myNumber + 1
    Counter = Counter + 1
  Loop Until myNumber = 10
  "Loop has been executed " & Counter & " time(s)."
End Sub
```

Do While...Loop

A 'Do While' loop is a group of statements enabling you to define a loop that will repeat until a certain condition remains 'True'. 'Do': launches the loop and must be the first statement 'Loop': ends the loop and must be the last statement 'While': lets the loop repeat while the condition is 'True' Condition: a numeric or string expression with a result of 'True' or 'False' 'Exit Do': exits the loop at this very line and lets the program continue behind the 'Loop'-statement 'Iterate Do': jumps directly to the 'Loop'-condition

Syntax:

```
Do While Expression
  [Statements]
Loop
```

Examples: In the first example, the loop repeats as long as 'xyz' remains 5:

```
Do While xyz = 5
(lines of code)
Loop
```

Please note the lines of code will never be executed if 'xyz' is not 5 when entering the loop. To overcome the problem of "never executed code" move the condition to the end of the loop:

```
Do
  (lines of code)
Loop While xyz = 5
```

Now the lines of code execute at least one time before the loop checks for the value of 'xyz' and decides whether to exit or repeat execution. If there is a second condition that arises somewhere within the loop making it necessary to exit the loop before the lines of code within it end, the 'Exit'-statement may be helpful:

```
Do
  (lines of code)
  If abc = 0 Then Exit Loop
  (lines of code)
Loop While xyz <> 5
```

If 'abc' becomes zero at the 'If'-statement within the loop, the program exits the loop. The rest of the code lines are skipped. Sometimes it is necessary to skip the rest of the code lines but nevertheless stay in the loop to be repeated. Now you may use 'Iterate' instead:

```
Do
  (lines of code)
  If abc = 0 Then Iterate Loop
  (lines of code)
Loop While xyz <> 5
```

If 'abc' becomes zero at any moment within the loop,the program skips the rest of the code lines within the loop and jumps directly to its "Loop While xyz = 5." The loop checks for the value of xyz and decides what to do. You may use the condition with a number of expressions like 'And' and 'Or':

```
Do While x < 10 And y < 10
  (lines of code)
Loop
```

The loops repeats while x and y are smaller than 10. Note: Please be careful when nesting several loops within each other. Sometimes a variable checked for the outer loop is changed within the inner loop. In this case, the outer loop never ends:

```
Do While counter < 10
  (lines of code)
  counter = 0
  Do
    counter = counter + 1
  Loop While counter <> 5
Loop
```

Good programming practice would recommend using separate variables for each loop. The same might happen with a 'For…Next' loop within:

```
Do While counter < 10
  (lines of code)
  For counter = 1 To 5
    (lines of code)
  Next counter
Loop
```

Moreover, be careful not to interchange the different loops:

```
Do While counter < 10
  (lines of code)
  For i = 1 TO 5
    (lines of code)
  Loop
Next i
```

This results in an error because 'Loop' has to appear after 'Next.'

```
x = 5
Do While x = 5
  Print x
Loop
```

Do...Loop Until

A 'do loop until' repeats until a condition is set. Example:

```
x = 0
Do
  Print x
  x = x + 1
Loop Until x > 3
```

Syntax:

```
Do
  [Statements]
Loop Until Expression
```

Do...Loop While

A 'do loop while' repeats while a condition is set. Example:

```
x = 0
Do
  Print x
  x = x + 1
Loop While x < 3
```

Syntax:

```
Do
  [Statements]
```

```
Loop While Expression
```

Do Until...Loop

The 'do until loop' is a group of statements enabling you to define a loop that repeats until a certain condition remains 'True.'

```
x = 0
Do Until x > 3
  Print x
  x = x + 1
Loop
```

Syntax:

```
Do Until Expression
  [Statements]
Loop
```

While...Wend

'While' initiates a 'While…Wend'-loop. The loop ends with 'Wend' and execution reiterates through the loop as long as the expression is 'True'. Example:

```
While True
  Print 1234
Wend
```

Syntax:

```
While Expression
  [Statements]
Wend
```

While...End While

This is the same as 'While…Wend', but with slightly different keywords. This loop reiterates through the loop as long as the expression is 'True'.

```
While True
  Print 1234
End While
```

Syntax:

```
While Expression
  [Statements]
End While
```

Explicitly leave a loop

Normally a loop ends when its condition allows it. Sometimes it might be useful to exit a loop before the condition is met. Manually exit a loop using the 'Exit Do'-statement. To leave 'For'-loops, use 'Exit For'.

- 'Exit For' – leave a for loop
- 'Exit Do' – leave a do loop

For example, use 'Exit Do' in a 'If'-statement or 'Select Case'-statement if you want to leave an infinite loop.

Syntax:

```
Exit For
Exit Do
```

Explicitly test a condition

A loop condition is tested after every loop iteration. However, it might be useful to test the loop condition earlier. Manually test a loop condition using the 'Iterate Do'-statement. As a result, the current loop condition is tested. To test 'For'-loop conditions, use 'Iterate For'.

- 'Iterate For' – manually test loop condition for a 'For'-loop
- 'Iterate Do' – manually test loop condition of a 'Do'-loop

Syntax:

```
Iterate For
Iterate Do
```

Nested control structures

Embed control structures into other control structures by putting a 'For'-loop into an 'If'-statement. This is called nested control structures. You can embed as many control structures as needed, but you should indent every control structure line so that your code is easier to read. The examples in this manual are always indented and formatted.

Procedures

Your programs have been short, each designed to demonstrate a single programming technique. When you start writing real programs, however, you will discover they can grow to many pages of code. When programs increase in length, they become harder to organize and read. Professional programmers use modular programming to decrease the length of programs. Modular programming uses procedures. A procedure is like a small program within your main program. KBasic source code is inside a procedure, normally. A procedure is a set of commands inside the written words 'Sub' and 'End Sub' or 'Function' and 'End Function'. There are different types of procedures.

- Sub (inside a module or class) - Sub-procedures contain commands and statements but do not return a value or cannot be used in expressions. Event-procedures are always sub-procedures. An event-procedure is related to a form or control. When KBasic notices an event of a control occurred, it calls the related event-procedure.
- Functions (inside a module or class) - Function-procedures contain commands and statements. Function-procedures always return a value, e.g. the result of a complex math operation. Because functions return values, they can be used in expressions. Functions like subs can have arguments.
- Sub (inside a class) - Better named as method, without return value. This is like a sub-procedure.
- Function (inside a class) - Better named as method with return value. This is like a function-procedure

Sub-Procedure

A sub-procedure can have arguments, e.g. variables, expressions, or constants that are given to the sub-procedure while calling it. Many built in-functions of KBasic have arguments.

Syntax:

```
Sub Name([Arguments])
   [Statements]
End Sub

Sub Name([Arguments]) [Throws Name, ...]
   [Statements]
End Sub
```

Function-Procedure

A sub-procedure can have arguments, variables, expressions, or constants that are given to the sub-procedure when calling it. Function-procedures return values. Many built in-functions of KBasic have arguments (e.g. Abs). Syntax:

```
Function Name([Arguments]) [As Type]
  [Statements]

End Function

Function Name([Arguments]) [As Type] [Throws Name, ...]
  [Statements]
End Function
```

Arguments

For all practical purposes, a procedure can have as many arguments as needed. You must be sure the arguments you specify in the procedure's call exactly match the type and order of the arguments in the procedure's sub line. To use more than one argument in a procedure, separate the arguments with commas. You can pass one or more arguments to a procedure. Keep in mind the arguments are passed to the procedure in the order in which they appear in the procedure call. The procedure's sub line must list the arguments in the same order they are listed in the procedure call. If a procedure does not have arguments, you must not write any expression or variable inside the braces when calling it.

All statements and commands are executed only after calling the procedure. Arguments have names. If you have many arguments, you can separate them by a comma (,). Every argument is like a variable declaration and leads to automatically declared variables when the statements and commands of a procedure are executed.

Syntax of arguments:

```
Name As Type

[ByVal | ByRef] Name As Type

[ByVal | ByRef] Name [As Type]

[ByVal | ByRef] Name [()][As Type]

[ByVal | ByRef] [Optional] Name [()][As Type] [= Expression]
```

The type can be one of the built in types (scalar types), 'Variant' or object/class type. If you do not tell KBasic the type, it assumes the default type that is normally 'Variant'.

'ByRef' and 'ByVal' are modifiers. If you call a procedure and one argument is a variable and the matching argument is declared 'ByRef', then KBasic performs all operations on that argument on the given variable and does not copy the variable's value. 'ByVal' leads to a copy of the variable's value. If the commands inside the procedure changes the value of the argument, the variable is not affected. If you use 'ByRef' it is changed!

Named and optional arguments

There are two ways to give arguments to a procedure. One is to name the arguments, the other is the position relevant (default) one. Arguments – the normal way:

```
PassArg(Frank", 26, #2-28-79#)
```

Named arguments:

```
PassArg(intAge = 26, birthDate := #2/28/79#, strName := "Frank")
```

If you use named arguments, you are able to list the arguments in the order as you like. It is probably more readable. Example:

```
Sub passArg(strName As String, intAge As Integer, birthDate As Date)
          Print strName, intAge,  birthDate
End Sub
```

A named argument contains the argument name, colon (:) and equal sign (=), and the expression the argument should have. Named arguments are also useful when you use procedures with optional arguments. Optional arguments are not always present in the procedure, depends upon if they have been given to the procedure. Optional arguments can have default values as seen in the following example:

```
Sub optionaleArg(str As String, Optional strCountry As String = "Germany")
...
End Sub
```

If you do not give a optional argument to a procedure, the default value is used.

It might be useful to check the argument with the 'IsMissing'-function:

```
Sub optionalArg(str As String, _
Optional strCountry As String = "Germany")
          If IsMissing(strCountry) Then
                    Print str
          Else
                    Print str, strCountry
          End If
End Sub
```

You can call the procedure above with the following lines:

```
optionalArg(str := „test1", strCountry := "Germany")
optionalArg(str := „test1")
```

Arguments with a default value are also possible as seen in the following example:

```
Sub optionaleArg(str As String, strCountry As String = "Germany")
...
End Sub
```

Writing function-procedure

A function-procedure contains commands and statements, which are after 'Function' and before 'End Function.' A function-procedure is like a sub-procedure but it can return a value. That is the only difference between them. A function-procedure can also have arguments. A value can be returned using the function name as an variable, which is used in an assignment statement (old style). You should use the keyword 'Return' for return values (new style). Example:

```
Sub Main()
          temp = InputBox(Prompt:= _
                    "Fahrenheit?")
          MsgBox "Fahrenheit = " & Celsius(temp)
End Sub

Function Celsius(fGrad)
          Celsius = (fGrad - 32) * 5 / 9
End Function

Main()
```

Call of sub-procedures and function-procedures

To call a sub-procedure within another procedure, write the name of the sub-procedure to call and all needed arguments. The keyword 'Call' is not needed. If you use it, all arguments must be placed inside braces. You ought to use procedures to organize and split your code to get it more readable. Example:

```
Sub Main()
          MultiBeep 56
          DoMessage
End Sub

Sub MultiBeep(no)
          Dim counter As Integer
          For counter = 1 To no
                    Beep
          Next counter
End Sub

Sub DoMessage()
          Print "Tee time!"
End Sub

Main()
```

Add comments to a procedure

When creating a new procedure or changing code, comment the new or changed code. Comments have no effect on the program when it is executed, they only help other developers understand the code. Comments start with ('). This character tells KBasic to ignore all text until reaching the end of line. Find more information about comments in the previous chapter about comments. Call of procedures with same name: You can call a sub-procedure from everywhere. If it is a different module, it might be useful to use module name to address the sub-procedure. Example:

```
Sub Main()
          Module1.myProcedure()
End Sub
```

Hints for calling procedures

When changing the name of classes or modules, be sure to change all places the class or module is used or KBasic runs into an error. You could use the file replace in the KBasic menu. To avoid name collisions, give your procedures unique names.

Leave procedure

You can leave the procedure at any line inside of a procedure. For that, use the keywords 'Exit Sub' inside a sub-procedure or 'Exit Function' inside a function-procedure.

```
Syntax:

Exit Sub

Exit Function  ' inside a function-procedure
```

Calling procedures with more than one argument The following example shows calling a sub-procedure with more than one argument.

```
Sub Main()
          costs (99800, 43100)
          Call costs (380950, 49500) ' old style
End Sub

Sub costs (price As Single, income As Single)
          If 2.5 * income <= 1.8 * price Then
                    Print "House too expensive."
          Else
                    Print "House is cheap."
          End If
End Sub
```

Use of brackets while calling a function-procedure

In order to get the return value of a function-procedure, you need a variable. Example:

```
answer3 = MsgBox("Are you satisfied with your income?")
```

If you do not need the return value, ignore it and use the function-procedure like an ordinary sub-procedure. Example:

```
MsgBox "Task done!"
```

Writing recursive procedures

Procedures have limited memory, so if a procedure calls itself, it consumes memory again. When procedures call themselves they are called recursive procedures. Example:

```
Function doError(Maximum)
           doError = doError(Maximum)
```

```
End Function
```

These errors can be hidden when two procedures are calling each other once and once again or if no break out condition is used. In some situations recursive procedures might be useful. Example:

```
Function Fakulty (N)
          If N <= 1 Then      ' end of recursive
            Fakulty = 1       ' (N = 0) no more calls
          Else ' Fakulty is called again
            ' when N > 0.
            Fakulty = Fakulty(N - 1) * N
          End If
End Function
```

Carefully test a recursive procedure to verfiy it works as expected. If an error occurs, check the break out condition. Try to minimize memory consumption:

Release unused variables

Avoid data type 'Variant' Check the logic of the procedure. Use loops inside loops instead of recursion calls.

Overloading of procedures

In other languages, the name of the method (or function, or procedure) is enough to distinguish it from other methods in the program. In KBasic, procedures are not only distinguished by their name but also by their arguments. If you call a procedure and there are two or more procedures with the same name, KBasic tries to verify which procedure you mean by checking the arguments.

Defining procedures with the same name, but different arguments is called overloading. It might be useful in some situations. Do not mix it up with overriding.

Event procedures

When KBasic recognizes that a form or control has raised an event, it automatically calls the related event-procedure within a form.

General procedures

KBasic calls an event-procedure after an event occurs. Normal procedures are only called explicitly. If you need several event-procedures to perform a task, create one sub-procedure that is called inside the event-procedure. This is better than copying all code in each event-procedure and duplicating code. Create normal procedures in forms or modules. If the normal procedure is related to a form only, place it there. If it is related to the entire program, place it in a module.

Functions

Functions are very similar to subs in that both are procedures, but functions return a value and can be used in expressions.

Syntax:

```
Function Name([Arguments])[As Type]
  [Statements]
End Function

Function Name([Arguments])[As Type] [Throws Name, ...]
  [Statements]
End Function
```

The arguments of a function-procedure are used as the arguments of a sub-procedure. Define a function-procedure by using the keyword 'Function.' You should also define a return type for every function. For more information about using procedures see the previous chapter.

Data Types of functions

Just as variables have data types, function-procedures have data types. When you define a function you also define a return type, which is the data type of the function. If you do not define a return type, the default data type is assumed to be 'variant.'

Function Return

If you would like to set the return value of a function and exit the function, use 'Return.' With this statement you immediately leave the function.

Syntax:

```
Return Expression
```

Properties

Most objects and controls have properties that you can think of as nothing more complicated than attributes. For example, a 'CommandButton' control has a property called 'Caption' that determines the text that appears in the button. Many other controls, such as the familiar 'Label' control, also have a 'Caption' property. Some properties are common in KBasic whereas others are associated with only a single control or object. Those properties are built into KBasic, but it is also possible to define your own properties within your user defined class.

Defining a property is like defining a procedure. In fact, a property contains two property-procedures. One property-procedure reads the property (Get) and the other writes the property (Set). Additionally, you have to declare a variable that contains the value of the property. Why should I use a property, when I use a variable instead? It is easier to access a variable but it might be useful to check something when accessing a variable; with properties you are able to check values or conditions when you try to access the property. It is like data hiding. You can code so that the value of the property is always correct.

For details, please refer to the examples that come with KBasic.

Syntax:

```
Property Set Name(Argument)
  [Statements]
End Property

Property Get Name As Type
  [Statements]
End Property


Property Name As Type

  Get
    [Statements]
  End Get

  Set(Argument)
    [Statements]
  End Set

End Property
```

Conversion

Normally you do not have to change the data type of an expression. It is usually done for you without your intervention by KBasic. But sometimes it might be important to force a conversion or else your calculation would be wrong. There are some built in conversion functions for any data type:

```
y% = CInt (expression)  ' returns Integer
y% = CLng (expression)  ' returns Long
y! = CSng (expression)  ' returns Single
y' = CDbl (expression)  ' returns Double
y@ = CCur (expression)  ' returns Currency
```

```
y = CVar (expression)   ' returns Variant
y = CBool (expression)  ' returns Boolean
y = CByte (expression)  ' returns Byte
y = CShort (expression) ' returns Short
y = CDate (expression)  ' returns Date
```

Modifiers / Scopes

Modifiers and scopes are a very important aspect of the KBasic programming language and of modern programming languages in general.

Scopes tell KBasic when a variable, a constant, or a procedure can be used, from which place in the source code files, actually. There are different scopes (abstract places in code):

- Procedure scope
- Global scope
- Private module scope
- Public module scope
- Private class scope
- Protected class scope
- Public class scope

Scopes are useful when you would like to organize your program or want to avoid name collisions of variables.

When you refer to a variable within your method definitions, KBasic checks for a definition of that variable first in the current scope, then in the outer scopes of the current method definition. If that variable is not a local variable, KBasic then checks for a definition of that variable as an instance or class variable in the current class, and then, finally, in each parent class in turn.

Because of the way KBasic checks for the scope of a given variable, it is possible for you to create a variable in a lower scope such that a definition of that same variable hides the original value of that variable. This can introduce subtle and confusing bugs into your code. The easiest way to get around this problem is to make sure you do not use the same names for local variables as you do for instance variables. Another way to get around this particular problem, however, is to use 'Me.variableName' to refer to the instance variable and just variable to refer to the local variable. By referring explicitly to the instance variable by its object scope you avoid the conflict.

Local scope

A variable declared inside a procedure is not usable outside the procedure, only the code of the procedure in which the variable has been declared can use it. The following example demonstrates how a local variable may be used. There are two procedures, one of them has a variable declared:

```
Sub localVar()
  Dim str As String
  str = "This variable can only be used inside this procedure"
  Print str
End Sub

Sub outOfScope()
          Print str  ' causes an parser error
End Sub
```

Private module scope

If you declare a variable as 'Private,' it is only visible to the module in which it is declared. When declared as 'Public,' the variable is visible to the entire program (all modules and classes). Default is 'Private' when you use the keyword 'Dim' to declare variables.

```
Example:
' add to the declaration section of your
' module the following lines
Private str As String

Sub setPrivateVariable()
  str = "This variable may be only used inside this module"
```

```
End Sub

Sub usePrivateVariable()
  Print str
End Sub


Public module scope
When you declare a variable as 'Public' inside a module, it is global and visible to all parts of your program.
Example:
' add to the declaration section of your
' module the following lines
Public str As String
```

All procedures are as default 'Public' except the event-procedures because KBasic automatically writes 'Private' for every event-procedure declaration. If you want to have a normal procedure as 'Private,' write the keyword 'Private' before the declaration of that procedure.

User Defined Data Type (Type...End Type)

A user defined data type is very useful when object-oriented programming is not available. It is like a class, many different variables are held together but without methods. Many kind of data types are allowed inside a user defined data type, even other user defined data types can be included.

```
Syntax:
Type Name
 Name [(Index)] As Type
  ...
End Type
```

Enumeration (Enum...End Enum)

Enumeration is a new way of grouping constants related to the same subject. It is a list of names and every name has a constant value.

```
Syntax:

Enum Name
  Name [= Expression]
  ...
End Enum
```

Classes

A simple application can contain a form, while all source code is inside the form module of that form. But if the application grows bigger and bigger it might be useful, to write source codes from different forms at one place, so you need a place to write the source codes down outside the forms. Here comes the classes. Create a class, which contains a methods, which is useful for your forms. You KBasic code is stored in classes or modules. You can archive your code within classes. Every class consists of the declaration part and the methods you have inserted. A class can contain:

- Declarations - for variables, types, enumerations and constants
- Methods (also called procedures) - which are not assigned to a special event or object. You can create as many procedures as you want, e.g. sub-procedures without return value or function-procedures.
- Signal/Slots-Methods - These are special methods, which are only useful together with the bindings. See the documentation of the bindings in the KBasic IDE for more information.
- Properties - Are variables, which are accessed through two special methods (get and set method). Properties are accessable without the need to write braces, when you use them.

You can put several classes or modules in one file, but you should not do that.

Classes unlike procedures are not executed

A class consists of a declaration part only; you must create the methods yourself. Additionally, a class contains no main program. Methods are executable inside classes and execute if the proper event is raised or if another part of

your program has called a method of that class.

Edit Class

It is not much different editing text in a word processor or in KBasic's Software IDE. You have a cursor showing the current position you can type. You can also find and replace inside your source code as in a word processor.

Syntax:

```
[Abstract] Class Name Inherits ParentClassName

  [Static] Dim Name As Type
  [Static] Public Name As Type
  [Static] Protected Name As Type
  [Static] Private Name As Type
  Const Name As Type
  Public Const Name As Type
  Protected Const Name As Type
  Private Const Name As Type
  ...


  [Public | Protected | Private]
  Enum Name
    Name As Type
    ...
  End Enum
  ...


  [Public | Protected | Private]
  Type Name
    Name As Type
    ...
  End Type
  ...


  [Public | Protected | Private]
  Property Name As Type

    Get
      [Statements]
    End Get

    Set(Argument)
      [Statements]
    End Set

  End Property
  ...


  [Public | Protected | Private]
  Constructor Name([Arguments])
    [Statements]
  End Constructor
  ...


  [Public | Protected | Private]
  Destructor Name( )
    [Statements]
  End Destructor


  [Static] [Public | Protected | Private]
  Function Name([Arguments]) [As Type] [Throws Name, ...]
    [Statements]
  End Function
  ...


  [Static] [Public | Protected | Private]
  Sub Name([Arguments]) [Throws Name, ...]
    [Statements]
  End Sub
  ...


  [Public | Protected | Private]
  Slot Name([Arguments])
    [Statements]
  End Slot
  ...
```

```
  [Public | Protected | Private]
  Signal Name([Arguments])
    [Statements]
  End Signal
  ...

End Class
```

Bigger example:

```
Class Salsa Inherits Waltz

  Public Enum class_enum
     Entry
     Entry2
     Security = Entry
  End Enum

  Public Type class_type
    element As Integer
  End Type

  Const classConst = 4

  Public publicInstanceVar As Integer
  Private privateInstanceVar As Integer
  Protected protectedInstanceVar As Integer

  Static Public publicClassVar As Integer
  Dim publicModuleType As module1.module_type
  Dim publicModuleType2 As module_type

  ' parent constructor call inside constructor

  Sub meExplicit()
    Dim localVar = Me.publicInstanceVar ' it is the same with parent
    Dim localVar2 = Me.publicClassVar
    Dim localVar3 = Salsa.publicClassVar
    Dim localVar4 = Salsa.classConst
    Dim localVar5 = Me.classConst
'
    Dim localVar6 As class_enum
    localVar6 = Salsa.class_enum.Entry
'    Dim localVar7 As Me.class_enum   '  full type name not allowed
    Dim localVar8 As class_type
  End Sub

  Sub meImplicit()
    Dim localVar = publicInstanceVar
    Dim localVar2 = publicClassVar
    Dim localVar3 = classConst
    Dim localVar4 As class_enum
    Dim localVar5 As class_type

  End Sub

  Sub classSub()
    Const localConst = 6
    Dim n = localConst
  End Sub

  Sub classSubWithArgument(i As Integer)
    Dim localVar = i
  End Sub

  Function classFunction() As String
    Return "hello"
  End Function


  Static Public Sub test() Throws Waltz
    Throw New Waltz
  End Sub

  Private pvtFname As String

  Public Property Nickname As String

    Get
      Print "Hi"
    End Get

    Set ( ByVal Value As String )
      Print "Hi"
    End Set

  End Property
```

```
End Class
```

Module

A simple application can consist of only one form while the complete source code is in one form module. As your applications grow larger you probably would like to use the same code in different forms. To do so, place this code in a global module file as it is accessible by the entire application. You KBasic code is stored in classes or modules. You can archive your code within modules. Every module consists of the declaration part and the procedures you have inserted.

A module can contain:

- Declarations - for variables, types, enumerations and constants
- Procedures - which are not assigned to a special event or object. You can create as many procedures as you want, e.g. sub-procedures without return value or function-procedures.

You can put several classes or modules in one file but you should not.

Global Modules

Global modules are always present. You can access them from everywhere in your program. Put procedures you often use in forms or event-procedures within global modules.

Modules are not executed, unlike procedures are

A module consists of a declaration part only. You have to create the procedures for yourself. Additionally, a module contains no main program. Procedures are executed if another part of your program has called a procedure. Edit Module Editing your code in a word processor is no different than in KBasic Software IDE. You have a cursor showing the current position and you can use find and replace inside your source code as in a word processor.

Syntax:

```
Module Name

  Dim Name As Type
  Public Name As Type
  Private Name As Type
  Const Name As Type
  Public Const Name As Type
  Private Const Name As Type
  ...


  [Public | Private]
  Enum Name
    Name As Type
    ...
  End Enum
  ...


  [Public | Private]
  Type Name
    Name As Type
    ...
  End Type
  ...


  [Public | Private]
  Function Name([Arguments]) [As Type] [Throws Name, ...]
    [Statements]
  End Function
  ...


  [Public | Private]
  Sub Name([Arguments]) [Throws Name, ...]
    [Statements]
  End Sub
  ...
```

```
End Module
<code>



Bigger example:
<code>
Module module1

  Public Type address
    age As Integer
  End Type

  Public Type module_type
    element AS integer
  End Type

  Public Enum module_enum
     Entry
     Entry2
     Security = Entry
  End Enum

  Const moduleConst = 7

  Public publicModuleVar As Integer
  Private privateModuleVar As Integer


  Sub moduleExplicit()
    Dim localVar = module1.publicModuleVar
    Dim localVar2 = module1.moduleConst
    Dim localVar3 As module_enum
    localVar3 = module1.module_enum.Entry
  End Sub

  Sub moduleImplicit()
    Dim localVar = publicModuleVar
    Dim localVar2 = moduleConst
    Dim localVar3 As module_enum
    localVar3 = module_enum.Entry
    Dim localVar4 As module_type
  End Sub

  Sub moduleSubWithDefaultArgument(ko As Integer = 6)
    Dim localVar = ko
  End Sub

  Sub moduleSubWithOptionalArgument(Optional ko As Integer)
    If Not IsMissing(ko) Then
      Dim localVar = ko
    End If
  End Sub

  Sub moduleSub()
    Const localConst = 6
    Dim n = localConst
  End Sub

  Sub moduleSubWithArgument(i As Integer)
    Dim localVar = i
  End Sub

  Sub moduleSubWithArgumentShadowing(i2 As Integer)
    Dim localVar = i2
    Dim i2 = localVar + 99
    Dim i3 = i2
  End Sub

  Sub subOverloading ( )
    Print "sub1"
  End Sub

  Sub subOverloading ( i As Integer = 1)
    Print "sub2"
  End Sub

  Function moduleFunction() As String

    subOverloading()
    subOverloading(88)

    Return "hello"
  End Function

  Function moduleFunctionRecursive(ByRef i As Integer) As  Integer
    If i > 6 Then Return 1''i

    ''i = i + 1
```

```
    Return moduleFunctionRecursive(1)''i)
  End Function

End Module
```

Error handling

Developers in any language always wanted to write bug-free programs, programs that never crash, programs that can deal with any situation with grace and that can recover from unusual situations without causing the user any undue stress. Good intentions aside, programs like this do not exist. In real programs, errors occur either because the programmer did not anticipate every situation the code would get into or because of situations out of the programmer's control, bad data from users, corrupt fields that do not have the right data in them, and so on.

Three possible error sources exist:

- Error while compiling - These kind of errors occur when statements are written incorrectly. Maybe you wrote a keyword wrong, or misused a keyword in the wrong place, or you forgot to write all needed marks (braces, dots and so on).
- Runtime error - These kind of errors occur when KBasic finds an error when your program executes. An example of such an error is a division by zero, which is not allowed in any math expression.
- Logic error – Logic errors occur when syntax and runtime behavior is okay, but the program does not work as intended or expected. You can only be sure that your program works as you want if you test and analyze any result.

When a runtime error occurs, KBasic stops the execution of your program if no error handler has been defined in the place the error occurred. Old error handling syntax is supported like 'On Error GoTo,' but you should use the new error handling syntax (exceptions).

Exceptions

In KBasic, the strange events/errors that may cause a program to fail are called exceptions. Exception handling is an important feature of KBasic. An exception is a signal showing that a non-normal stage has been reached (like an error). To create an exception means to signal this special stage. To catch an exception is to handle an exception; performing some commands to deal with this special stage to return to a normal stage.

Example:

```
Public Sub tt2() Throws samba

  Try
    test()
  Catch (b As rumba)
    Print "tt2: got you!"
    b.dance()
  Finally
    Print "tt2: will be always executed, whatever happend"
  End Catch

End Sub
```

Exceptions are walking from the origin to the next control structure (caller of the current procedure). First it is tested if a 'Try-Catch'-statement or a throw statement has been defined in the current scope (procedure etc.). If not, KBasic looks up the caller of the current scope and tries again to find a 'Try-Catch'-statement or a 'Throw'- statement and so on. If it is not caught, KBasic shows the error to the user and the execution of the program stops. Exception-objects

An exception is an object that is an instance of a 'Object' or any other class. Because exceptions are objects, they can contain data and methods as any other class or object. Exception handling

The 'Try-Catch'-statement is needed to catch an exception. 'Try' encloses the normal code, which should run fine. 'Catch' contains the commands that should be executed if an exception has been raised. 'Finally' has some commands that should be executed whether an exception has happened or not.

```
Try
'Try' encloses the normal code, which should run fine. 'Try' does not work alone. It comes with at least one 'Catch'-stateme
Catch
```

After a 'Try' and its commands, define as many 'Catch'-statements as necessary. 'Catch'-statements are like a variable declaration, but the exception object is only created, then it matches.

Finally

'Finally' is useful when you have file accessing or database closing commands that must always execute even when an exception occurs. If you have defined a 'Catch'-statement and 'Finally'-statement and the matching exception is raised, first the 'Catch'-statement executes,then the 'Finally'-statement.

Declaration of exceptions

KBasic demands that every procedure that can raise an exception must define a exception with a 'Throw'-statement in its declaration. Example:

```
Sub mySub() Throws myException1, myException2
' statements which can raise myException1 or myException2
...
End Sub
```

Define and generate exceptions

Raise your own exceptions using the 'Throw'-statement. After 'Throw,' you must write the object to throw. You often create these object before throwing it, using the 'New'-statement: Throw New MyException1()

Syntax:

```
Throw ExceptionObject
```

If an exception is thrown, the normal program execution is left and KBasic tries to find a proper 'Catch'-statement for these exception. It searches all statements including the place the exception was thrown. All 'Finally'-Statements are executed on this path. To use exceptions is a creative way of dealing with errors; it is easy and clean to read. Often you can use a simple error object, but sometimes it is better to use a newly created error object. Syntax:

```
Try
  [Statements]
Catch (Name As Exception)
  [Statements]
End Catch

Try
  [Statements]
Catch (Name As Exception)
  [Statements]
Catch (Name As Exception)
  [Statements]
End Catch

Try
  [Statements]
Catch (Name As Exception)
  [Statements]
Finally
  [Statements]
End Catch
```

Complete example:

```
Class rumba

  Sub dance
    Print "rumba.dance"
  End Sub

End Class

Class samba

  Sub dance
    Print "samba.dance"
  End Sub
```

```
End Class


Public Sub test() Throws rumba, samba
  Throw New rumba  ' return rumba = new rumba
End Sub

Public Sub tt2() Throws samba

  Try
    test()
  Catch (b As rumba)
    Print "tt2: got you!"
    b.dance()
  Finally
    Print "tt2: will be always executed, whatever happend"
  End  Catch

End Sub


Public Sub tt()

  tt2()

Catch (c As samba)
  Print "tt: got you!"
  c.dance()
Finally
  Print "tt: will be always executed, whatever happend"
End Sub


tt()
```

Exception at the end of a procedure

It is possible to catch an exception at the end of a procedure, which covers the whole procedure. It is easier to read the code and is something like the old 'On Error GoTo.' Write down the needed catch and finally statements. A try statement is not needed because the whole procedure is meant.

Example:

```
Public Sub tt()

  tt2()

Catch (c As samba)
  Print "tt: got you!"
  c.dance()
Finally
  Print "tt: will be always executed, whatever happend"
End  Sub
```

Syntax:

```
Catch (Name As Exception)
  [Statements]
Finally
  [Statements]
End Catch
```

The KBasic development environment

The KBasic development environment contains windows, toolbars, and editors that make developing your KBasic application easy. It is actually an integrated development environment (IDE). When a development environment is said to be integrated, the tools in the environment work together. For example, the compiler might find an error in the source code. In addition to displaying the error, KBasic opens the source file in the text editor and jumps to the exact line in the source code where the error occurred. A large part of application development involves adding and arranging controls on forms. KBasic provides tools to make designing forms a simple process.

Windows

KBasic windows are used to develop your programs, including monitoring the status of your projects at any time.

These windows include:

- Form designer – uses drag and drop controls, as well as arranges them in this main development area. It is the primary tool you use to create your programs.
- Project window – works with the different parts of your project in the project window. Its views include objects and files.
- Property window – lists and lets you control the properties for your controls. You can resize, name, change visibility, assign values, and change colors and fonts of your controls.
- Toolbox window – the central place where often used controls can be accessed
- Source code editor – where you add and customize source code for your project in any phase of the development cycle.

Toolbars

KBasic provides an extensive set of toolbars. Toolbars can be docked in the KBasic window or floated on the screen.

Editor

KBasic has one editor for creating and managing KBasic projects. You can use this editor to control the development of your projects, such as editing source code and manipulating classes. The source code editor is a tool for editing source code.

Debugger

One of the most powerful tools in the KBasic environment is the integrated debugger (only in the professional version). The debugger allows you to watch your programs execute line by line. As your program executes, you can observe the various components of the program to see how they are behaving. By using the debugger, you can monitor: the values stored in variables which subs/functions/methods are called the order in which program events occur

Classes and objects of KBasic

There are two types of objects in KBasic: visual objects and invisable objects. A visual object is a control, is visible at runtime, and lets users interact with your application. It has a screen position, a size, and a foreground color. Examples of visual objects are forms and buttons. An invisible object is not visible at runtime, such as a timer. Some objects can contain other components, such as an application window containing a button. With KBasic, you add visual objects/controls to your forms to assemble applications.

Projects

Projects keep your work together. When developing an application in KBasic, you work mainly with projects. A project is a collection of files that make up your KBasic application. You create a project to manage and organize these files. KBasic provides an easy yet sophisticated system to manage the collection of files making up a project. The project window shows each item in a project. Starting a new application with KBasic begins with the creation of a project. So before you can construct an application with KBasic, you need to create a new project. A project consists of many separate files collected in one project directory, where one *.kbasic_project file is and many other files:

- *.kbasic_module
- *.kbasic_class
- *.kbasic_form

Forms

In your Kbasic-application, forms are not only masks for inputting and changing data but they are the graphical interface of your application. In the eyes of the beholder, they are the application! By creating your application using forms, you control the program flow with events that are raised in the forms.

Central meaning of forms

Each form in a KBasic application has a form module with event procedures. Those event procedures react to events raised in the form. Additionally, every module can contain other non-event procedures. A form module is part of every form. When you copy a form, its form module is automatically copied, too. If you delete a form, its form module is deleted as well. KBasic creates a form module automatically, so you need only to write the event procedures and other procedures. Forms hold your KBasic program together!

Procedure in forms

The procedures in form modules of a form are private procedures of this form; you can use them only inside the form module. Because the procedures are private inside a form module, you can use the equally named procedures in many forms. But the names of procedures in global modules must be equal. It is possible to create a procedure in a form module that has the same name as a procedure in a global module. In this case, KBasic uses two rules to recognize the correct procedure to execute:

KBasic searches the current form module at first

If KBasic does not find a procedure with the needed name, it searches the global modules (but not the other form modules) for the needed procedure name.

All calls inside a module in which the procedure is declared reaches this procedure. Calls outside the module run the public procedure. So all calls inside a module, in which the procedure is declared, reaches this procedure. Calls outside the module run the public procedure.

Compatibility VB6 = KBasic

KBasic supports 100% of the syntax of VB6. Many components are equal as well. It is possible to develop GUI applications with well-known BASIC syntax in a modern fashion. KBasic comes with truly Java-like object orientation and backward support for VB6 and QBasic, as it is 100% syntax compatible. KBasic combines the expressive power of object-oriented languages like C++ with the familiarity and ease of use of VB6. It allows developers with an installed base of VB6 applications to start developing for a mixed Windows, Mac OS X, and Linux environment without having to face a steep learning curve: KBasic uses the familiar visual design paradigm and has a full implementation of the BASIC language.

Migrating from VB6 to KBasic

Some information which might help you to migrate are listed in the following:

- do not use ( ) to access arrays, better use [ ]
- do not use 'Option OldBasic' or 'Option VeryOldBasic'
- do not use 'On Error Goto', better use 'Try Catch'
- do not use 'Nothing', better use 'Null'
- avoid the use of the data type 'Variant'
- do not use 'class_initialize', better use 'Constructor', the same for the destructor
- do not use 'Call' when calling a sub or function
- do not use 'Optional' and 'IsMissing' with arguments in subs or functions, better use the default value of an argument
- use always 'Do While…Loop' and 'Do …Loop While' instead of the other loops
- always write many comments in your source code
- use in conditions 'AndAlso' and 'OrElse' instead of the binary operators 'And' and 'Or'
- avoid the use of 'ByRef' with primitive data types to get faster execution speed
- do not use 'Data' and 'Def*', like 'DefInt'
- use enumerations instead of many integer constants
- use constants, instead of the use of numeric literals many times in your source code
- avoid the use of 'GoSub', better to use real functions or real subs
- avoid the use of 'GoTo', better to use loops and other control flow language elements

- ‘Let’ and ‘Set’ are not needed
- use ‘For Each’, when it is possible
- use ‘For i As Integer = 0 To 10 Next’, instead of declaring the counter variable outside of the loop
- name the variable names without giving them suffixes
- use always ( ) when you call a sub or function

KBasic's virtual machine

The KBasic virtual machine is the environment in which KBasic programs are executed. It is a kind of abstract computer and defines the commands a KBasic program can use. These commands are named p-code or pseudo code. In general this means KBasic-pcode is the same for the virtual machine as a machine code statement is for the CPU. A p-code is a command with length of 2 bytes generated by the KBasic-compiler and interpreted by the KBasic-interpereter. When the compiler compiles a KBasic program it produces a set of p-codes and saves them. The KBasic-interpreter executes these pseudo-codes. It is important to know that the name ‘KBasic’ means much more than a programming language. It means also a complete computer environment. KBasic contains two components: KBasic design (the programming language) and KBasic runtime (the virtual machine).

What are the main functions of the virtual machine?

The virtual machine (VM) does the following tasks:

- Assign allocated memory to created objects
- De-allocate memory automatically
- Handle the stack and registry of variables
- Access the host system, like using devices

Appendix

Argument

A value that is passed to a procedure or function. Also see Parameter.

Arithmetic Operations

Mathematical operations such as addition, multiplication, subtraction, and division that produce numerical results.

Array

A variable that stores a series of values accessed using a subscript.

BASIC

Beginner’s All-Purpose Symbolic Instruction Code, the computer language upon which KBasic is based.

Bit

The smallest piece of information a computer can hold. A bit can be only one of two values, 0 or 1.

Boolean

A data type representing a value of true or false.

Boolean Expression

A combination of terms that evaluates to a value of true or false. For example, (x = 5) and (y = 6).

Branching

When program execution jumps from one point in a program to another, rather than continuing to execute instructions in strict order. Also see Conditional Branch and Unconditional Branch.

Breakpoint

A point in a program at which program execution should be suspended to enable the programmer to examine the results up to that point. By setting breakpoints in a program, you can more easily find errors in your programs. Also see Debugging.

Byte

A piece of data made up of eight bits. Also see Bit.

Compiler

A programming tool that converts a program's source code into an executable file. KBasic automatically employs a compiler when you run a program or select the File menu's Make command to convert the program to an executable file (only Professional Version). Also see Interpreter.

Concatenate

To join end to end two text strings into one text string. For example, the string "OneTwo" is a concatenation of the two strings "One" and "Two."

Conditional Branch

When program execution branches to another location in the program based on some sort of condition. An example of a conditional branch is an If/Then statement that causes the program to branch to a program line based on a condition such as If (x=5).

Constant

A predefined value that never changes.

Data Type

The various types of values that a program can store. These values include Integer, Long, String, Single, Double, and Boolean.

Debugging

The act of finding and eliminating errors in a program.

Decrement

Decreasing the value of a variable, usually by 1. Also see Increment.

Double

The data type that represents the most accurate floating-point value, also known as a double-precision floating-point value. Also see Floating Point and Single.

Empty String

A string that has a length of 0, denoted in a program by two double quotes. For example, the following example sets a variable called str1 to an empty string: str1 = “”.

Event

A message that is sent to a program as a result of some interaction between the user and the program.

Executable File

A file, usually an application, that the computer can load and run. Most executable files end with the file extension .EXE on Windows.

File

A named set of data on a disk.

Floating Point

A numerical value that has a decimal portion. For example, 12.75 and 235.7584 are floating-point values. Also see Double and Single.

Form

The KBasic object that represents an application’s window. The form is the container on which you place the controls making up your program’s user interface.

Function

A subprogram that processes data in some way and returns a single value representing the result of the processing.

Global Variable

A named value accessed from anywhere within a program module.

Increment

Increasing the value of a variable, usually by 1. Also see Decrement.

Infinite Loop

A loop that can’t end because its conditional expression can never evaluate to true. An infinite loop ends only when the user terminates the program. Also see Loop and Loop Control Variable.

Initialize

Setting the initial value of a variable.

Integer

A data type representing whole numbers between –2,147,483,648 to 2,147,483,647. The values 240, –128, and 2 are examples of integers. Also see Long.

Interpreter

A programming tool that executes source code one line at a time unlike a compiler that converts an entire program

to an executable file before executing any of the program's commands. Also see Compiler.

Literal

A value in a program that is stated literally. That is, the value is not stored in a variable.

Local Variable

A variable accessed only from within the subprogram in which it is declared. Also see Global Variable and Variable Scope.

Logic Error

A programming error that results when a program performs a different task than the programmer thought he programmed it to perform. For example, the program line If X = 5 Then Y = 6 is a logical error if the variable X can never equal 5. Also see Runtime Error.

Logical Operator

A symbol comparing two expressions and resulting in a Boolean value (a value of true or false). For example, in the line if X = 5 and Y = 10 then Z = 1, and is the logical operator. Also see Relational Operator.

Long

A data type that represents integer values from -2^{32} - 1 to $+2^{32}$. Also see Integer.

Loop

A block of source code that executes repeatedly until a certain condition is met.

Loop Control Variable

A variable holding the value that determines whether a loop continues to execute.

Machine Language

The only language a computer truly understands. All program source code must be converted to machine language before the computer can run the program.

Mathematical Expressions

A set of terms using arithmetic operators to produce a numerical value. For example, the terms (X + 17) / (Y + 22) make up a mathematical expression. Also see Arithmetic Operations.

Method

A procedure associated with an object or control that represents an ability of the object or control. For example, a Command Button's Move method repositions the button on the form.

Numerical Literal

A literal value that represents a number, such as 125 or 34.87. Also see Literal and String Literal.

Numerical Value

A value that represents a number. This value can be a literal, a variable, or the result of an arithmetic operation.

Object

Generally, any piece of data in a program. Specifically in KBasic, a set of properties and methods that represent some sort of real-world object or abstract idea.

Order of Operations

The order in which KBasic resolves arithmetic operations. For example, in the mathematical expression (X + 5) / (Y + 2), KBasic performs the two additions before the division. If the parentheses had been left off, such as in the expression X + 5 / Y + 2, KBasic would first divide 5 by Y, then perform the remaining addition operations.

Parameter

Often meaning the same as "argument," although some people differentiate argument and parameter where an argument is the value sent to a procedure or function and a parameter is the variable in the function or procedure that receives the argument. Also see Argument.

Procedure

A subprogram performing a task in a program but does not return a value. Also see Function.

Program

A list of instructions for a computer.

Programming Language

A set of English-like keywords and symbols that enable a programmer to write a program without using machine language.

Program Flow

The order in which a computer executes program statements.

Property

A value representing an attribute of an object or control.

Read-Only Property

A property whose value cannot be changed from within a program. However, a program can retrieve (read) a property's value.

Relational Operator

A symbol that determines the relationship between two expressions. For example, in the expression X > 10, the relational operator is >, which means "greater than." Also see Logical Operator.

Return Value

The value a function sends back to the statement that called the function. Also see Function.

Runtime Error

An system error that occurs while a program is running. An example is a divide-by-zero error or a type-mismatch error. Without error handling such as that provided by the Try/Catch statement, runtime errors often result in a program crash.

Scope

See Variable Scope.

Single

The data type that represents the least accurate floating-point value, also known as a single-precision floating-point value. Also see Double and Floating Point.

Source Code

The lines of commands making up a program.

String

A data type that represents one or more text characters. For example, in the assignment statement str1 = "I'm a string," the variable str1 must be of the String (or Variant) data type.

String Literal

One or more text characters enclosed in double quotes.

Subprogram

A block of source code that performs a specific part of a larger task. In KBasic, a subprogram can be either a procedure or a function. Also see Function and Procedure.

Unconditional Branch

When program execution branches to another location regardless of any conditions. An example of an unconditional branch is the GoTo statement.

User Interface

The visible representation of an application, usually made up of various types of controls, that enables the user to interact with the application.

Variable

A named value in a program. This value can be assigned and reassigned a value of the appropriate data type.

Variable Scope

The area of a program in which a variable can be accessed. For example, the scope of a global variable is anywhere within the program, whereas the scope of a local variable is limited to the procedure or function that declares the variable. Also see Global Variable and Local Variable.

Variant

A special data type that can represent any type of data. More accurately, KBasic manages the data type of a Variant value for you. This data type is very inefficient and should be avoided when possible.

Contact/Impressum

www.kbasic.com

email: info@kbasic.com

Quality by Bernd Noetscher

Made in Germany (European Union)

Trademarks