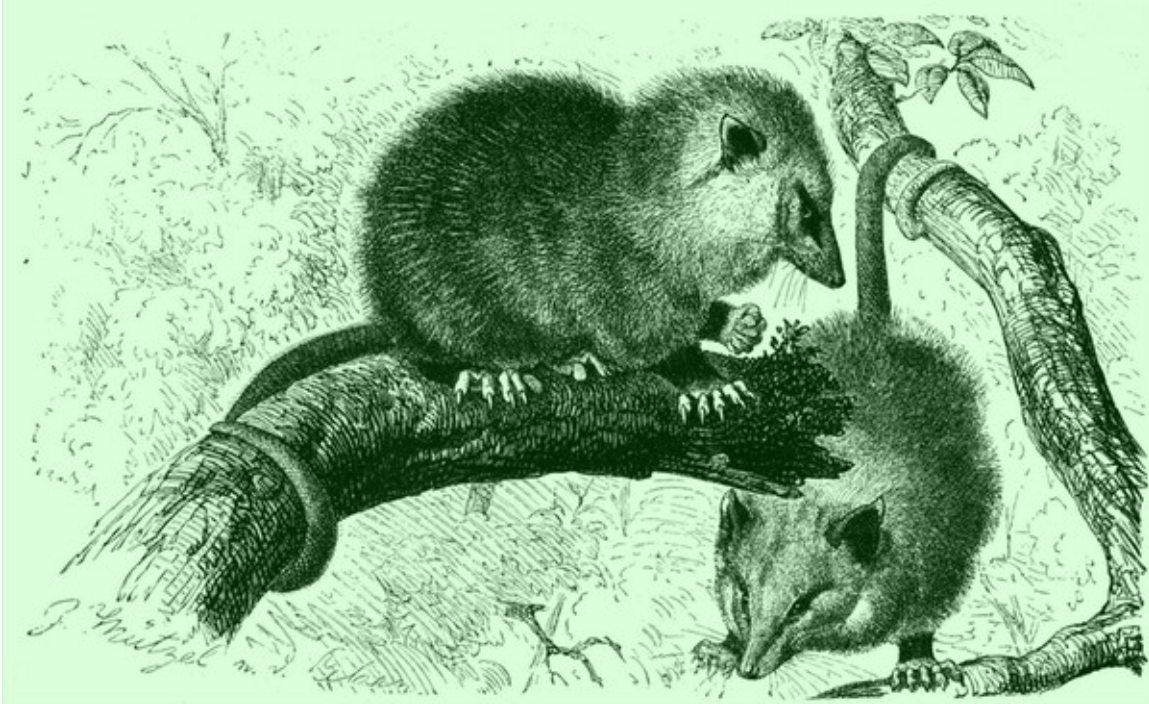


Programming with yab

Michel Clasquin-Johnson



PROGRAMMING WITH YAB

Published on Smashwords by Michel Clasquin-Johnson

© 2017 Michel Clasquin-Johnson

ISBN: 9781370847495

Copyright notes for the 2nd Smashwords edition

“Programming with yab” is published exclusively at Smashwords as a FREE e-book, and will always remain free. I encourage you to share copies with your friends. Code samples in the e-book are hereby placed in the Public Domain.

However, if you have obtained your copy from someone else and enjoyed it, please consider signing up with Smashwords and downloading an official copy. That way, you will be notified by email whenever a new edition comes out. This e-book is intended to be a living document. as Haiku and yab grow and develop, the e-book will also be augmented and rewritten.

About the cover image

There is a strange convention that the cover of a programming e-book must show an animal. I think O'Reilly started it, but other publishers are doing it too. So my choice is the opossum. An ancient and rather primitive animal, but its unspecialised nature and ability to eat almost anything has enabled it to survive and even thrive in the face of more modern opposition. Which reminds me of a certain programming language

Image credit: Public domain picture from Wikimedia Commons.

Table of Contents

Chapter 0: Introduction

- 0.1 Why yab
- 0.2 Why not yab?
- 0.3 Installing Haiku
- 0.4 Installing yab
- 0.5 How to learn yab

Chapter 1: Hello, World!

- 1.1 Hello, Terminal!
- 1.2 Hello, .BAS file!
- 1.3 Hello, Alert!
 - Exercise 1
- 1.4 Hello, yab!
 - Exercise 2
- 1.5 Hello, Bling-bling!
- 1.6 Running the app
 - Exercise 3
- 1.7 Bonus section: Hello, Dolly!

Chapter 2: Shortening a URL

- 2.1 Non-interactive URL shortening
 - 2.1.1 The program
 - 2.1.2 What's new?
 - 2.1.2.1 Switching!

2.1.2.2 Iffy business

2.1.2.3 A matter of routine

Exercise 4

2.2 Interactive URL shortening

2.2.1 What's new?

Exercise 5

2.3 Batch URL shortening

Exercise 6

2.3.1 What's new?

2.3.2 Next, please!

Exercise 7

2.4 URL shortening in a GUI

2.4.1 What's new?

2.5 Final thoughts on URL shortening

Solutions to Exercise 3

Solutions to Exercise 4

Solutions to Exercise 6

Chapter 3: The yab IDE

Chapter 4: The Yabadabbadoo IDE

4.1 Introduction

4.2 Yabadabbadoo Tutorial

4.2.1 First Steps

4.2.2 Changing the window title

4.2.3 Creating a text area

4.2.4 Pimping the menu

4.2.5 Activating the editing functions

4.2.6 Saving a file

4.2.7 Starting a new file

[4.2.8 Opening a file](#)

[4.3 Wrapping up your project](#)

[4.4 Wrapping up your project - part two](#)

[4.5 Teditor - full code](#)

[4.6 Yabadabbadoo FAQ](#)

[Chapter 5: Learning more about yab](#)

[5.1 The BeSly database](#)

[5.2 The yab discussion forum](#)

[About the author](#)

[About this e-book](#)

[Appendix 1: Database-like functions in yab](#)

[Appendix 2: Preparing yab apps for distribution](#)

[Appendix 3: Why you should be writing apps for Haiku](#)

[Appendix 4: A note about arrays](#)

[Appendix 5: Using a treebox](#)

[Appendix 6: Full-screen programs](#)

[Glossary of yab commands](#)

[Glossary of handy PEEKs](#)

Chapter 0: Introduction

This e-book will teach you how to program in the *yab* programming language. The only contemporary platform on which yab runs is the Haiku Operating System (<http://www.haiku-os.org>), so I will assume that you have installed Haiku - a recent version with package management, not the ancient alpha4.1 release - and that you are familiar with its basic elements like the Tracker and the Deskbar. We are not going to waste several chapters on "what is Haiku?" or "what is a file?" If you are interested, skip ahead to *Appendix 3: Why you should be writing software for Haiku*. But we do need to talk a little bit about where yab comes from.

Yab is a dialect of BASIC, which is an acronym for Beginners' All-purpose Symbolic Instruction Code. The BASIC language was invented in 1964 by John Kemeny and Thomas Kurtz, and was based on two older computer languages, Fortran and Algol.

That original form of BASIC later became known as *Dartmouth Basic*, after the college where Kemeny and Kurtz worked when they developed the language. But as computers became more commonplace, other people started to develop their own dialects of BASIC to work on different machines. Experienced BASIC programmers will look at my code and immediately see that I cut my teeth on Sinclair BASIC, for example.

One dialect that was developed around 1995 was *yabasic* or "yet another basic", which was available for Windows and Linux. Its creator, Marc-Oliver Ihm, never saw it as anything other than a hobby project, but yabasic came with some pretty advanced features that set it apart from other dialects. The most important of these was that although yabasic was still an interpreted language (in other words, it needed both the yabasic program itself and the script that the user wrote) the interpreter and the script could be glued together (in the yabasic world this is called "binding") so that you could distribute your program as a single file. No longer did you have to tell your user "first install yabasic, then run this program". In the long run, this was not sustainable on Haiku, as we shall see.

Yabasic is still [available](#), and if your program runs in text mode and uses none of the Haiku additions, you should be able to load up your yab script into yabasic and fire it right up in Windows or Linux! Perhaps something to keep in mind....

Yabasic was basically a language for writing text-based applications, There was a "graphics mode" that could draw simple circles and squares in a separate window, but that never really took off. Then Jan Bungeroth took the language and ported it to the Be Operating System (BeOS). He added "hooks" into the BeOS system so that it could draw windows and buttons and respond to system messages, and renamed it yab. The copyright notice says this happened in 2006. I seem to remember playing with early versions of yab quite a bit before that, but no matter, really. Yab was later ported to Zeta OS and then to Haiku. The current maintainer of yab is Jim Saxton (bbjimmy).

0.1 Why yab?

Why should you learn to write in yab? Well, yab is still recognisably a descendant of BASIC, a language that was designed from the beginning to be human-friendly rather than computer-friendly. What would you expect this line to do?

```
print "Hello, World!"
```

It puts the phrase *Hello, World* on the screen. Simple. BASIC syntax is *almost* like reading English, which is something nobody has ever claimed for C++, Lisp or Forth.

Or, at least that is true for yab. Dialects like Microsoft Visual Basic have gone in for "object orientation" in a big way. The result is that their code now resembles other computer languages more than good old BASIC. Here is a little yabasic/yab program to print out the lyrics to the song "99 bottles of beer"

```
b = 99
while(b > 0)
  print b, " bottles of beer on the wall,"
  print b, " bottles of beer."
  print "Take one down, pass it around."
  b = b-1
  print b, " bottles of beer of beer on the wall."
  print
wend
```

OK, If you haven't programmed before that will look odd, but you can sort of understand it. It's not completely outrageous. How would that look in Visual Basic.NET?

Step 1: Create a new Console Application and then add the following code in the principal module:

```
Module Bottles
  Sub Main()
    Dim Count As Int32
    For Count = 99 to 1 step -1
      Console.WriteLine(Count & " bottles of beer on the wall, " &
        Count & " bottles of beer.")
      Console.WriteLine("Take one down, pass it around, " & (Count -
        1) & " bottles of beer on the wall.")
    Next
  End Sub
End Module
```

That is still BASIC, but it is clearly more computer-friendly than human-friendly. Don't even get me started on what that program would look like in other languages. If you are interested, they are all listed on 99-bottles-of-beer.net.

The joy of yab is that it remains a *procedural* language, in which instructions are followed one by one, starting at the top. Despite that, you can do things with it for which you would normally require a much more complex *object-oriented* language. The object orientation is all hidden in the operating system, and the programming language simply

accesses it. Once you become used to this, it is quite addictive.

For example, with yab you can create a basic notepad in about twenty lines of code. When you right-click on that notepad, you will see the options to cut, copy and paste text in a context menu, even though you *never programmed it in*. It is simply a necessary part of what every text editing area in Haiku needs to be able to do, and every text editing area inherits that ability. This is object orientation, but you, the yab programmer, don't need to worry about the convoluted code it actually requires.

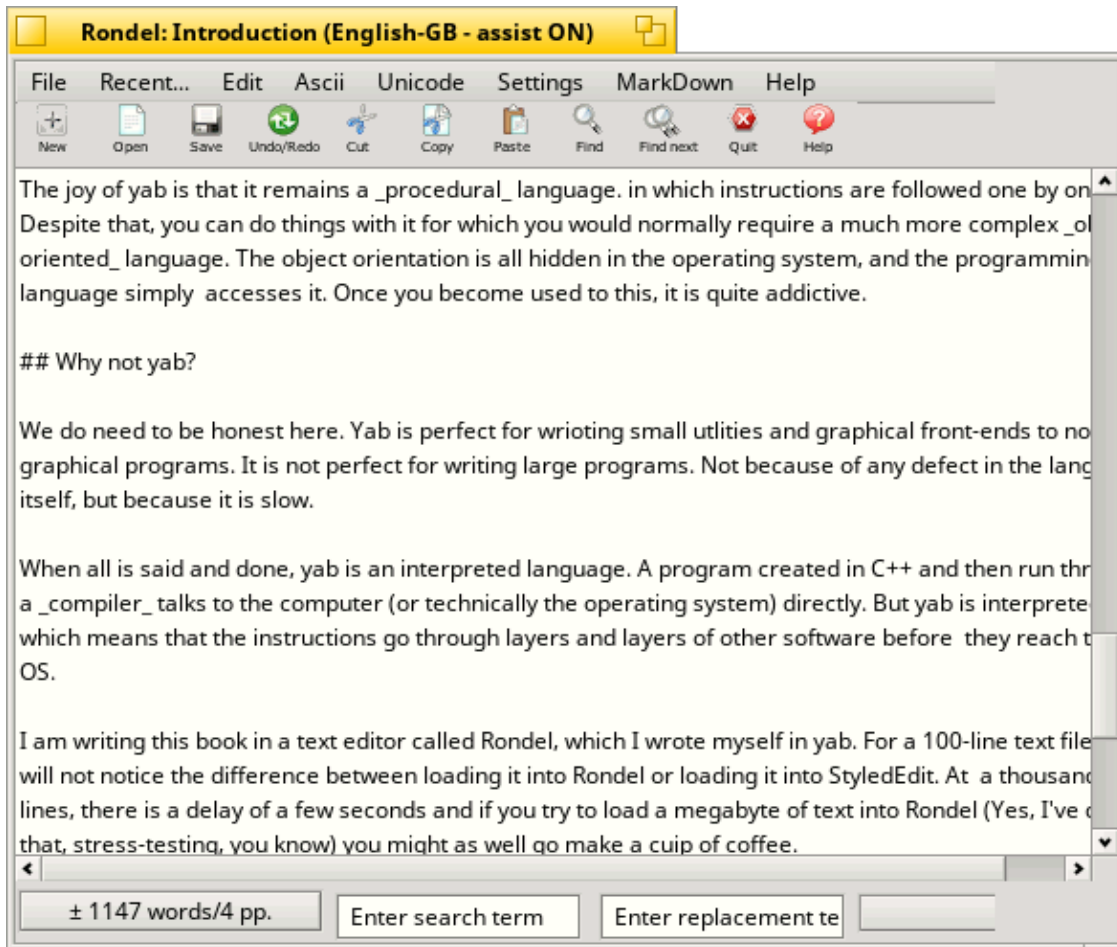
This makes yab a unique programming experience. There are languages on other platforms that come close. Applescript, for example. But nothing is quite like yab.

0.2 Why not yab?

We do need to be honest here. Yab is perfect for writing small utilities and graphical front-ends to non-graphical programs. It is not perfect for writing large programs. Not because of any defect in the language itself, but because it is slow.

When all is said and done, yab is an interpreted language. A program created in C++ and then run through a *compiler* talks to the computer (or technically the operating system) directly. But yab is interpreted, which means that the instructions go through layers and layers of other software before they reach the OS.

I am writing this e-book in a text editor called Rondel, which I wrote myself in yab. For a 100-line text file, you will not notice the difference between loading it into Rondel or loading it into StyledEdit. At a thousand lines, there is a delay of a few seconds and if you try to load a megabyte of text into Rondel (Yes, I've done that, stress-testing, you know. It was, in fact, Tolstoy's War and Peace), you might as well go make a cup of coffee. So you are not going to write the Haiku equivalent of Microsoft Word using yab.



But honestly, how often do you want to load a megabyte of data into a text editor? There are millions of great *little* apps waiting to be written in yab.

Yab is also not the best choice for writing graphics-heavy programs like games or bitmap editors. Yes, it can be done. Jim Saxton's FatElk repository contains a fun little game called Loopdloop. But that is a *little* game, not a first-person shooter, or a flight simulator.

The best use for yab is to create a graphical front-end for something you would otherwise do in Terminal. For example: I once found out that you can access the tinyurl.com service from Terminal to turn a long URL into a short one. So I wrote a program called TinyTim to make things a little easier. We will be dissecting TinyTim later in the e-book.

0.3 Installing Haiku

First of all, you need a Haiku OS installation. Yab is not available on all hardware platforms, so stick with the officially supported version, which right now is *x86_gcc2_hybrid*. That's quite a mouthful. All *x86* really means is that it will run on any

32-bit or better PC with an Intel CPU. That means anything from a 25-year old Pentium to today's hot multi-core i7 Xenon server. *gcc2* means that it runs version 2 of the GNU Compiler Collection and *hybrid* means that it can also run software written for version 4 (or 5) of the GCC. Don't worry too much about it; just get the officially supported version. That's where all the software is right now. Yab is not even available on some of the others. We have managed to get a yab runtime on the *x86_64* platform, but not the entire development package.

At the time of writing, the last official release of Haiku-OS is Alpha 4.1. Avoid it. You want a more recent version with Package Management enabled. If Alpha 5 or Beta 1 is out by the time you read this, install that. Otherwise, follow the links on the Haiku website and install a nightly build.

Installing, updating and maintaining Haiku is beyond the scope of this e-book. But there is one issue to consider: Raw silicon or virtual machine?

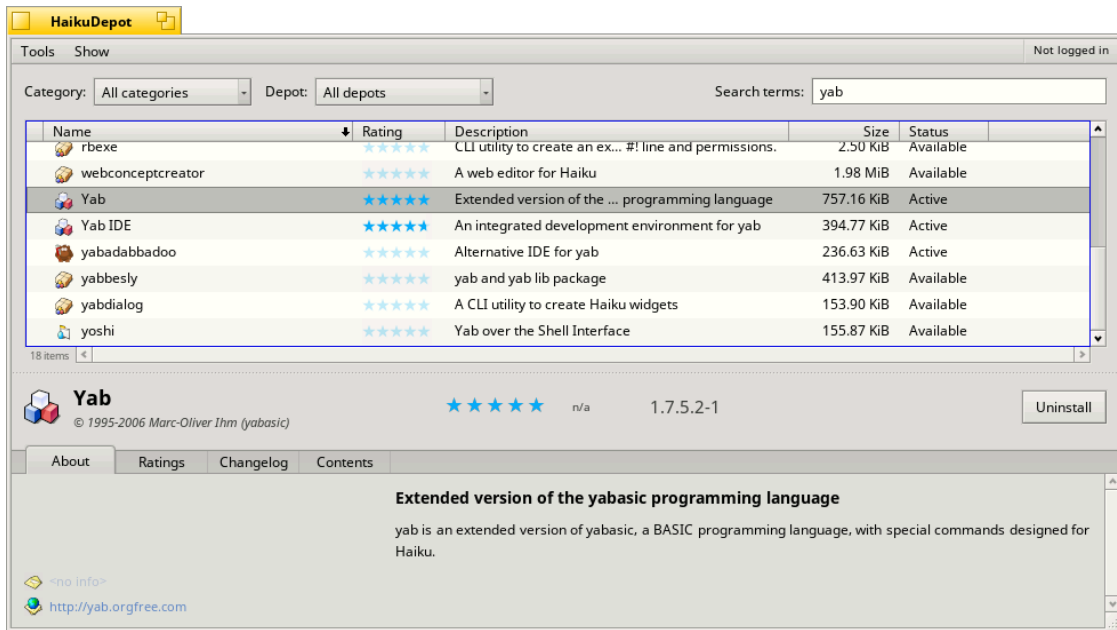
A lot of people run Haiku inside virtual machines, especially within VirtualBox. There are even people hacking on parts of the OS itself in this way, or so I've heard. In my experience, Haiku can be set up to run quite well under VirtualBox if your host operating system is Windows or MacOS. In my experience it runs dog-slow if the host OS is Linux, but maybe there is some optimisation that I missed. In any case, yab is a high-level language - it does not address the hardware directly, but always politely asks the operating system to do the heavy lifting. So there is no reason not to learn yab that way.

But I have a soft spot for running an operating system the way it was meant to be run: from a real partition on a real hard drive (or solid-state drive, these days). That is how I run Haiku and how I develop in yab. Keep that in mind as you read on, it may affect some of what I have to say.

Once you have Haiku with Package Management set up, and your internet link is up and running, you are ready to install yab.

0.4 Installing yab

Start up Haiku. In the Deskbar menu, look for the HaikuDepot app and run it. In the search area at the top left, type yab and press Enter. If you don't see yab listed, you may need to go to HaikuDepot's "Show" menu and make sure that "Available packages" is listed.



Alternatively, open a Terminal and type

```
pkgman install yab yab_ide
```

That is enough to get started with. You won't need the `yab_ide` package for a while, but go ahead and install it anyway. It will install a large-ish working directory at `/boot/home/yab_work`. Inside that directory you will find a number of example programs, some of which I am shamelessly going to use in this e-book. When you are done with the e-book, go through the example files I did not refer to and see how the techniques in the e-book are used there.

Further documentation can be found in two files: `/boot/system/apps/yab-IDE/Documentation/yabasic.html`, and `/boot/system/apps/yab-IDE/Documentation/yab-Commands`

The HTML file deals ONLY with the non-graphical yabasic core of the language while the text file exclusively contains information on the commands added when yabasic became yab. Put links to those files on your desktop. Refer to them often.

0.5 How to learn yab

In this e-book we will start from the presupposition that the best way to learn something is to actually do it. So each section will present an actual working program, not isolated snippets of code, and discuss the new features in that program.

Chapter One will bring you the oldest tradition in computing history: *Hello World*. If you have some programming experience you might be tempted to skip it. Don't. That chapter will already contain some points about yab that will not be repeated.

Then we will get ourselves a good grounding in text-mode apps before we start worrying about writing any graphical programs. Again, don't skip these sections. These chapters are where we will explore the yabasic guts of yab, what kinds of loops it allows, how it handles data, and so on.

Only after all that will we start to create real Haiku apps with clickable menus and buttons. Trust me, it will be worth the wait.

This is part of my programming philosophy. Nowadays, there are lots of languages available, though not for Haiku, that lets you "paint" the interface first and then supposedly just drop in the code to make it work.

I don't agree with that. My approach is that I first write the bit that does the actual work, the absolute minimum that this program must be able to do. Then I proceed to write the bells and whistles around it. The other aspect of my programming philosophy is that I try to keep the number of supporting files to an absolute minimum. Ideally, a program should consist of a single executable that can be put anywhere. This comes from my BeOS days and I'll admit that it is a little archaic now that we have package management.

There are two IDEs (Integrated Development Environments) available for yab. I like and use both (actually I wrote one of them), but you should not do so while you are learning to use the language. There is an essential purity about learning to control your computer with nothing but a Terminal and a text editor. When you have gone through that process, you will actually know what happens behind the scenes when you click on "Run" in the IDE's toolbar. Don't deprive yourself of that. We will deal with both IDEs in chapter 3 and 4.

There will be a series of appendices with more technical information, code snippets to illustrate advanced techniques, and who knows what else? The appendices form the section most likely to grow and expand with successive editions of this e-book. I don't want to renumber the appendices with each new edition, so they are listed simply as they occurred to me. Each one is rated as Beginner, Intermediate or Expert, so take note of that, and some may be written by other yab fanatics. Finally, there will be glossaries of the yab commands and tweaks.

When you see code, which will be formatted

```
like this
```

You should type that into your text editor, either as a new file, or as an addition to an old one. Don't cut and paste it. Type it in. That really is the only way to learn how a computer language works. You need to get the feel for it in your fingertips, until muscle memory no longer allows you to make errors. Well, not *too* often, anyway! As Zed A Shaw writes in his famous book "Python the Hard Way":

You must type each of these exercises in, manually. If you copy and paste, you might as well not even do them. The point of these exercises is to train your hands, your brain, and your mind in how to read, write, and see code. If you copy-paste, you are cheating yourself out of the effectiveness of the lessons.

But sometimes I will be showing you code (or pseudo-code) just as a demonstration, not for you to type in. Such snippets will be indented

```
like this
```

In this e-book I will use the terms "commands", "functions" and "keywords" indiscriminately, to the horror of Computer Science graduates everywhere. Commands will be indicated in CAPITAL LETTERS in the main text, but not in code sections. I will also use the word "loop" to include decision tree structures. That's just part of BASIC culture.

We will be installing some other software from my repository as we go along, so if you haven't done so already, open a Terminal and type the following command:

```
pkgman add-repo http://clasquin-johnson.co.za/michel/repo
```

This e-book aims to give you a good introduction to programming in yab. But it is not going to be a *complete* introduction. It will cover the most important things you need to start writing yab applications, but it will not cover every single command in the glossary. I have been programming in yab for many years, but there are features that I know about but that I've never needed for the kinds of program I like to write, and I constantly see other people using features I did not even know existed! In fact, as I am writing this e-book, I keep seeing things and thinking "Hey, I should incorporate that into Rondel...". Programming in any language is an on-going journey of discovery. Programming in yab is like having a friendly dog along for the trip, always eager to please.

You are now ready to start programming with yab. Let's go!

Chapter 1: Hello, World!

There is a long tradition in the computer world that your first program in any language should do just one thing: put the words "Hello, World" on the screen. I see no reason why yab should be any different. But yab can do it in a number of ways.

1.1 Hello, Terminal!

First, let's try a direct command. Open a Terminal and type:

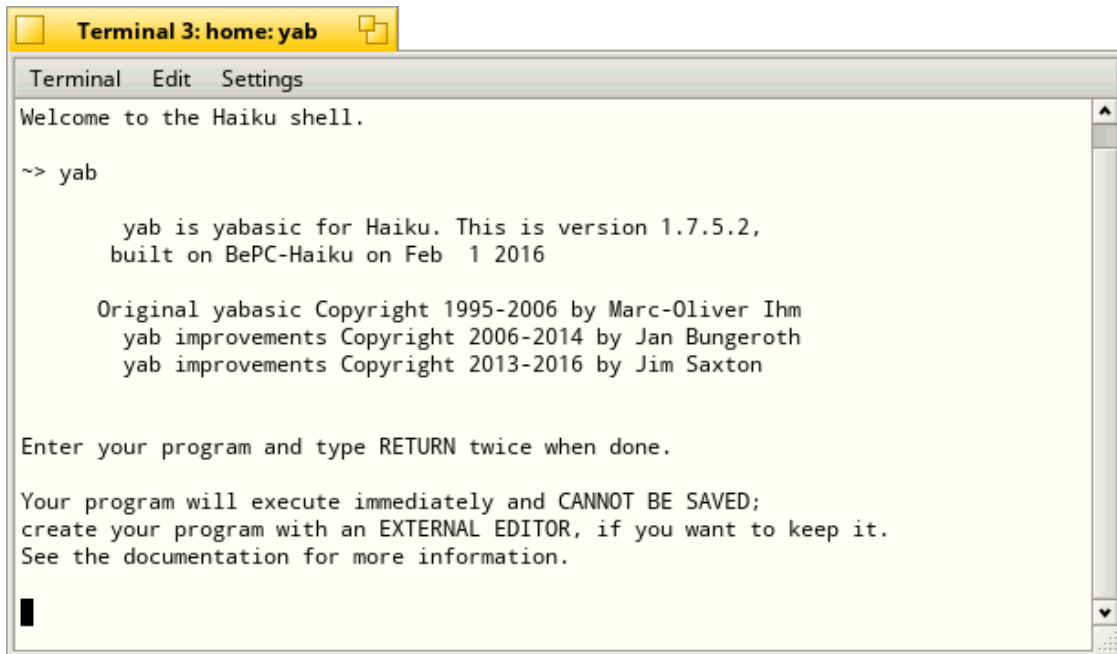
```
yab -execute 'print "Hello, World" '
```

If you had all the single and double quotation marks in place, you should see the phrase "Hello World" in your Terminal. I put a space at the end, between the last double quotation mark and the last single quotation mark, but that was just to show you which was which. That space is not really needed.

Let's try a more interactive approach. in your Terminal, type

```
yab
```

You should now see something like this:



OK, OK, we got it. Type the following line, then press the ENTER key twice:

```
print "Hello, World!"
```

yab runs your one-line program (it prints the message you just told it to print) and exits. PRINT, in BASIC, means "type to the screen", not to the printer!

1.2 Hello, .BAS file!

But really, it's OK doing this when your program is a single line. Once it gets longer, like,

say, a thousand lines, you really don't want to type the whole thing in every time you want to run a program. This is why we store our programs in text files. Open your favourite text editor (StyledEdit will do just fine) and type in the exact same command as above. Call the file *helloworld.bas* when you save it.

BASIC programs traditionally use the extension .BAS. This is Haiku - extensions are not really needed. Still, tradition counts for something too. If you also use another BASIC on your system, you may consider using the extension .YAB for your yab programs. If so, it will be up to you to make the necessary changes to all the instructions that follow.

Open a Terminal in the same folder where you saved helloworld.bas and type the following command:

```
yab helloworld.bas
```

This line calls the yab interpreter and tells it to load and execute the program helloworld.bas. If your program was in a different folder you could have used

```
yab /boot/home/SomeOtherFolder/helloworld.bas
```

But if the full path to that file had spaces in it, you would have had to enclose it inside quotation marks:

```
yab "/boot/home/Some Other Folder/helloworld.bas"
```

or you could have *escaped* it

```
yab /boot/home/Some\ Other\ Folder/helloworld.bas
```

Spaces in file names are the bane of a yab programmer's life. If your program mysteriously does not run, this is one of the first things to check for. The next thing to look for is a missing quotation mark. Yab uses these *a lot*, and if one is out of place, you will get an error message. The error system in yab is not very smart and you may need to look several lines back from where yab think the fault is (rarely forward, by the way).

1.3 Hello, Alert

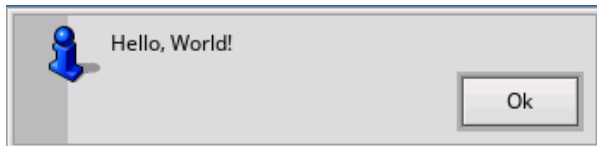
But enough mucking around with a terminal interface. In haiku, you can get "Hello, World!" onscreen with a single bash command:

```
~> alert "Hello, World!"
```

Unsurprisingly, you can do the same thing in yab. Take the text file called helloworld.bas and edit it in StyledEdit until it reads as follows

```
alert "Hello, World!", "Ok", "info"
```

When you run helloworld.bas now you should get this on your screen:



If not, check that you entered the line exactly the way it is in the line above.

The ALERT command (also called a keyword) in yab takes *parameters* that define what, exactly, needs to be done. Here the first parameter is the text to be displayed. The second parameter is the text on the single button, and the final parameter is the icon to be displayed. Parameters are separated by commas and surrounded by double quotation marks *unless* they are numbers.

In yab, 1 and "1" are not the same thing. The first is the *number* 1, which in some parameters will be an actual number followed by 2 and 3 and so on, but which can also be a different way of writing that something is TRUE. In the latter case 0 means FALSE.

"1", on the other hand, is a *string*, a sequence of signs like letters used to indicate something meaningful that is *not* a number. Strings can be written directly, within double quotes: "abcdefg" or they can be given a name, which must end with a dollar sign: *mystring\$*. numbers are written without quotes: 12345.678, and are named without the dollar sign: *MyNumber*. Those are the only data types you need to know about in yab. Your poor friends learning C++ need to keep a dozen or so data types straight...

Giving a string or number a name is called *assigning a value to a variable*. This is quite easy:

```
my_variable = 23
my_variable$ = "twenty=three"
```

If it makes it more understandable for you, you can use the old LET keyword

```
let my_variable = 23
let my_variable$ = "twenty=three"
```

Exercise 1

Question: What would happen if we changed the code in helloworld.bas to:

```
alert "Ok", "Hello, World!", "info"
```

Well, try it and see! The point of this exercise is that parameters to yab commands must be supplied in strict order. You are not free to supply them in the order that makes sense to you as you can sometimes do in bash.

You can write *alert*, *Alert* or *ALERT* as you please. Commands are not case-sensitive in yab. But I strongly suggest that you develop a style of writing these things and stick to it. In yab, many things are not case-sensitive, but some are. Unfortunately you just have to remember which are and which are not, but you get used to it after a while. The best approach is to pretend that everything *is* case-sensitive and write your program that way.

And about those commas that separate the parameters: if you like (and I do) you can put a space after the comma to make your code a little easier to read.

1.4 Hello, yab!

So, we can make a little alert. That's a good first step, but it is a little ... meh. Let's do something a little more ambitious: a really prominent "Hello, world!" message that your user can't ignore. This example uses the DejaVu font package, so just install it quickly if you don't already have it: in Terminal, type

```
pkgman install dejavu
```

and press Enter. All done? Excellent. Now create the following file as helloworld2.bas. Your e-book reader might wrap some of these lines, so read them carefully. Turn word wrap off in your text editor.

```
#!/yab
#This program is a better Hello World
//we can make comments
REM in different ways
//open a window
window open 100,100 to 400,150, "HelloWorldWindow", "Example
Program"
//make a view on the window
//the window itself is a kind of view, but DRAW SET does not work
well on it
view 0,0 to 300,50, "HelloWorldView", "HelloWorldWindow"
// set the font - need the dejavu package for this example.
draw set "DejaVu Sans, Bold, 40", "HelloWorldView"
//draw the text
draw text 5, 40, "Hello, World!", "HelloWorldView"
//start an infinite loop that can only be broken if the
//user closes the program
while(instr(message$, "Quit") =0)
    print "checking for message from Haiku ..." //this shows the
processes
wend
print "exiting program" //delete this when you understand it
exit
```

Open a Terminal and enter the following command

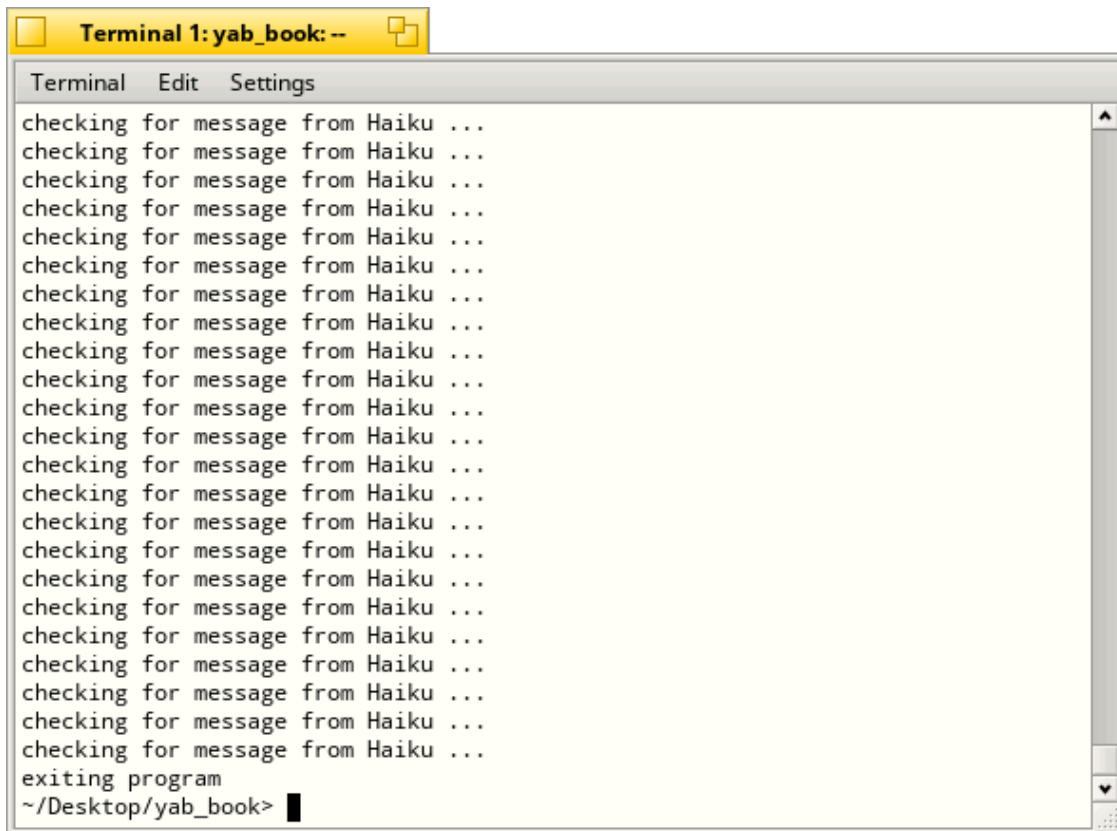
```
yab helloworld2.bas
```

Assuming that you retyped the code perfectly, you should see the following appear on screen:



If not, take a look at the error message and adjust the code until it does work. In this case that means copying it more accurately, but if you were writing code on your own it would mean hunting for that elusive missing quotation mark or comma.

Close the program from the window tab and take a look at the Terminal from which you started the program. It should look something like this:



Now that's more like it! Let's go through it line by line and see what is going on:

```
#!yab
```

This first line is not really used at the moment, but we will start using it later on..

Basically it tells Haiku what kind of script/program this is. This only works on the first line of a script and is called a "shebang".

If you like, you can save a few microseconds by specifying exactly where yab can be found:

```
#!/bin/yab
```

and

```
#!/boot/system/bin/yab
```

are the same thing on a Haiku system. Some programmers will use the *env* utility to find the executable:

```
#!/bin/env yab
```

All of those will work, but I am happy to let the system scour the PATH and find the executable for me. Note that there is no space after the exclamation mark.

A yab program always start on line 1 and if left to itself, will go down the lines one by one towards the last line unless *you* tell it "wait, go to this other lot of lines first, and don't come back here until you are finished, then go to the next line". This sort of command is called a *loop* and we will be doing a lot of those later on.

Normally the end of a line is signified by a linefeed code, which you create by pressing the ENTER key on your keyboard. But you can put more than one command on a line by using a colon:

```
print "Hello,": print "World!"
```

is the same as

```
print "Hello,"  
print "World!"
```

This can be useful in creating tiny little programs called *one-liners*, but try to avoid the colon format for now.

Right, let's make like our program and move on to line number 2.

```
#This program is a better Hello World  
//we can make comments  
REM in different ways
```

These three lines show the three ways you can comment your code. Commenting is great when you come back to your code six months later, or when someone else tries to figure out your code.

Yab does not support multiline comments. Every new comment line must start with one of these three comment markers. Take your pick of them and stick to it, but *//* is a little more versatile than the other two, because it can come at the end of a line as well as at the beginning.

```
//this is a comment
Print "not a comment" //but this is a comment.
```

So I strongly suggest that you use this format. OK, back to our program. To save space I will not reproduce the comments in the following discussion.

```
window open 100,100 to 400,150, "HelloWorldWindow", "Example
Program"
```

Well, yes, you need a window to do anything else, so let's open one. The format is

```
window open x, y, to x + width, y + height, "ID", "Title"
```

You can think of x and y as "how far from the left and top of the screen" (in pixels). Here we have asked for a window that starts 100 pixels from the left and 100 from the top and goes on to 400 pixels from the left and 150 from the top. Our window is therefore 300 x 50 in dimension. Remember that: we will need it in the next step. Keep in mind that the window tab will take up around 20 pixels all by itself, but Haiku will compensate for that. You only need to worry about it if your program takes over the whole screen.

By the way, a yab program can control more than one window. My program *FontMonkey* has one window with a list of fonts, and another that displays the currently selected one.

Next, we need to create a view on our window. A view is a kind of invisible area on which we can draw things and erase them if necessary.

```
view 0,0 to 300,50, "HelloWorldView", "HelloWorldWindow"
```

In the early days of yab, this was a compulsory step. Today the window itself is also a view, and I have written a lot of programs that just draw straight onto the window. But there are a few commands that don't work well on the window view, and changing font size and appearance is one of them.

The syntax is straightforward: make a view of 300 pixels wide and 50 pixels high. Call it HelloWorldView and put it on top of the window called HelloWorldWindow, starting at the very top left. A real nerd would have made it "view 0,0 to 299,49", but yab is very forgiving of these things - it actually lets you create a view larger than the window if you want. In this particular case, I covered the entire window with a single view, but you may want to have multiple views on your window in your own programs. That way, you can wipe out a view, and whatever is on top of that view, without disturbing anything else.

OK, let's pick up a font

```
draw set "DejaVu Sans, Bold, 40", "HelloWorldView"
```

The font selection command DRAW SET is a little tricky and sometimes you need to tweak it until you get it right. Let's actually put some text on the view now.

```
draw text 5, 40, "Hello, World!", "HelloWorldView"
```

Now how did I know to put the text 5 pixels from the left and 40 from the top? I didn't! I tried something and ran the program. No, left a bit, and run it again, Down a few pixels,

run it again. Repeat until you have something aesthetically pleasing. It gets quicker with practice.

But if we leave it there, our window, view and text would flash onto the screen and exit too fast to notice. We need to do something to keep it there. Ladies and gentlemen, welcome to our first loop:

```
while(instr(message$, "Quit") =0)
    print "checking for message from Haiku ..."
wend
print "exiting program"
exit
```

The WHILE ... WEND loop is one of several kinds of loop available in yab. In later chapters we will talk about different kinds of loops, but here is how this one works.

```
    WHILE something is the case
        do this
        and this
    WEND means go back to the line starting with WHILE
    continue from here
```

So this loop will continue running forever until the condition we specified under WHILE is no longer TRUE. And what is that condition?

```
    instr(message$, "Quit") =0
```

INSTR() is a built-in function that looks at one string of characters and sees if another string is contained within it. So the string "at" is contained within "Pirates" and INSTR() would give us a result of 4 because that is where the "at" starts within "Pirates". More interestingly, if INSTR() cannot find the shorter string within the longer one, it gives a result of 0.

The string we will examine is called *message\$*. This is a system variable in yab and a name you must never, ever use for your own strings. Every couple of microseconds, yab asks Haiku, "any messages for me?" If there is, yab puts it into the string called *message\$*, until the next one.

In this case I happen to know that when the user clicks on the Close button in the window tab, Haiku sends a message to the yab program that includes the string "Quit". Not "quit", or "QUIT", just "Quit". so I use INSTR() to look for "Quit" inside *message\$*. If it is not there, INSTR() gives a result of 0 and we merrily go around one more time on the WHILE ... WEND loop.

But if the result is anything other than zero (I don't need to know what it is, just that it is not zero) that means that a string containing "Quit" was sent to the program by the operating system, which means that the user must have clicked on the Close button. Therefore exit the loop and continue.

The two statements that start with PRINT are just there to show this process happening. Once you understand it, you can delete them. This use of PRINT statements is one of the

main ways I debug my programs, by the way. But a PRINT statement inside a loop really slows things down, and we don't want that.

Finally, we use the EXIT command to close down the program. Don't just let your program peter out for want of instructions. Clean things up nicely by giving the EXIT command. It doesn't have to be at the actual end of your code. In fact, once we start using subroutines it rarely is. Also, it is good programming practice to provide only one point at which your program will actually quit.

Before we move on, please note that WHILE...WEND has an evil twin called REPEAT...UNTIL. They do pretty much the same thing, cycling through the loop until a condition is met. The difference between the two is that while WHILE...WEND states its conditions at the top of the loop, REPEAT...UNTIL does it at the end. Pick one and stick with it. The following two little programs are exact equivalents:

```
while(a<10)
    a=a+1
wend
print a
```

```
repeat
    a=a+1
until(a=10)
print a
```

Exercise 2

Change the font size from 40 to, say, 64. Now change the size of the window and the view and move the text around until everything looks good again.

1.5 Hello, Bling-bling!

OK, we're getting there, and this is already the longest "Hello World" chapter in the history of computing, but the job's not over yet.

Anyone can grab our window by the bottom-right corner and resize it. The window can be minimised and zoomed. All of that just undoes our hard work, so let's put a stop to it. While we are at it, let's make that window tab look slightly different.

We are a little short of space here, so I am not going to reproduce the shebang line and comments from here on out, OK?

Make a copy of helloworld2.bas and call it helloworld3.bas. I know, I know, total lack of imagination. Edit the new file until it reads like this:

```
window open 100,100 to 400,150, "HelloWorldWindow", "Example
```

```

Program"
window set "HelloWorldWindow","look", "floating"
window set "HelloWorldWindow","flags", "Not-Zoomable"
window set "HelloWorldWindow","flags", "Not-Minimizable"
window set "HelloWorldWindow","flags", "Not-Resizable"
window set "HelloWorldWindow","flags", "Accept-First-Click"
view 0,0 to 300,50, "HelloWorldView", "HelloWorldWindow"
draw set "bgcolor", 255,255,255, "HelloWorldView"
draw set "highcolor", 155,200,155, "HelloWorldView"
draw set "DejaVu Sans, Bold, 40", "HelloWorldView"
  draw text 5, 40, "Hello, World!", "HelloWorldView"
while(instr(message$, "Quit") =0)
wend
exit

```

The first thing to note is that the WHILE ... WEND_ loop is now empty, and that is fine. Yab will tolerate a loop that exists only to keep a window onscreen. Let's have a look at the new elements.

```

window set "HelloWorldWindow","look", "floating"

```

This sets the window tab to look a little more modest

```

window set "HelloWorldWindow","flags", "not-zoomable"
window set "HelloWorldWindow","flags", "not-minimizable"
window set "HelloWorldWindow","flags", "not-resizable"
window set "HelloWorldWindow","flags", "accept-first-click"

```

These commands can be combined into one line, but I find it easier to maintain my code if I keep them on separate lines. They tell the window to change its behaviour, but also its looks. If a window is now non-zoomable, then the zoom button does not even show up.

```

draw set "bgcolor", 255,255,255, "HelloWorldView"
draw set "highcolor", 155,200,155, "HelloWorldView"

```

And finally, we set the background colour to white and the letters themselves to a shade of teal (Well, I think it is teal, but I am a little colour-blind). Our entry in the Competition to Make the World's Fanciest Hello World Program looks like this:



1.6 Running the app

Open a Terminal and cd to the directory where you stored helloworld3.bas. Make sure that the shebang line is in place on line 1 of that file. Type the following bash command:

```
chmod +x helloworld3.bas
```

You can now run `helloworld3.bas` from the Terminal like any other program, just by typing its name. Or try double-clicking it in Tracker.

But what's this? Our app does not get its own entry into the Deskbar, instead it is on a submenu of a sad-looking entry for `yab`. And it won't show up in Recent Applications either. For that we have to make a "standalone" version.

There are two ways to do this, "binding" and the BuildFactory. In this chapter we will restrict ourselves to binding the program. This means that we take our script and glue it onto a copy of the `yab` interpreter. That sounds complicated, but `yab` makes it easy enough. In Terminal, type:

```
yab -bind helloworld3 helloworld3.bas
```

You should see the following message:

```
---Info in helloworld3.bas, line 15: Successfully bound  
'/bin/yab' and 'helloworld3.bas' into 'helloworld3'
```

Now you have to *chmod* `helloworld3`, just like you did before.

```
chmod +x helloworld3
```

Go ahead and double-click on `helloworld3` now. Cool!

But we are not quite finished. *helloworld* now appears in the Deskbar's list of currently running applications but it has an ugly generic icon and it won't show up in Recent Applications. Let's fix that.

Rightclick on `helloworld3`, select Add-ons, then FileType. You should see this:

helloworld3 application type

File

Signature:

☐ Application flags

☐ Single launch ☐ Args only

☒ Multiple launch ☐ Background app

☐ Exclusive launch

Icon

Supported types

Add...

Remove

Version info

Development

Long description:

Drag in an icon of your own choice and fill in the rest until it looks like this:

helloworld3 application type

File


Signature:

☒ Application flags

☒ Single launch ☐ Args only

☐ Multiple launch ☐ Background app

☐ Exclusive launch

Icon 

Supported types

Add... Remove

Version info

1 0 0 Development 0

Long description: Hello World 3

Close the FileType window and click Save when prompted. Now when you run the program, your icon shows up and your app will appear on the Recent Applications list.

There is a more programmatic way to do all this: you will find it in Appendix 2.

The other way to make a standalone version is to use the BuildFactory in the official yab IDE. We'll get to that in chapter 3, and in chapter 4 we'll discuss the difference between the two methods.

One final note before we move on: "standalone" versions of yab apps are not as standalone as they used to be. There was a time when even a one-line yab program, once bound, would balloon to nearly a megabyte. But since then much of yab has been moved into the library *libyab.so*, which is contained in the yab HPKG. Standalone apps are now a more reasonable minimum size of about 220 KB, but the downside is that yab must be

installed for the app to run.

While the creation of HPKGs is beyond the scope of this e-book, we should therefore just note that your HPKG's *.PackageInfo* file should contain a section like one of these:

```
requires {  
    lib:libyab.so  
}
```

```
requires {  
    yab  
}
```

We've come a long way in just one chapter. In the next chapter, we are going to do something more ambitious. The next program we write will do something actually useful, and it will access the Internet to do it.

Exercise 3

There are at least two ways to ensure that the user has the DejaVu font package installed. One is a feature of Haiku, the other can be done within your own yab program, but requires a command we have not covered yet. Refer to the Haiku and yab documentation to figure out how to do it. The answers are at the end of Chapter 2.

1.7 Bonus section: Hello, Dolly!

This section is *optional* and just for fun, really. It is more advanced material, so feel free to come back to it after you have finished the next three chapters. It demonstrates how to build a Hello World message so utterly obnoxious that your user will desperately want to close it down. But the program wants to live and slips away from the mouse ...

```
#!/yab  
//helloworld4.bas - the "Hello, World" from hell  
xorigin = 100: yorigin = 100  
window open xorigin, yorigin to 400,150, "HelloWorldWindow",  
"Example Program"  
window set "HelloWorldWindow","look", "floating"  
window set "HelloWorldWindow","flags", "Not-Zoomable"  
window set "HelloWorldWindow","flags", "Not-Minimizable"  
window set "HelloWorldWindow","flags", "Not-Resizable"  
window set "HelloWorldWindow","flags", "Accept-First-Click"  
view 0,0 to 300,50, "HelloWorldView", "HelloWorldWindow"  
draw set "bgcolor", 0,0,0, "HelloWorldView"  
draw set "highcolor", 255,255,255, "HelloWorldView"  
draw set "DejaVu Sans, Bold, 40", "HelloWorldView"
```

```

    draw text 5, 40, "Hello, World!", "HelloWorldView"
while(instr(message$, "Quit") =0)
    xorigin = setxy("x"):yorigin = setxy("y")
    window set "HelloWorldWindow","moveto", xorigin, yorigin
    draw set "bgcolor",
change color(),change color(),change color(), "HelloWorldView"
    sleep 0.2
wend
exit

sub setxy(whichdim$)
    local xbound, ybound
    xbound = peek("desktopwidth")
    ybound = peek("desktopheight")
    switch whichdim$
        case "x"
            xorigin = xorigin + int(ran(40))
            if xorigin > xbound - 300 xorigin = 0
            return xorigin
            break
        case "y"
            yorigin = yorigin + int(ran(20))
            if yorigin > ybound - 50 yorigin = 0
            return yorigin
            break
    end switch
end sub

sub change color()
    local newcolor
    newcolor = int(ran(256))
    return newcolor
end sub

```

Chapter 2: Shortening a URL

Have you visited http://www.This-is_an_Extremely-Long-And-Unwieldy-URL.com/wher-

ever/135-792-4680.html lately? Didn't think so. Luckily there is a web service called *tinyurl.com* that will take that long web address and give you a much shorter one that will get your users to that same page. If your user types in that short URL, they will magically be redirected to the long one. A great service, I use it all the time.

We don't need to know how tinyurl works. What is interesting to us is that it has a little API that allows people to access it from the command-line. It requires the curl command, which is standard in Haiku, and from bash it looks like this

```
curl http://tinyurl.com/api-create.php?
url=http://www.Very_Long_URL.com/blablabla.html
```

OK, that is a bit unwieldy. We are going to write a few variations of a utility that will allow us to shorten that command. Yep, that's ironic.

From this point on, the applications are going to get a little long, so let's just number the lines for easy reference. But even though I number the lines, you should NOT do that when you retype the code.

2.1 Non-interactive URL shortening

The first variation will be a command-line app. let's call it ShortURL1.bas. The way this will work is that we want to type the following into Terminal

```
ShortURL1.bas www.Very_Long_URL.com/blablabla.html
```

And the program must then spit out the tiny URL back to the Terminal. That's it.

Or not. What will the program do if the user neglects to give that URL at the end, or if the user types one of these?

```
ShortURL1.bas -h
ShortURL1.bas --help
```

We need a fallback procedure that will display some basic information about how to use the program.

This is not a very profound program and let's face it, in real life I'd probably write it in bash rather than yab. But creating one in yab anyway gives us an excuse to learn some new yab tricks.

2.1.1 The program

```
1  #!yab
2  //ShortURL1.bas
3  //Feed a long URL to tinyurl.com and get a short URL back
4  longurl$ = peek$( "argument" )
```

```

5  sendthis$= "curl http://tinyurl.com/api-create.php?url=" +
  "'" + longurl$ + "'"
6  switch longurl$
7      case ""
8      case "-h"
9      case "--help"
10         ShowHelp()
11         break
12     default
13         print system$(sendthis$)
14 end switch
15 exit
16
17 sub ShowHelp()
18     print "ShortURL1.bas"
19     print
20     print "Obtain a short URL from tinyurl.com"
21     print "Usage:"
22     print "ShortURL1.bas <LONG_URL>"
23     print "ShortURL1.bas -h or --help"
24     print
25 end sub

```

Type that up (**without** the line numbers), *chmod +x* the file and find a nice long URL in your browser. Then run it from Terminal with the command;

```
ShortURL1.bas That_nice_long_URL_comes_here
```

Does it run? It should look something like this:

```
Terminal 3: yab_book: --
Terminal Edit Settings
Shortcuts      ShowDeskbarPosition  ShowDesktopWidth
ShortURL1.bas  ShowDeskbarWidth      ShowImage
ShowDateInYabFormat ShowDeskbarX          ShowPeople
ShowDeskbarExpanded ShowDeskbarY          ShowScrollbarWidth
ShowDeskbarHeight ShowDesktopHeight     ShowTabHeight
~/Desktop/yab_book> Short
Shortcuts      ShortURL1.bas
~/Desktop/yab_book> ShortURL1.bas https://discuss.haiku-os.org/t/converting-applications-to-the-layout-api/4438
---Error in ./ShortURL1.bas, line 5: syntax error at "\"0x0a"
---Error: Program not executed
~/Desktop/yab_book> ShortURL1.bas https://discuss.haiku-os.org/t/converting-applications-to-the-layout-api/4438
sh: -c: line 0: unexpected EOF while looking for matching `''
sh: -c: line 1: syntax error: unexpected end of file

~/Desktop/yab_book> ShortURL1.bas https://discuss.haiku-os.org/t/converting-applications-to-the-layout-api/4438
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
   0      0     0     0     0      0     0      0  ---:--:  ---:--:  ---:--:
100    26  100    26     0      0    35      0  ---:--:  ---:--:  ---:--:  4
1
http://tinyurl.com/j4eut52
~/Desktop/yab_book>
```

If you read the text in that picture closely you will see that I made two errors before I got it to run! The first one was a missing ". The second was that I forgot to close off the command I sent to curl with a second '. So don't despair if your program does not run immediately. Just keep plugging away and see what went wrong. It happens to all of us. And no, I didn't put those errors in deliberately. That will come later.

Now try forgetting about the long URL and just issuing the command *ShortURL1.bas* by itself. You get the help text for the program. But if you try to run

```
ShortURL1.bas --help
```

you get some generic help for yab itself, not for your program! What went wrong?

Nothing really, it's a bug in yab. If you *bind* your program like we discussed in the previous chapter, your help text will appear as it should. Try it now:

```
yab -bind shorturl ShortURL1.bas
```

```
chmod +x shorturl
```

```
shorturl --help
```

Notice something? The help text still says to run *ShortURL1.bas*, not just *shorturl*. Oops! Keep it in mind for the next exercise.

2.1.2 What's new?

OK, let's talk about what is new here.

```
4  longurl$ = peek$( "argument" )
```

PEEK\$("argument") is a built-in yab function that looks if there was anything on the command line after the name of the program itself. We feed the results of this into the string variable *longurl\$*.

```
5  sendthis$= "curl http://tinyurl.com/api-create.php?url=" +  
  "'" + longurl$ + "'"
```

We set up another string variable called *sendthis\$* into which we put the command we will later send to the system. This is called string concatenation. You can make up a long string by gluing short strings together with +. You'll be doing that a lot.

What is that + "'" business? Well, URL's should not contain spaces. But some do, so we take care of that by putting the URL between a set of single quotation marks. Instead of

```
curl http://tinyurl.com/api-create.php?  
url=http://www.Very_Long_URL.com/bla bla bla.html
```

we are going to send

```
curl http://tinyurl.com/api-create.php?  
url='http://www.Very_Long_URL.com/bla bla bla.html'
```

It's a small change, but it can make the difference between getting a valid short URL and an invalid one. so "'" is a single quotation mark enclosed by double quotation marks to show that we are inserting a one-byte string containing just that single quotation mark here.

2.1.2.1 Switching!

```
6  switch longurl$
```

Now we come to a different kind of "loop". Actually, there is a different techy term for it, but "loop" is good enough for us. the SWITCH loop runs like this

```
    SWITCH a_variable  
        CASE whatever  
            do_something  
            break  
        CASE whatever_else  
            do_something_else  
            break  
    DEFAULT  
        do_this_if none_of_the_above_apply
```


END SWITCH

a *variable* can be a *numeric* variable or a *string* variable (which is what we have in this program), and CASE statements can be piled up (as we have done here) if they all have the same outcome. I don't need to make separate CASE...BREAK sections for "-h" and "--help" because regardless of which one it is, the desired action is the same.

So what we are saying here is *Take a look at the string variable longurl\$. If it is empty, if it is "-h" or if it is "--help", then do the command in line 10. In any other case, do the command in line 13.*

2.1.2.2 Iffy business

Is there any other way of doing this? Sure. We could have used the IF ...THEN loop.

```
if longurl$ = "" then
    ShowHelp()
elseif longurl$ = "-h" then
    ShowHelp()
elseif longurl$ = "--help" then
    ShowHelp()
else
    print system$(sendthis$)
endif
```

Or, in a more concise way:

```
if (longurl$ = "") or (longurl$ = "-h") or (longurl$ = "--help")
then
    ShowHelp()
else
    print system$(sendthis$)
endif
```

In your first few programs, don't try to be concise. Rather spell the flow of your program out as fully as possible. But a word about IF ... THEN. It is a very versatile kind of loop. If you have only a single instruction it can be written in a single line:

```
IF (something is true) (do something)
IF a$ = "Hello, World" print a$
```

Note that the THEN is not used in this case. However, if it is more comfortable for you, you can write this as

```
IF a$ = "Hello, World" THEN
    print a$
ENDIF
```

The full form of this loop gives you a lot of control over what is about to happen it is structured like this:

```

IF (something is true) THEN
    do something
ELSEIF (something else is true)
    do something else
ELSE //if none of the above are true
    do something completely different.
ENDIF

```

You can have as many ELSEIFs as you like, and the final ELSE is optional.

By the way, can you see how I use indenting to make it clear how the loops are structured? Indenting (also called *whitespace*) is optional in yab, but get into the habit of using it - it will make your programs far more readable. You can use tabs or spaces, as you prefer. Again, this is arguably not a real loop, since it won't loop forever, but in BASIC it has traditionally been discussed under that heading.

2.1.2.3 A matter of routine

```

10          ShowHelp( )

```

What is this *ShowHelp()* business? it is a subroutine. A subroutine is like creating your very own new yab command, which you can then use whenever you need it. In yab, subroutines come AFTER the main body of the program. Remember that: this is not bash or Pascal. Aside from that, you can arrange them in any order you like.

Yab recognizes two kinds of subroutine. The first one goes like this

```

GOSUB the_subroutine
....
....
LABEL the_subroutine
....
RETURN

```

This exists only to provide backwards compatibility with old programs. DO NOT use this in new programs! The new way of writing subroutines goes like this

```

the_subroutine( )
....
....
SUB the_subroutine( )
....
END SUB

```

Why is this a better way? Because it lets you pass both numeric and string variables around between your main program and your subroutine. The subroutine can then use these as *local* variables. Local variables can have the same name as those in the main program (bad idea, but possible), but they can be reassigned, killed and recreated within the subroutine without affecting their value in the main program. Each subroutine is also allowed to return ONE value back to the main program. It ends up looking like this:

```

a_variable = 1
a_variable = the_subroutine()
....
....
SUB the_subroutine()
    LOCAL b_variable b_variable = 2
    RETURN b_variable
END SUB

```

or you can do something like this:

```

a_variable = 1
a_variable = the_subroutine(a_variable)
....
....
SUB the_subroutine(b_variable)
    b_variable = 2
    RETURN b_variable
END SUB

```

In the second case we passed a global variable called *a_variable* to the subroutine, but we decided to call it *b_variable* internally to the subroutine, where it is automatically included as a local variable. When we RETURN that value, it becomes *a_variable* again!

We'll see this in action in a later chapter, but for now, let's just accept that the GOSUB method is an antiquated relic. Forget I ever mentioned it, OK?

Two notes here. Subroutine names are **case-sensitive**. *my_subroutine()* is NOT the same as *My_Subroutine()*. Also, if your subroutine is going to return a number, or return nothing, name it something like *my_subroutine()*. But if it is going to return a string, you must name it like this: *my_subroutine\$()*. Every subroutine can RETURN only *one* value. There are ways around that, but that would take us into the field of Advanced yab Programming. Maybe in my next e-book?

```

13          print system$(sendthis$)

```

The SYSTEM and SYSTEM\$ commands both do the same thing: *temporarily drop out of the yab program, activate the default shell, normally bash, and do something there*. If the SYSTEM command was used, the exit code is returned. If you used SYSTEM\$ what you get back is the standard output, i.e. a string. In either case, if you want that output, it is up to you to capture it. So this line could actually have been clearer if it were two lines:

```

a$ = system$(sendthis)
print a$

```

Oops, sorry, force of habit. I told you not to be concise and here I am doing just that.

System calls are essential if you are going to write GUI front-ends. But you shouldn't do it too often. Every system call means that you have the whole of your yab program and an entire bash environment loaded into memory and then you still have to load the program you were calling. It gets slow. Really slow. There is also an alternative to both of these:

the LAUNCH command. I'll leave it to you to investigate that one.

```
15  exit
```

This is the ONLY place where the program is allowed to exit. It's a rule I have broken myself in some of my programs, but try to stick to it. See the documentation for the subtle difference between EXIT and END. Then just keep on using EXIT.

```
17  sub ShowHelp()  
18      print "ShortURL1.bas"  
19      print  
20      print "Obtain a short URL from tinyurl.com"  
21      print "Usage: "  
22      print "ShortURL1.bas <LONG_URL>"  
23      print "ShortURL1.bas -h or --help"  
24      print  
25  end sub
```

And here is our subroutine. It is quite simple; all it does is print a bunch of stuff in Terminal. We didn't truly need a subroutine for this: all this could have been done inside the SWITCH loop. But this way is easier to read and adapt later on. PRINT without anything behind it will print an empty line. Some purists prefer to use PRINT "".

Exercise 4

Fix the program so that it always gives the correct information when used with the --help parameter, whether it is bound or not HINT: look at the Glossary of PEEK commands at the end of this e-book. Solutions at the end of this chapter.

2.2 Interactive URL shortening

Suppose we wanted to shorten more than one URL. Wouldn't it be great if we could just start the program and have it ask us for the long URL? Then, when it is finished, perhaps it could just put the short URL into the clipboard for us and ask for the next long URL?

So what we are saying is that this program would run on an endless loop until we told it otherwise. We'll need to specify some sort of code to let the user get out of the program.

```
1  #!yab  
2  //ShortURL2.bas  
3  //Feed a long URL to tinyurl.com (interactive)  
4  do  
5      print  
6      print "Please enter the long URL to be sent to  
tinyURL.com."  
7      print "You can copy from the clipboard if you like."  
8      print "Or enter EXIT to quit"  
9      line input longurl$
```

```

10      if lower$(longurl$) = "exit" break
11      sendthis$= "curl http://tinyurl.com/api-create.php?url="
+ "" + longurl$ + ""
12      result$ = system$(sendthis$)
13      print
14      print "Your short URL is " + result$
15      print "Do you want to copy this to the system clipboard
(Yes/No/Exit)"
16      line input longurl$: if lower$(longurl$) = "exit" break
17      if lower$(left$(longurl$, 1)) = "y" clipboard copy
result$
18 loop
19 exit

```

2.2.1 What's new?

A new kind of loop! DO...LOOP is designed to run forever. There is no condition built into it that will automatically terminate the loop, as we saw in WHILE...WEND. It is up to you to issue the EXIT or BREAK command that will either close the program completely or exit the loop and carry on.

LINE INPUT should be clear enough. It lets the user enter something, like a URL or the term EXIT. But what if the user types *exit* or *Exit* instead? We use the LOWER\$() function to convert the user's input to lowercase, then we see if what comes out of that is equal to *exit* if so, we BREAK out of the loop. That brings us to line 19 ... which is an EXIT command. You could save a line of code by making line 10

```

      if lower(longurl$) = "exit" exit

```

But this way is tidier. Then we ask if the user wants to copy this result to the clipboard. This time, we are not even going to look at the whole of the user's input. We use LEFT\$() to look just at the first letter after we converted it to lowercase. If that letter is y, we use CLIPBOARD COPY to put the result in Haiku's system clipboard. And then the loop goes back to the beginning and asks for the next URL.

You don't need to *type* those URLs, by the way. You can cut one from your browser's address bar and paste it into the Terminal where ShortURL.bas is running.

Exercise 5

This program will fail if the user is one of those people who always presses the space key after anything. We don't really want to send "EXIT " to tinyurl.com. Look up the documentation for the RTRIM\$() command and adapt line 10 to fix this problem.

Done? Now consider the user who disregards your instruction and types "QUIT" instead. How would you deal with that?

Much of programming, in any language, is not about the bit of code that does the actual work. We have used the same line sending a command to `tinyurl.com` several times now. Most of what you will be doing is anticipating things that could go wrong when the user does something unexpected.

2.3 Batch URL shortening

Let's make this a little more interesting. Suppose the user has a text file of URLs to shorten. Can we write a yab program to open that file, read the URL's and spit out a new file with all the shortened ones? You bet.

We are going to make things a little easy on ourselves. Instead of asking the user which file to open, let's just assume that it is in a file called *URLS.txt* and that this file is in the same folder as your program. The output file will be called *shorturls.txt*.

Before you do anything else, create *URLS.txt* and paste in a few URLs, one URL per line. You can throw in a few empty lines too - we'll be testing for that. If the input files has a blank line, or a blank line with a space, we are going to put a blank line or a space in the output file too, so the two files will have the same structure.

```
1  #!bas
2  //ShortURL3.bas - read a file, send URLs in it to tinyURL.com
3  //then gather the results in an output file
4
5  inputfile$ = "URLS.txt"
6  outputfile$ = "shorturls.txt"
7  totallines$ = system$("wc -l " + inputfile$)
8  totallines$ = left$(totallines$, instr(totallines$,
inputfile$)-1)
9  totallines = VAL(totallines$)
10 open inputfile$ for reading as #1
11 open outputfile$ for writing as #2
12 for f = 1 to totallines
13     line input #1 longurl$
14     if (longurl$ = "") or (longurl$ = " ") then
15         print #2 longurl$
16     else
17         sendthis$= "curl http://tinyurl.com/api-create.php?
url=" + "'" + longurl$ + "'"
18         result$ = system$(sendthis$)
19         print #2 result$
20     indef
21 next f
22 close #1
23 close #2
24 exit
```

Exercise 6

If you typed this program in perfectly, it will not run! I have introduced two deliberate errors. Run the program from Terminal and use yab's error messages to track them down and fix them. Answers at the end of this chapter.

```
Terminal 2: yab_book: --
Terminal Edit Settings
/Storage/yab_book> yab ShortURL3.bas
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           %             %         Dload  Upload  Total   Spent    Left   Speed
0          0      0      0      0      0      0      0  --:--:--  --:--:--  --:--:--
100        26      0      26      0      0     32      0  --:--:--  --:--:--  --:--:--  3
8
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           %             %         Dload  Upload  Total   Spent    Left   Speed
0          0      0      0      0      0      0      0  --:--:--  --:--:--  --:--:--
0          0      0      0      0      0      0      0  --:--:--  --:--:--  --:--:--
100        26    100      26      0      0     15      0  0:00:00  0:00:00  --:--:--  1
100        26    100      26      0      0     15      0  0:00:00  0:00:00  --:--:--  1
5
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           %             %         Dload  Upload  Total   Spent    Left   Speed
0          0      0      0      0      0      0      0  --:--:--  --:--:--  --:--:--
100        26      0      26      0      0     35      0  --:~:~:~  --:~:~:~  --:~:~:~  3
100        26      0      26      0      0     35      0  --:~:~:~  --:~:~:~  --:~:~:~  3
9
/Storage/yab_book>
/Storage/yab_book>
```

2.3.1 What's new?

By now, you should be able to look at a simple program like this and see the flow of logic within it. There will be new commands, but you can almost see what they do just from their names and their context in the program. If not, time to hit the documentation!

In real life I would probably have used a WHILE...WEND loop for this, but I've already explained that one, so I shoehorned this program into a FOR...NEXT loop instead. We'll discuss that below. For now, take note that we need to know the number of lines in URLs.txt. One way would be to open the file, read it line by line and keep count. Instead, we will shell out to the system and use the `wc -l` command to find the number of lines. That gives us a result like this

6 URLs.txt

We therefore now have to isolate that number at the beginning of the string. Remember, it could be 6, or 16, or 116, so just looking at the first character won't do.

My solution is to use INSTR() to find out where the filename is located. Subtract one

position to take care of the space and we can now use LEFT\$ to cut everything. Now we use VAL() to convert that string into a number.

So, we have an upper limit. Now we can set up a FOR...NEXT loop. We will discuss this structure in detail below, but for now, let's just say that it means "Do this a specified number of times". In our case, "start with one and repeat until you reach the number of lines in the input file".

Besides that, the other important new command here is OPEN. It lets you access a file on your computer. Of all the BASIC commands, OPEN is the one that varies the most wildly among the various BASIC dialects, and yab tries valiantly to be compatible with them all.

When you OPEN a file, you associate it with a *channel*. This channel can be any number from 1 to 124, on Haiku anyway. Other BASICs on other operating system typically restrict you to nine or so channels. You use your favourite OPEN syntax to open a file into a channel. You now use the same PRINT and LINE INPUT commands we saw above, but divert them to the channel. In that way we can read or write to the file. When you are done you CLOSE the channel immediately. Do not leave it open in case you are going to write to it later. CLOSE it. You can always OPEN it again later. Leaving channels open can lead to the most baffling errors you'll ever come across in yab. You have been warned! The # symbol can be optionally used to indicate a channel. I use it, again because of the way I learnt BASIC long ago.

For each line in the input file, I read it with LINE INPUT, specifying that I want to read it from #1 rather than waiting for the user to type something in. If the line is empty or contains a space, I PRINT it to #2, which is the output file. If it is a URL, then send it off to tinyurl.com, capture the results in the variable *result\$* and print that to channel 2. Move on to the next line, or exit the loop if we're done.

2.3.2 Next, please!

The FOR ... NEXT loop is one of the most fundamental structures in BASIC. You will hardly ever write a program without it.

You use FOR ... NEXT when you want to repeat a set of actions a known number of times. Not known to you, necessarily, but known to the program. The format is:

```
FOR f = 1 TO 10
do something
NEXT f
```

That "f" can be any letter from a to z, and the lower and upper values can be any whole numbers. This is perfectly acceptable:

```
for f = 27 to 35681246234987
```

Might take a while to run, though.

FOR...NEXT variables are not "used up". If I use f as my counter and close the loop with

NEXT f, I can start the next loop with f again, no problem. It's not as if you are restricted to 26 FOR...NEXT loops per program!

FOR ... NEXT loops can be embedded inside one another:

```
FOR f = 1 to 10
  do something
  FOR n = 1 to 20 STEP 2
    do something else
  NEXT n
NEXT f
```

The STEP command is included here just for demonstration purposes. *for f = 1 to 10* and *for f = 1 to 20 step 2* are in fact completely equivalent: both will cycle ten times. if you leave the STEP part off, you automatically get a step value of 1. And 99% of the time that is what you want.

The letter after NEXT is *optional*. Yab will keep track of what the next NEXT refers to. But specifying it helps to keep you straight on which loop to close off now. Sorry, you can only nest 26 levels deep, because that is how many letters there are in the English alphabet! If you ever write something nested 26 levels deep, and it works, you have my admiration. I've rarely needed to go beyond three or four.

By the way, I always use "f" for my first FOR...NEXT loop, then "n" for the second, embedded one. That has to do with the way the keyboard on my first computer, a Sinclair ZX Spectrum, worked. If you want to start at "a", then use "b", and work your way up the alphabet, that's great. Develop your own style to do these things and stick to that style. It will help you when you work on the program again a year later.

But as I've noted above, you may not necessarily know the value of the upper and lower bounds. Perhaps your program does something to establish the values of *variable1* and *variable2*. Now you can go ahead and say

```
for f = variable1 to variable2
```

That may fail, though, if *variable1* is bigger than *variable2*. Unlike some other dialects of BASIC, yab does not allow you to run a FOR ... NEXT loop backwards:

```
for f = 10 to 1
```

Nope, that will crash. There is a way to fake it:

```
for f = 10 to 1 step -1
```

but that is an advanced trick. 99% of the time you will want your FOR...NEXT loop to run normally, with a STEP value of 1. If you are going to compute your FOR...NEXT variables, test them for size first

```
IF variable2 > variable1 THEN
  FOR f = variable1 TO variable 2
    do something
  NEXT f
```

```

ELSE
    do something else
ENDIF

```

Exercise 7

Look at this one liner

```
for f = 1 to 3.9: print f : next f
```

What will be the last thing it prints? 3 or 4? Run it and see!

This is important if you are going to compute the values in a FOR...NEXT loop. The loop is not going to "round up" or "round down" the number you give it. It will take the integer part and ignore everything after the decimal point. The code to get around that problem looks like this:

```
variable = INT(variable + 0.5)
```

Use the yab documentation to figure out what I just did.

2.4 URL shortening in a GUI.

OK, let's get back to what makes yab special and program a real GUI application to shorten those URLs.

The code that follows is actually a little app of mine called TinyTim that is available on my repository. As before, the line numbers are just for convenience, and you should not retype them. In fact, there are some very long lines in this program, and the line numbers will help you to see what is happening if your e-book reader is wrapping those lines.

```

1  #!yab
2  //Fill in these fields with your own particulars. The
variables will be used in the About Box
3  ProgramName$ = "TinyTim"
4  AuthorName$ = "Michel Clasquin-Johnson <clasqm@gmail.com>"
5  ProgramVersion$ = "V0.1"
6  ProgramBriefDescription$ = "TinyTim allows you to send a long
URL to tinyurl.com and copy the resulting short URL to the
clipboard."
7  ProgramLicense$ = "Public Domain"
8
9  //*****Global Variables****
10 // set DEBUG = 1 to print out all messages on the console
11 DEBUG = 1
12 //change this to DEBUG = 0 when you are ready to bind the
program for distribution
13 LongURL$=""
14 ShortURL$="Shortened URL will appear here after conversion"

```

```

15
16  ## Commands to run before the Main Loop come here.
17  OpenWindow()
18
19  #####End of Prologue#####
20
21  //Main Message Loop
22  dim msg$(1)
23  while(not leavingLoop)
24      nCommands = token(message$, msg$(), "|")
25      for everyCommand = 1 to nCommands
26          if(DEBUG and msg$(everyCommand)<>"") print msg$
(everyCommand)
27          switch(msg$(everyCommand))
28              case "_QuitRequested":
29              case "MainWindow:_QuitRequested":
30                  leavingLoop = true
31                  break
32              case "aboutButton"
33                  alert ProgramName$ + " " + ProgramVersion$ +
"\n" + "by " + AuthorName$ + "\n\n" + ProgramBriefDescription$ +
"\n\n" + ProgramLicense$, "OK", "none"
34                  break
35              case "tinyurlButton":
36                  Launchurl$()
37                  break
38              case "copyButton":
39                  Copy2Clipboard()
40                  break
41              default:
42                  break
43          end switch
44      next everyCommand
45  wend
46  CloseWindow()
47  exit
48
49  sub OpenWindow()
50      //Setup the main window here
51      window open 100,100 to 500,280, "MainWindow",
ProgramName$
52      window set "MainWindow", "look", "floating"
53      window set "MainWindow", "feel", "modal-app"
54      window set "MainWindow", "flags","not-zoomable"
55      window set "MainWindow", "flags","not-resizable"
56      text 10,25 to 49,40, "URLlabel", "URL:", "MainWindow"
57      text 10,68 to 52,88, "Submitlabel", "Submit",
"MainWindow"
58      text 10,105 to 59,125, "Resultslabel", "Result:",
"MainWindow"
59      draw rect 70,10 to 390,50, "MainWindow"
60      Textedit 71,11 to 389,49, "URLField", 0,"MainWindow"

```

```

61      Button 70, 60 to 390, 90, "tinyurlButton", "Submit this
URL to tinyurl.com", "MainWindow"
62      View 70, 100 to 390, 130, "ShortURLView", "MainWindow"
63      Button 70, 140 to 280, 170, "copyButton", "Copy to
clipboard", "MainWindow"
64      option set "copyButton", "enabled", 0 //don't need this
until we have a short URL
65      Button 290, 140 to 390, 170, "aboutButton", "About",
"MainWindow"
66      DisplayShortURL()
67      ExamineClipboard()
68  end sub
69
70  sub CheckLongURL()
71      if LongURL$ = "" then
72          Alert "Please enter a long URL first", "OK",
"warning"
73          return 0
74      end if
75      If (left$(LongURL$, 4) <> "www.") and (left$(LongURL$, 5)
<> "http:") then
76          alert "Is this a valid URL? Please start your URL
with www or http. URLs are case sensitive and with no tabs,
spaces or trailing lines, please", "OK", "warning"
77          return 0
78      endif
79      if instr(LongURL$, "\n") <> 0 then
80          alert "There is an open line or more than one line in
your long URL (possibly at the end, did you press ENTER?) Please
remove it.", "OK", "warning"
81          return 0
82      endif
83      return 1
84  end sub
85
86  sub CloseWindow()
87      window close "MainWindow"
88  end sub
89
90  sub Copy2Clipboard()
91      if ShortURL$ = "" return
92      clipboard copy ShortURL$
93  end sub
94
95  sub DisplayShortURL()
96      draw flush "ShortURLView"
97      Draw text 0,20, ShortURL$, "ShortURLView"
98  end sub
99
100 sub ExamineClipboard()
101     //this should only happen at program launch
102     local clipcontent$

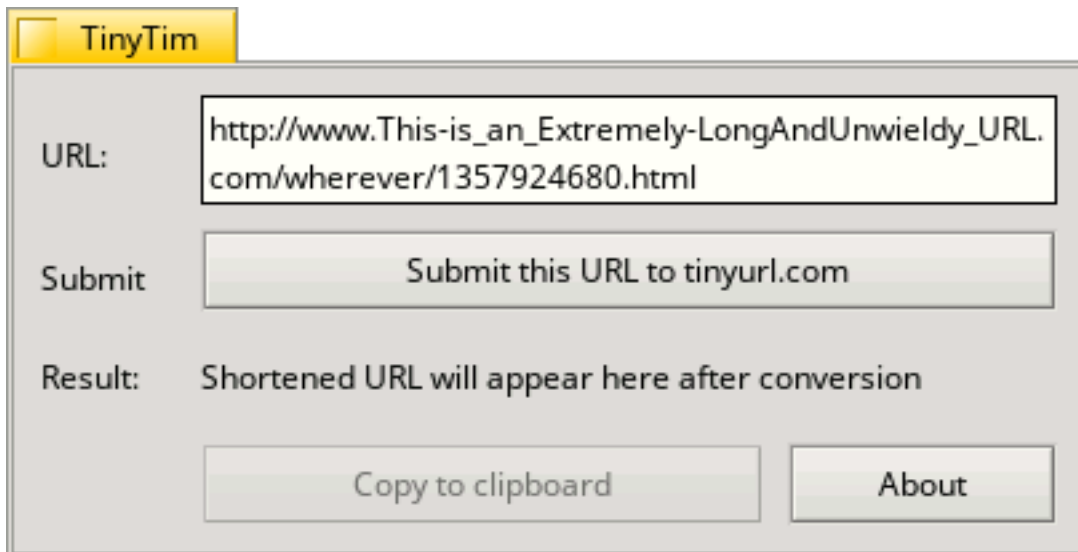
```

```

103     clipcontent$ = clipboard paste$
104     if (left$(clipcontent$, 4) = "www.") or (left$
(clipcontent$, 5) = "http:") then
105         textedit add "URLField", clipcontent$
106         LongURL$ = clipcontent$ // not strictly necessary,
but better to play safe
107     endif
108 end sub
109
110 sub Launchurl$()
111     local dothis$
112     LongURL$ = textedit get$ "URLField"
113     UIOnOff(0)
114     dothis$ = "curl 'http://tinyurl.com/api-create.php?url="
+ LongURL$ + "'"
115     ShortURL$ = system$(dothis$)
116     DisplayShortURL()
117     UIOnOff(1)
118 end sub
119
120 sub UIOnOff(state)
121     //state must be 1 or 0
122     option set "tinyurlButton", "enabled", state
123     option set "copyButton", "enabled", state
124     option set "aboutButton", "enabled", state
125 end sub

```

When we run this program and enter a URL, it should look like this:



2.4.1 What's new?

Quite a lot, but by now you should be able to look up any new command you come across. I am not going to comment on the details of the DRAW FLUSH command. Look

it up in the documentation. What is more important is that we have a look at an important structural component.

Lines 21 to 45 are the *main loop*. This is the engine that drives your application. It sits there, waiting for messages, mostly waiting for you to push a button. When a message arrives, it gets stored in the variable *message\$*. Yab does this for you automatically. We then cut it up into pieces. We happen to know that the different parts of *message\$* are separated by the | character, so we use the TOKEN command to separate the pieces that are not | and store them in an array called *msg\$()*.

Arrays are not used as much as they used to be in the BASIC world, but they do still have their uses. What an array does is collect a bunch of strings (or a bunch of numbers, never both together) and keep them together. So instead of having twelve variables called *a1*, *a2* and so on, we use the DIM command to create an array *a* with twelve slots and feed the twelve in there. Now we can refer to *a(1)*, *a\$(2)* and so on. It is a little more tidy.

Nice thing about arrays in yab: they grow as needed, and this is what happens here. We dimensioned an array with just one slot, but as we feed stuff into it, it creates new slots as we go. So don't worry about *msg\$()* running out of space. If you want to use arrays in your own programs, I suggest you read Appendix 4 first.

Now we take every variable contained in the array and compare them one by one to the names of the various buttons and controls. In fact, most of the messages yab sends internally are just one string long. When you press the About button, the main loop gets a message "aboutButton". How do we know? Because that's what we called it when we created the button! This is case-sensitive, by the way. But messages from elsewhere in the system take the form *BE_blablabla|moreblabla|Interesting_bit*, so we need to prepare for the possibility.

So we take the members of the array, one at a time, and run them through a SWITCH loop. If we get a hit, we send the program off to a subroutine to do something useful. Then it's back to waiting for the next message\$.

This main loop consists of a SWITCH loop inside a FOR ... NEXT loop inside a WHILE ... WEND loop. Unless the right command is issued, the program will keep on cycling inside the main loop forever, or until your laptop runs out of battery, whichever comes first.

This is not the *only* way to set up a main loop. You will recall that in the last chapter, for a very simple app, I just used the INSTR() function. Yes, those two lines actually were that program's main loop. But this is the canonical yab main loop. If you start a new project in the IDE (see the next chapter) this main loop will be set up for you automatically. All you need to do is to fill in the CASE statements.

No matter how many windows your program controls, you should have exactly ONE main loop. I tried to write a program with two main loops once and it was a nightmare to maintain.

Now look at lines 120 to 125. I don't want the user to push a button while curl is doing its

thing, so I need to grey out all the controls with the OPTION SET command. The easy way is to make up two subroutines, one to switch everything off and another to switch everything on again.

```
sub UIOff()  
option set "tinyurlButton", "enabled", 0  
option set "copyButton", "enabled", 0  
option set "aboutButton", "enabled", 0  
end sub  
  
sub UIOn()  
option set "tinyurlButton", "enabled", 1  
option set "copyButton", "enabled", 1  
option set "aboutButton", "enabled", 1  
end sub
```

Now I could just order the program to perform UIOff() or UIOn(), as required. Instead, I made a single subroutine that acts as a *toggle*. If I want things off, I request UIOnOff(0). When things are back to normal again, I issue the command UIOnOff(1). The subroutine picks up that 1 or 0 and saves it to a local variable called *state*. This local variable only survives as long as the subroutine does, and it does not interfere with any variables, local or global, with the same name anywhere else in the program. When it comes to the OPTION SET statements, I don't feed them a 1 or a 0, but the variable *state*. And so one subroutine does the work of two. Take another look:

```
sub UIOnOff(state)  
//state must be 1 or 0  
option set "tinyurlButton", "enabled", state  
option set "copyButton", "enabled", state  
option set "aboutButton", "enabled", state  
end sub
```

Now look at the ExamineClipboard() subroutine starting at line 100. This is used only once, when the program starts up, and we could just have placed it before the main loop. But once you get used to subroutines, it just makes sense to structure your program with them. What this does is to examine the state of the clipboard. If it starts with "www" or "http", it is probably a URL, and we pre-populate the textedit widget to save the user some time.

The *CloseWindow()* subroutine is not really, really necessary. If you use the EXIT command everything will close down anyway. But it appeals to the neat-freak in me to close the program down in an orderly way. Once you start writing programs with multiple windows, you will see why having a little subroutine like this for each one is actually the right way to go about it.. In this program I am only using the subroutine once, just before EXIT. But I could call it from anywhere.

2.5 Final thoughts on URL shortening

I hope that you have enjoyed the trip! From a single line of code that I found on the

Internet, we have generated three different CLI programs and one with a proper GUI. Along the way we have now learned all the loops and control structures that yab offers. There are many, many more commands available to you (see the Glossary), but by now you will have a good idea of how yab programs are constructed and how they function. Once you have that, you will be able to figure out how to write your own.

Don't try to memorize the exact syntax of every available command. If you know what they do, you can always look up the syntax in the documentation.

You can now program in yab. But before we proceed to the graduation ceremony, we need to do the advanced courses. Next up is the yab IDE.

Solutions to Exercise 3 (from the previous chapter)

The first possibility is that, if you are going to distribute your app as an HPKG, just change the *requires* section in your .PackageInfo file from

```
requires {
    yab
}
```

to

```
requires {
    yab
    dejavu
}
```

By now you should know the second way: somewhere early on in your program you put a SYSTEM command downloading dejavu.

```
system("echo y | pkgman install dejavu")
```

That will slow down your program a lot, though. The HPKG method is much to be preferred.

Solutions to Exercise 4

At first, it seems easy. Just change the ShowHelp subroutine:

```
17 sub ShowHelp()
18     print "ShortURL1.bas"
19     print
20     print "Obtain a short URL from tinyurl.com"
21     print "Usage:"
22     print "ShortURL1.bas <LONG_URL>"
23     print "ShortURL1.bas -h or --help"
24     print
25 end sub
```


Make the following changes:

```
22     print "shorturl <LONG_URL>"
23     print "shorturl -h or --help"
```

Now, if you have bound your program, you can run it without parameters, or with the -h or --help parameters, and it will give the correct information. But now it is wrong if we run it from a script! Is there a way to make it run correctly in both cases?

Yes, there is, but it will require a few extra steps:

```
17 sub ShowHelp()
18 if peek("isbound") then
19     print "shorturl"
20     print
21     print "Obtain a short URL from tinyurl.com"
22     print "Usage:"
23     print "shorturl <LONG_URL>"
24     print "shorturl -h or --help"
25     print
26 else
27     print "ShortURL1.bas"
28     print
29     print "Obtain a short URL from tinyurl.com"
30     print "Usage:"
31     print "ShortURL1.bas <LONG_URL>"
32     print "ShortURL1.bas -h or --help"
33     print
34 endif
35 end sub
```

PEEK and PEEK\$ are powerful commands to obtain information about your environment. See the *Glossary of PEEK commands* at the end of this e-book for a full list of these commands.

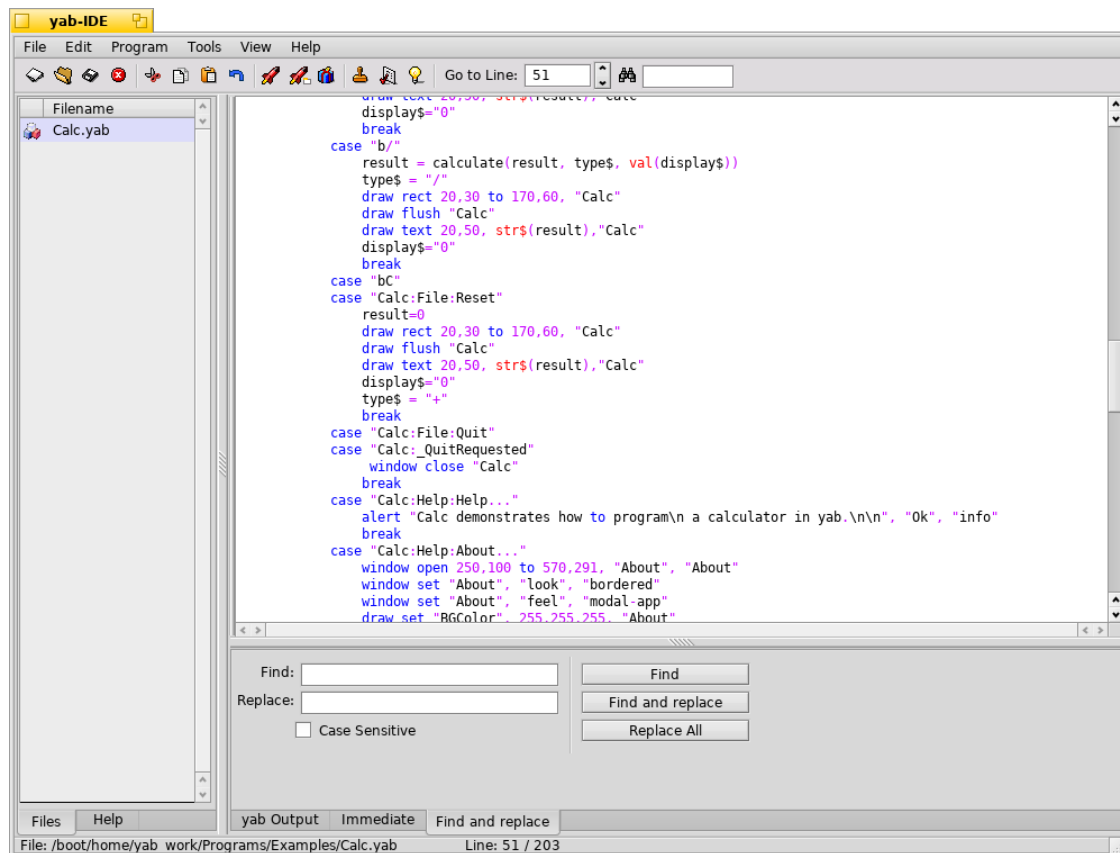
Solutions to Exercise 6

- Missing " at the end of line 6
- Change *indef* to *endif* in line 20

Chapter 3: The yab IDE

So far, we have been programming the old-fashioned way, with a text editor and a Terminal. This really is the best way to learn. It lets you get the feel of the language in your fingertips.

But you have that now. And so it is time to move on to the yab Integrated Development Environment (IDE). This is essentially a specialised text editor that does just one thing - write and run yab programs.



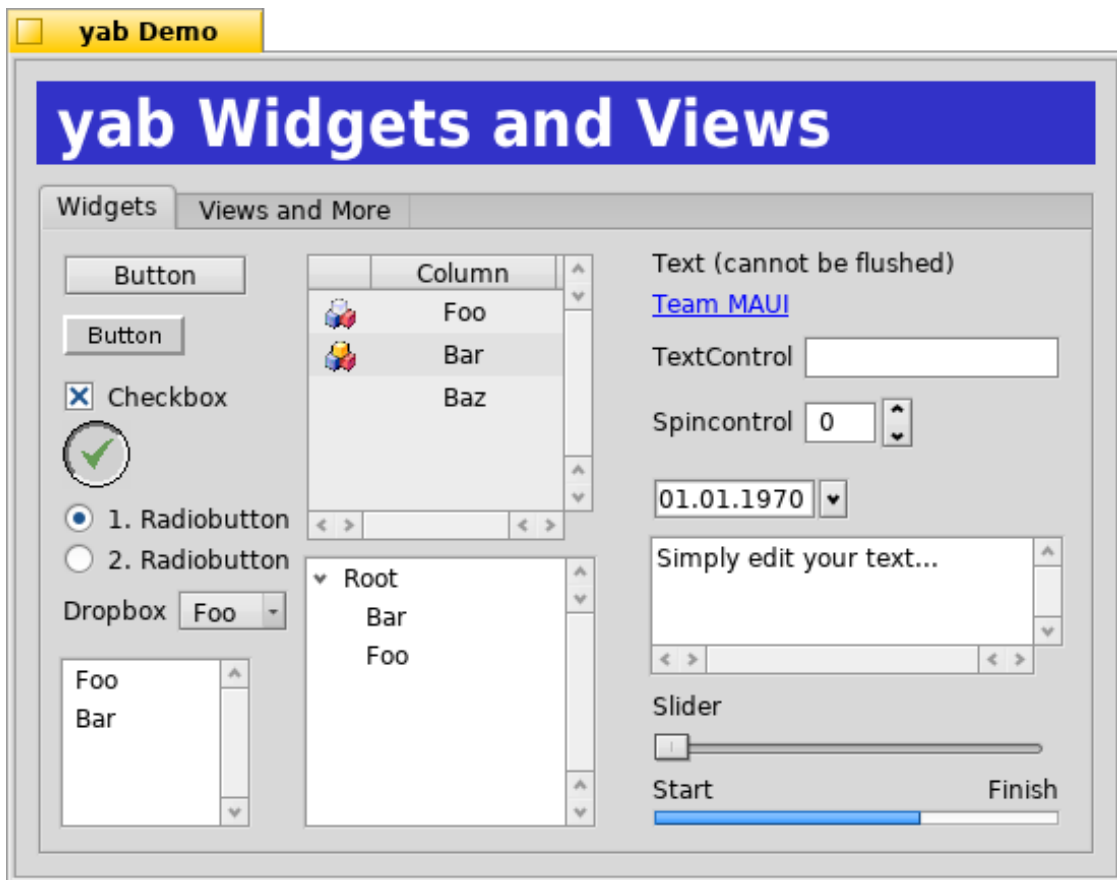
I am not going to bore you with a fine-grained description of every single command in the IDE. It's a computer program. If you have been around computers at all during the last thirty years, it will all be pretty self-explanatory. *File | Open* opens a new file, Search and Replace works the way it does in every other program and so on. Let's rather focus on the unique commands in the IDE, the ones directly related to yab.

On the left you will see a list of currently loaded files. Try not to have every yab file you possess loaded all the time - it slows things down a lot. At the bottom of that pane, however, you will see a tab called *Help*. Click on it and you will get explanations of

every single yab command. Pretty useful! When you're done researching, click on the Files tab again to get back to your work.

The *Program* menu lets you run your program in two different ways. You can either run it directly or open a Terminal and run your program from that. The *Run in Terminal* command is not only useful when you are writing command-line programs: it also comes in useful during debugging. But, as we shall see the regular *Run* command also has some tricks up its sleeve.

The way we are going to explore the IDE is to run and examine some of the sample programs that come bundled with it. Load up the following program:
`/boot/home/yab_work/Programs/Examples/AllInOne.yab`



This little demo shows all the widgets that yab gives you to play with. As you will see most of the code is about setting up the widgets, then there is a smallish main loop towards the end. Have you run the program? Great, now let's make one small change. Find these lines at the start of the Main loop:

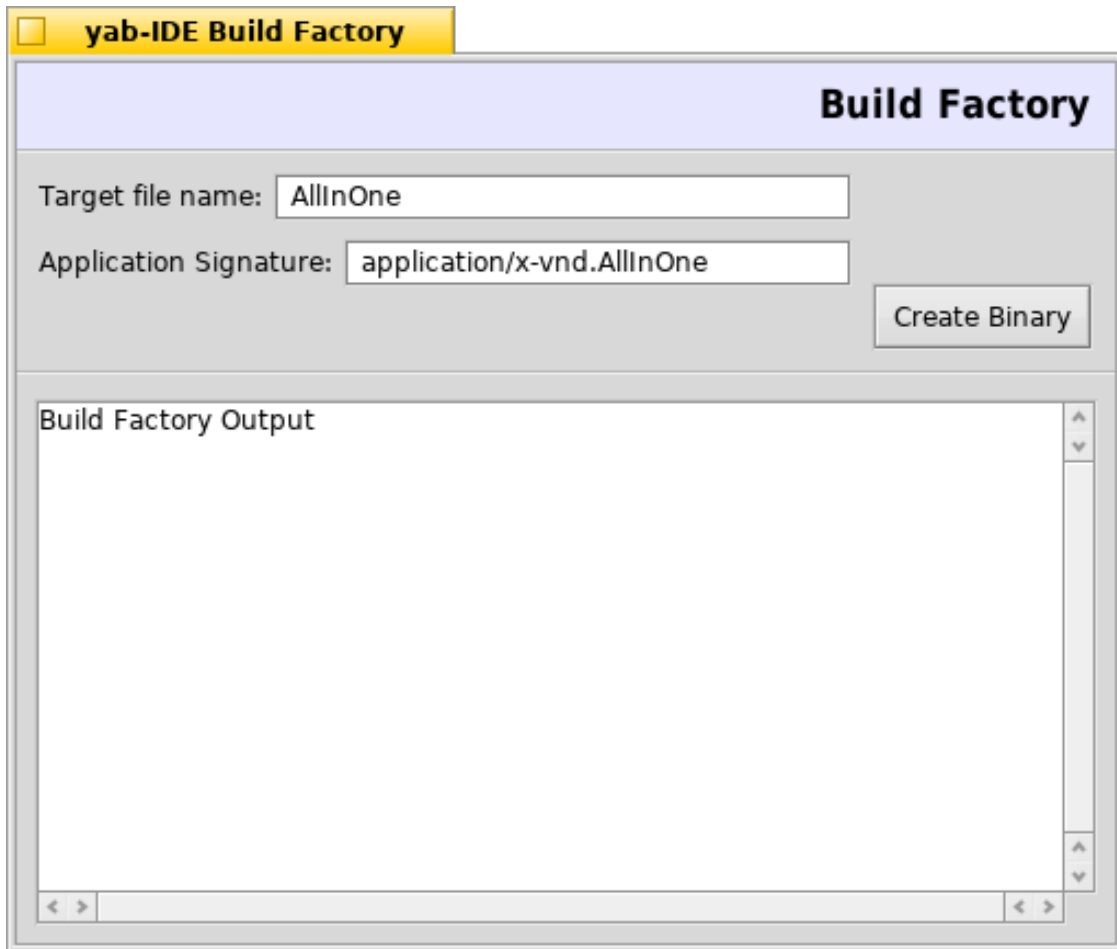
```
msg$ = message$  
  if(instr(msg$, "Popup")) then
```

Now insert a line in between so that it reads

```
msg$ = message$  
  if msg$ <> "" print msg$  
  if(instr(msg$, "Popup")) then
```

Now run the program again, click on a few things and see what happens in the *yab Output* pane at the bottom of the IDE. Suddenly, all those "messages" we have been talking about become as plain as daylight. Now do it again, but this time choose *Run in Terminal*. Where do you expect to see the messages appear this time?

The IDE does not offer an option to *bind* your program: instead it offers the much more powerful BuildFactory. We will discuss the difference between the two in the next chapter, but let's use the BuildFactory to make a standalone version of AllInOne.yab. Remove that line we just added. From the Program menu, select BuildFactory:



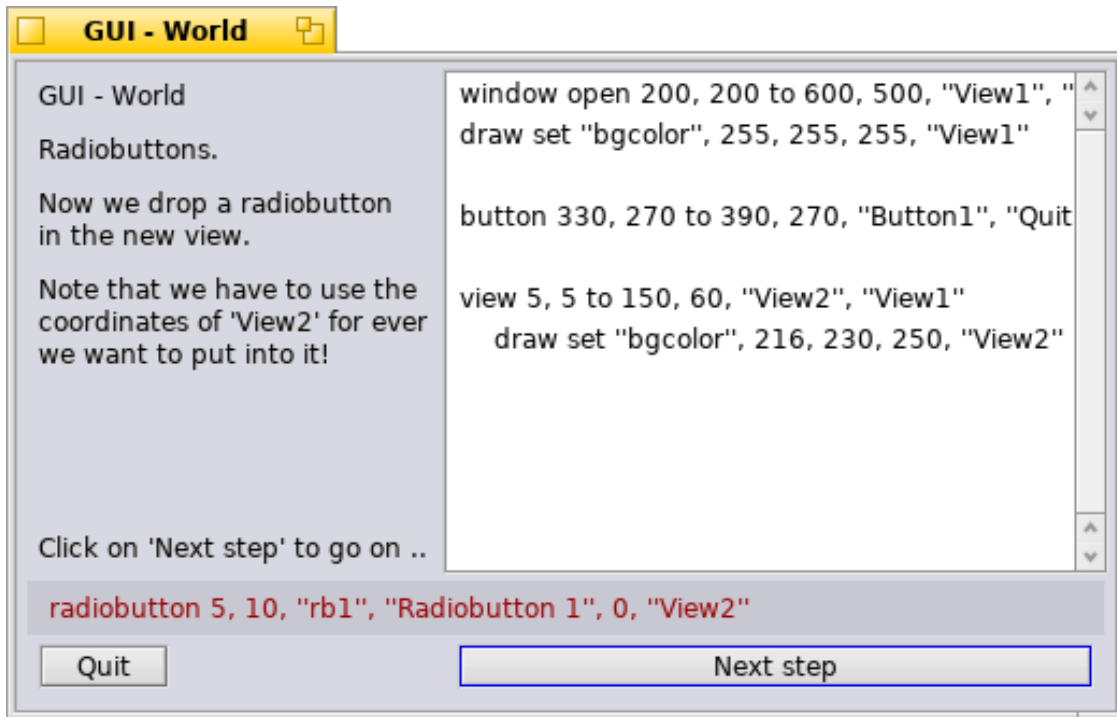
Click on *Create Binary* and after a few seconds you will see the following:



OK, the message says that the binary was created, but where is it? In the same folder where your program was, which in our case is `/boot/home/yab_work/Programs/Examples`. No need to *chmod* it, just double-click it and go.

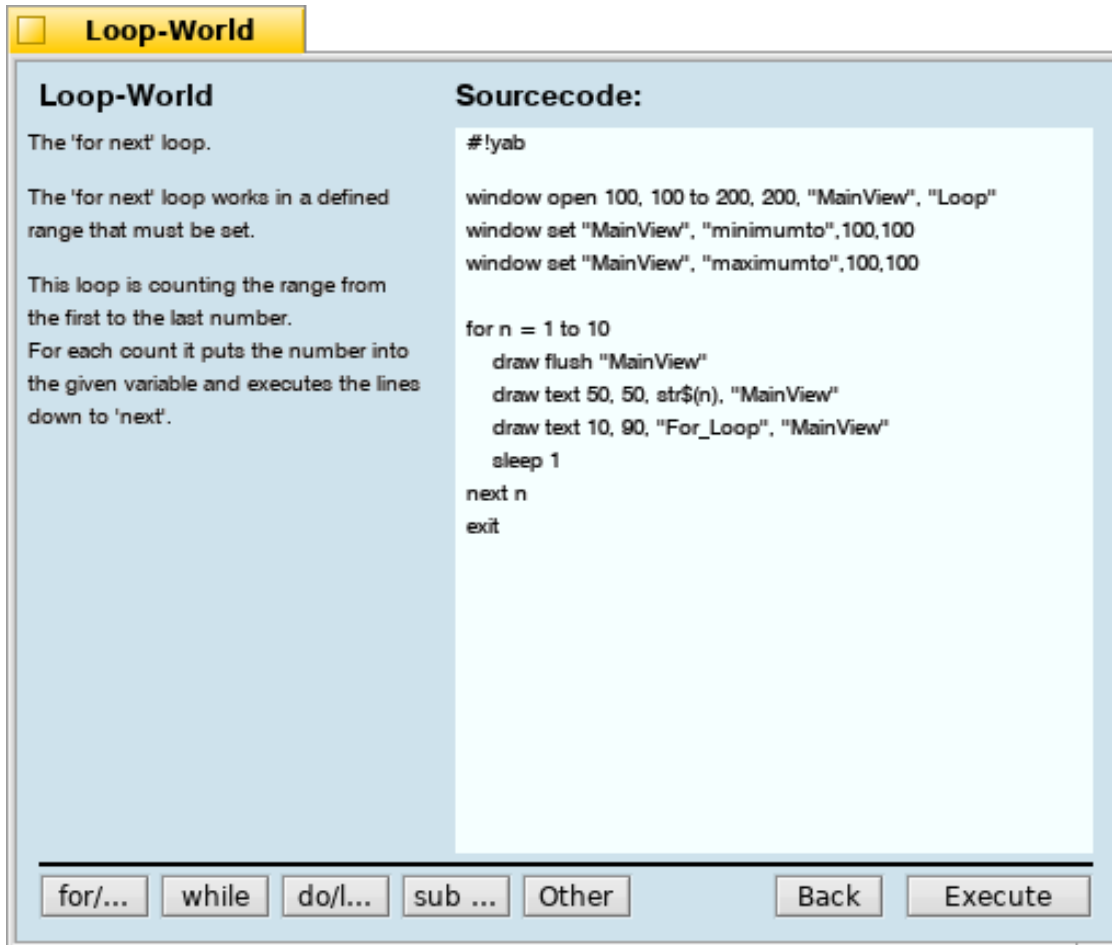
Please note that you still need to have yab loaded to run this binary. More specifically, you need `libyab.so`.

Most of the example programs are demos that require you to look at the code to see how it works. But a few of them are actually self-contained tutorials. *GUI-World.yab* shows you how to construct a window.



The code for *GUI-World* is quite ingenious. It switches strings to numbers using the `VAL()` function, and it does this inside the parameters to another command. This sort of thing can get tricky to disentangle

Loop-World.yab runs you through the loops available in yab. This tutorial program is stricter than we have been so far. We have been discussing IF...THEN and SWITCH command structures as if they were loops, as has been traditional in the BASIC world. But strictly speaking, of course they are not loops but choice control structures.



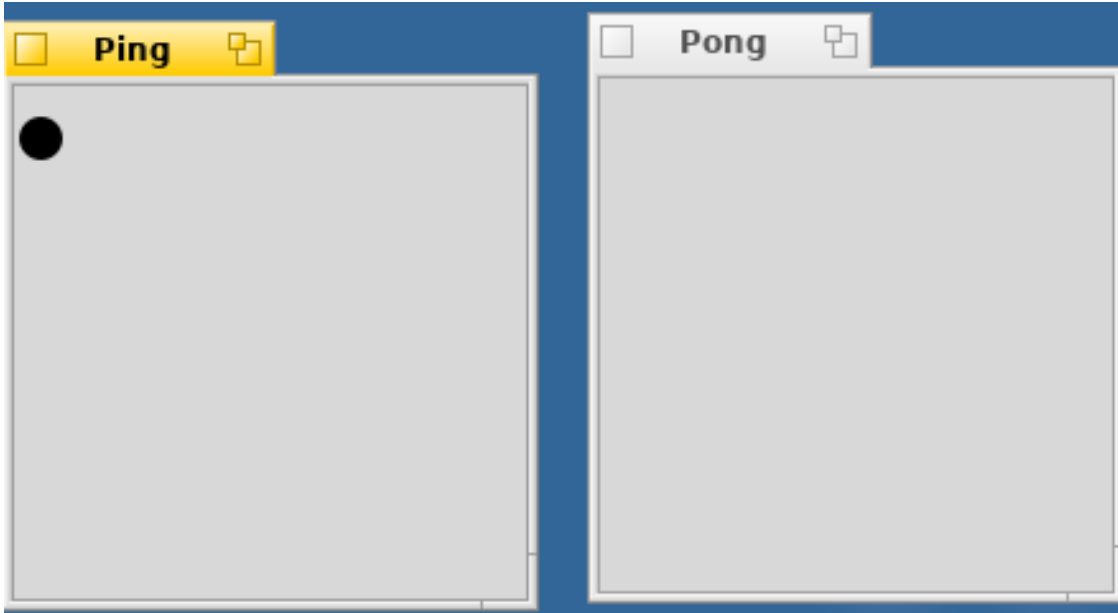
Most of the other apps with "World" or "Demo" in the title are fairly straightforward. Load them into the IDE and start playing with them. For example, *DropWorld.yab* has a dropzone marked by diagonal lines. What would it take to make those lines slant the other way round? Or could you use the technique in *ImageWorld.yab* to put an image on that dropzone instead of the diagonal lines?

Take a few days to play around with these examples in the IDE. If a program uses a command you have not encountered before, use the IDE's Help system to find out what it does. Don't be afraid of breaking things. If you are done and want to get the examples back to the way they were, just uninstall the *yab_ide* HPKG and then reinstall it.

A final word on some of the more interesting Examples. *Calc.yab* is a fully functioning desktop calculator. Run it through BuildFactory and put the binary in */boot/home/config/non-packaged/data/deskbar/menu/Applications* for quick access.

BlockReadWriteDemo.yab doesn't look like much, but it demonstrates the use of the fileblock library (standard with your yab installation). This allows yab to access files in a more database-like way, rather than sequentially reading it line by line. We'll discuss this in more detail in Appendix 1.

Ping.yab and *Pong.yab* demonstrate the important *message send* command. This allows you to send messages from one yab program to another! Sadly, if you want to send a message to a non-yab program, you will have to use something like *SYSTEM("hey " + The_message\$)*. Since the IDE allows only one app to run at a time, you will need to start at least the first instance of Ping by double-clicking it in Tracker. Now start Pong. The ball effortlessly bounces between the two. This demo was considered a big deal back in the BeOS days. Today we can replicate it with a few lines of yab.



There is one Example program left: *Tedit*. Ignore it, for we are going to reconstruct it in the next chapter.

Chapter 4: The Yabadabbadoo IDE

4.1 Introduction

Yabadabbadoo is an alternative IDE for yab written by myself and available on the clasqm repo. It attempts to replicate the old Microsoft QuickBasic editor. Rather than working in one big file, it saves each subroutine as a separate file, then glues all the parts together when the time comes to view, run or bind the program. This makes Yabadabbadoo the better option if your program involves lots of subroutines. I use it for that kind of program, but for more straightforward projects, I return to the official yab IDE. Also, Yabadabbadoo is set for creating GUI apps only. You *can* make a CLI app with it, but you'd have to start out by deleting three quarters of the code that the IDE helpfully inserts for you.

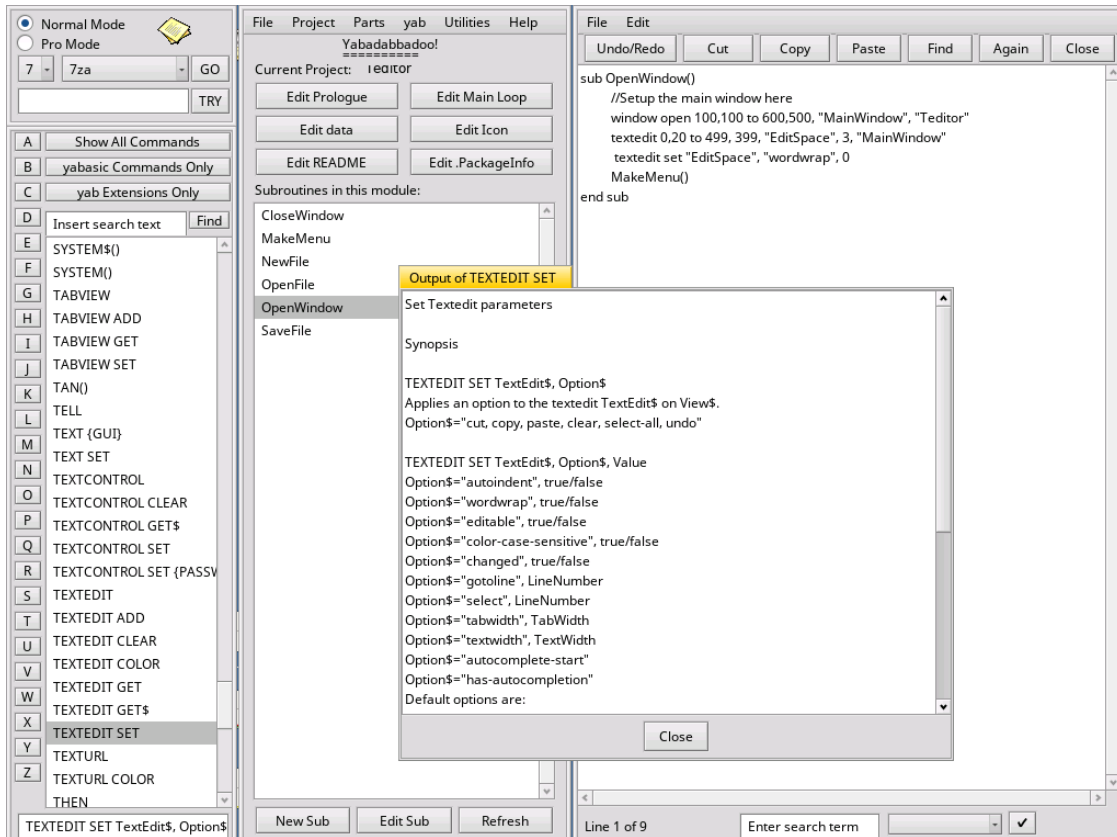
Yabadabbadoo is a deeply personal project: it reflects the way I like to work. Try it. If you don't like it, no hard feelings. It is also Public Domain software and distributed in script form, so feel free to grab any routine you find in there that looks useful to you. If you're feeling generous, give me a shout-out in your program's About box.

The rest of this chapter is adapted and updated from the Tutorial that comes with Yabadabbadoo itself. Let's make ourselves something we can really use!

4.2 Yabadabbadoo Tutorial

To ease you into using Yabadabbadoo, our sample project will be to write a text editor. Why a text editor? No special reason, except that I like writing text editors and yab makes it very easy. When you are done you will have the bare-bones skeleton of a text editor, which you can then customise to include the features you would like to see.

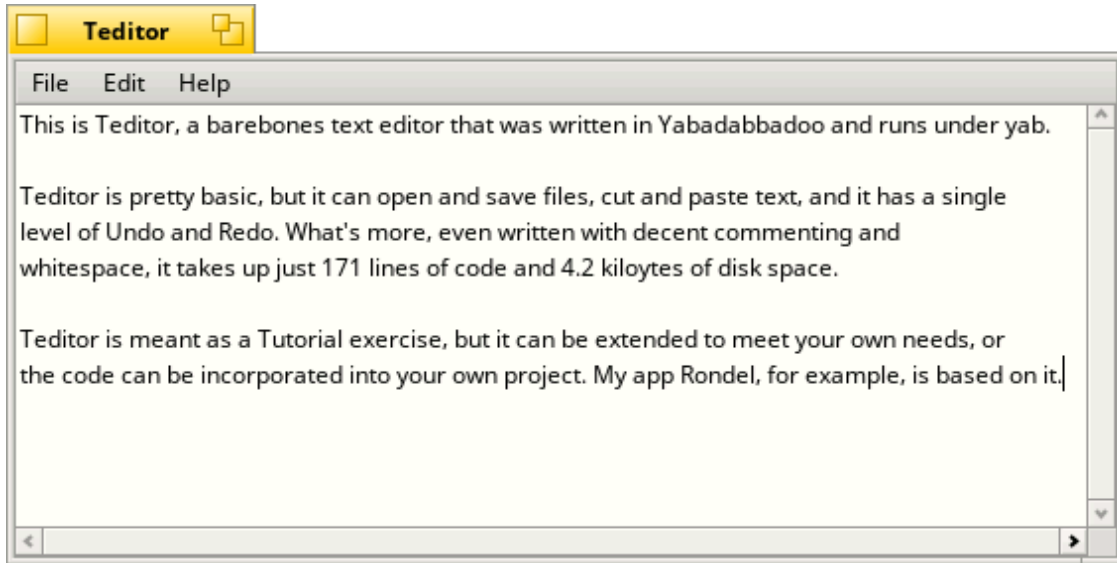
This tutorial will retrace some steps we have taken in previous chapters, but I'll try to keep the baby-talk to a minimum. Just think of it as revision before that important exam. After this chapter, you will be an *Officially Certified* (by me) *yab Programmer* and you'll be on your own, so there's no harm in repeating some of the basics. Also we can drop the line number business now. By now, you should know what I mean if I say "add this to the main loop" .



4.2.1 First steps

Start Yabadabbadoo. The opening wizard lets you create a new project. Create a new directory (anywhere EXCEPT on the Desktop) to house your project and open it. I suggest you make a directory called /boot/home/Projects, and within that, a directory called Teditor. Not a great name, but there's a reason for preferring a name with seven letters in it. You'll see later on. I am not going to give you the full code to start with: look for that at the end of the chapter

When it's done, Teditor will look like this:



Yabadabbadoo now creates a bare framework for a yab application. Depending on the speed of your computer, this may take a few seconds. Once all the buttons and menus become active, try selecting Run from the yab menu. A typical Haiku program window flashes onto the screen.

Of course, this program doesn't actually DO anything yet. It knows how to quit and how to display an About Box and ... well, that's all, actually. Try them out.

Before we do anything else, close the dummy program, and use the stats function in Yabadabbadoo's *Project* menu. Mmmm, 103 lines of code. Actually, if you look at the whole thing (use *yab | View yab file*), a lot of those are blank lines, put there to make the program easier to read. Others, starting with # or //, are comments, little notes you put in for yourself in case you ever need to revisit this code. You can squeeze this program down to 45 lines with no problems at all, and it will still run.

Also, you will see that tabs are used to indicate which commands are in a loop, that is, which commands are grouped together to form a coherent set of instructions that must be completed before the program is allowed to move on. This is called whitespace. yab does not insist on it. But using whitespace consistently is the mark of a good programmer. Learn to do it from the beginning.

in Yabadabbadoo click the *Edit Prologue* button. An editing window comes up. Can you see the following lines?

```
ProgramName$ = "Teditor"
AuthorName$ = "Name of Author"
ProgramVersion$ = "V0.1"
ProgramBriefDescription$ = "Brief Description of your program."
ProgramLicense$ = "Public Domain/Freeware/GPL/Artistic
License/Commercial"
ProgramAcknowledgements$ ="With thanks to ....."
```

Change them to something more like this:

```
ProgramName$ = "Teditor"
AuthorName$ = "Joe Bloggs"
ProgramVersion$ = "V0.1"
ProgramBriefDescription$ = "My unbelievable first Yabadabbadoo
program."
ProgramLicense$ = "Public Domain"
ProgramAcknowledgements$ ="With thanks to clasqm for creating
Yabadabbadoo!"
```

Now run the program from Yabadabbadoo's yab menu again. You don't need to save your changes: it is done automatically. Once Teditor's by-now familiar window comes up, try that About box from the Help Menu once more. Hey! It's different!

If nothing happens:

Most likely you deleted a quotation mark. One thing yab is very good at is wearing out the " key on your keyboard! What you have just entered are string variables and they are always enclosed in quotation marks. Time for our first debug session.

In Yabadabbadoo we debug the old-fashioned way, with a Terminal and a text editor. From Yabadabbadoo's *Utilities* menu, open up Tracker and a Terminal. in the Tracker window Navigate to `/boot/home/Projects/Teditor/binder`. Drop the *Teditor.yab* file you see there onto the Terminal. Click on the Terminal again to make it active and press Enter to run the program. An error message will appear in Terminal. These messages are rather cryptic and they don't always pinpoint the exact line where the problem showed up, but it's what we have. You can close the Tracker window now. The next time you need to run the program you can just hit the Cursor Up key in Terminal to repeat the last action.

But the error message gives a reference to a line in the program as a whole. We are editing it in bits and pieces. That is not such a problem right now since the prologue is the very first part of our program. It does become a problem when you are tracking down an error in a distant subroutine.

One thing you can do is to load Teditor.yab into Pe and scroll down to the offending line. But we are code heroes, so let's do it all in Terminal! Let's say an error was reported on line 14. We want to see the lines just before and after it, just in case yab got a little confused, Type the following command:

```
cat -n Teditor.yab | sed -n '10,18p'
```

and you'll see lines 10 to 18 of your program in the Terminal ... numbered!. Your actual file content will not have changed. Be careful with sed, though. it is a powerful, but fiendishly complicated beast.

Once you have identified the error, fix it in the Yabadabbadoo editor and try to run the program again.

I strongly suggest that you learn how to control the flow of your program this way, but if you're willing to take a chance, try the Run & Display function in the yab menu. This is

experimental: a sufficiently serious bug in your program can and will freeze Yabadabbadoo itself! If that happens, look in your temporary files cache (which should be /boot/system/cache/tmp on a PM system). In it you will find two files starting with YBD that will show you where the problem was.

OK, LET'S GET THE SHOW BACK ON THE ROAD

We'll be moving at a breakneck pace from here on, but in reality, what you will do is make one change at a time, test to see if it is working, then debug until it does.

4.2.2 Changing the window title

A program announcing itself as Main Window doesn't do it for me. Let's change that. Open the OpenWindow subroutine by selecting it from the list and clicking the "_Edit Sub" button (or just by double-clicking it). change the line that reads

```
window open 100,100 to 600,500, "MainWindow", "Main Window"
```

to read

```
window open 100,100 to 600,500, "MainWindow", "Teditor"
```

4.2.3 Creating a text area

A text editor needs a place to write. In yab this is called the textedit widget. Still in the OpenWindow subroutine, start a new line, as follows:

```
textedit 0,20 to 500, 400, "EditSpace", 3, "MainWindow"
```

OK, what is happening here?

In yab, everything happens on top of something called a View. You put buttons, menus, and everything else on top of a View. In very early versions of yab, you first made the main window, then you plastered a View on top of it before you could do anything. But now, the main window is itself a view. This is important to remember, or very little in the yab help files will make sense.

We know that our window is 500 by 400 pixels. But the menu takes the top 20 pixels (from 0 to 19) so we place the textedit widget 20 pixels lower down. Technically it should be 0,19 to 499, 399, but yab is a little forgiving that way. Every widget needs to have a name so we call this one "EditSpace". The 3 that follows (note the lack of quotation marks) indicate that we want both a vertical and a horizontal scrollbar. Finally, we say where we want to put this thing, which is on the View called MainWindow

The next step is up to you. By default, a textedit widget will wrap text as you type it, like a word processor. If that's what you want, fine. But you can turn it off with another line, if you like:

```
textedit set "EditSpace", "wordwrap", 0
```

or alternatively

```
textedit set "EditSpace", "wordwrap", false
```

will work just as well. Hmm, that horizontal scrollbar makes more sense now.

Run the program and enter some text. Now select some text and right-click on it. Surprise! You have Undo/Redo, Cut, Copy and Paste, and Select All available. How is that possible? We haven't programmed it in yet!

This shows the remarkable capabilities yab has inherited from the Haiku API. One statement gives you access to a large amount of functions deep within the operating system. The next step is to put these functions on the menu, along with some way to open and save files.

4.2.4 Pimping the menu

Open the MakeMenu subroutine. Between the two lines that start with "menu" insert the following code:

```
menu "File", "New", "N", "MainWindow"
menu "File", "Open", "O", "MainWindow"
menu "File", "Save", "S", "MainWindow"
menu "Edit", "Undo/Redo", "Z", "MainWindow"
menu "Edit", "Cut", "X", "MainWindow"
menu "Edit", "Copy", "Z", "MainWindow"
menu "Edit", "Paste", "Z", "MainWindow"
menu "Edit", "Select All", "A", "MainWindow"
```

That will change the menus of Teditor and assign standard shortcut keys. Please note that the order in which you issue menu commands does make a difference. The menu items are created in the order you supply them here.

By now, it might be a good idea to make a backup of our project. in Yabadabbadoo, use Project | Backup. Now go and see what has been placed on your Desktop.

4.2.5 Activating the editing functions

Time to enter the scary part of our program: the main loop!

This is the real engine of any yab program, where it spends most of its time waiting for you to do something. Then, when you do, it interprets those actions according to the rules you set down here.

Close the Teditor window if it is still open. Now, in Yabadabbadoo, click on the "Edit main loop" button. In the editing window, at the top you will see some really cryptic commands:

```

    dim msg(1)
while(notleavingLoop)
    nCommands=token(message, msg$(), "|")
    for everyCommand = 1 to nCommands
        if(DEBUG and msg$(everyCommand)<>"") print
msg(everyCommand)
        switch(msg(everyCommand))
            case "_QuitRequested"
            case "MainWindow:_QuitRequested"
            case "MainWindow:File:Quit"
                leavingLoop = true
                break

```

and at the bottom appears:

```

        end switch
    next everyCommand
wend
CloseWindow()
end

```

DO NOT TOUCH THOSE LINES OF CODE! Everything you do in the main loop happens between those sets of lines.

We're going to do the editing functions first, since they are fairly straightforward. We will go down the menus and add a command or two for each one. The main loop doesn't care about the order of these CASE statements. We just keep them grouped together to make future changes easier. The first BREAK statement should be on line 17. Open up a new line immediately below and type in the following:

```

    case "MainWindow>Edit:Undo/Redo":
        textedit set "EditSpace", "undo"
        break
    case "MainWindow>Edit:Cut":
        textedit set "EditSpace", "cut"
        break
    case "MainWindow>Edit:Copy":
        textedit set "EditSpace", "copy"
        break
    case "MainWindow>Edit:Paste":
        textedit set "EditSpace", "paste"

```

```

        break
    case "MainWindow:Edit:Select All":
        textedit set "EditSpace", "select-all"
        break

```

Test it and see.

Let's analyse an easy one. The program called Teditor sends a message. The main loop intercepts it and sees that it comes from the View called MainWindow, from the Edit menu, and from the menu item called Cut (note that there are **no** spaces after the colons). It therefore sends a message to the textedit widget called EditSpace to execute the cut function, which, as we have seen, it already knows how to do. Finally, it executes a BREAK to indicate that it is ready for the next message.

How did we get Undo and Redo on a single menu item? Well, if you undo something and immediately use Undo again, it undoes the undo!

A word about upper and lower case. In yab, variables and commands can be written in upper or lower case, but the names of widgets, data labels and subroutines must be written EXACTLY the way they were first defined. If you try to issue a command to a widget called "EDITSPACE" your program will crash, because it doesn't exist. Your best bet is to pretend that EVERYTHING is case-dependent, develop a coding style, and stick with that.

4.2.6 Saving a file

Losing all our work every time we close Teditor is a drag. Let's activate the Save menu. In the main loop:

```

    case "MainWindow:File:Save":
        SaveFile()
        break

```

This tells Yabadabbadoo to execute a subroutine called SaveFile. Don't bother testing it now. It doesn't exist yet.

We can actually do this with a few lines in the main loop, but a subroutine is neater and you need the practice. Click the *New Sub* button at the bottom of Yabadabbadoo's main window and enter the name of the new subroutine (just SaveFile, no brackets). The new subroutine appears at the bottom of the list, already selected. Now click the *Edit Sub* button. The new subroutine appears in your editor window. Enter the following lines between

```
sub SaveFile()
```

and

```
end sub
```



```

local file2save$, filename$

    file2save$ = textedit get$ "EditSpace"

    if file2save$ <> "" then

        filename$ = filepanel "save-file", "Save File As ...", "/"
        boot/home"

        open filename$ for writing as #1

        print #1 file2save$

        close #1

    endif

```

What's happening here? Back in the prologue we introduced some variables. Those were global variables: the entire program knows about them. Having too many global variables is not good programming practice. Global variables take up unnecessary memory and slow down the program. So here we have two local variables. They only exist while this subroutine is running, then they are gone until the next time you want to save a file. They can even have the same names as global variables (this is NOT recommended. But it can be done).

We take the content of the textedit widget and pour it into the string variable called file2save. Next we check if there is anything in there. We don't want to waste our time saving an empty file. If there is something in there, we go ahead and call up a filepanel to put in a name and directory to save the contents to. Then we open the file in channel 1. The OPEN command has many forms. This happens to be the one I am comfortable with.

Once a channel has been opened, we can divert the print command, which normally prints stuff to the Terminal screen, to the file. We then immediately close that channel. NEVER, EVER leave a channel open any longer than you have to!

But wait! suppose you start saving a file, then change your mind and click Cancel on the filepanel. Try it now.

It crashed, didn't it? Can you work out why?

The next line tries to open an empty string (you cancelled, so there is no filename) as a channel. That is an illegal instruction. You can fix that by inserting the following line just before the one that starts with OPEN

```

    if filename$= "" return

```

That RETURN means "get me out of this subroutine completely!"

OK, technically, this is a Save As rather than a Save. It throws up a filepanel every time,

even if you're working on a file you have already named and saved. I'll leave it to you to think about how to fix that. Here's a clue: put the filename in a global variable. Let's move on.

4.2.7 Starting a new file

This is a little trickier. What if there is already something in the `texedit` widget? We need to ask the user if it should be saved first. Then we clear out the crud and start with a fresh widget. If there is nothing, we do nothing. We could start another instance of `teditor`, but that is beyond the scope of this tutorial.

Remember that `SaveFile` subroutine we just created? Who says you can only call it from the main loop?

In the main loop:

```
case "MainWindow:File:New":
    NewFile()
    break
```

In the `NewFile` subroutine:

```
local sos
if textedit get$ "EditSpace" <> "" then
    sos = alert "Save Current Text First?", "Yes", "No", "",
"none"
    if sos = 1 SaveFile()
    textedit clear "EditSpace"
endif
```

Just one new thing here: the local variable `sos`, which we use to capture user input from the alert dialog, is not a string variable but a numeric variable. Strings and numbers are about as complicated as it gets in `yab`. Other languages may have a dozen or more kinds of variables.

4.2.8 Opening a file

This is the last part of our program. By now you should be able to figure out what to do in the main loop. In the new subroutine you are going to create (you decide on the name, but be sure to use the same one back in the main loop!), your code should read

```
local file2open$, anewline$
NewFile()
file2open$ = filepanel "load-file", "Open Which File?",
"/boot/home"
if file2open$ <> "" then
    open file2open$ for reading as #1
    while(not eof(1))
```

```

        line input #1 anewline$
        textedit add "EditSpace", anewline$ + "\n"
    wend
    close #1
    textedit set "EditSpace", "gotoline", 1
endif

```

But I'm not going to tell you what each part does, It's time to start working these things out yourself, line by agonising line. Welcome to the world of programming.

Before we go: As it stands, Teditor will Quit leaving unsaved changes behind. How would you make sure it didn't do that any more, by inserting a single line of code? By now, you should be able to work it out.

4.3 Wrapping up your project

My *stats* dialogue says 165 lines of code. That is not an exact measure: I cleaned up some empty lines as I went along. Yours might be slightly more. Also, we changed a few existing lines. Still in 70-odd lines of code we went from a bare window that did nothing to a functioning text editor! Take that, C++!

There is a LOT that can still be improved about Teditor. Trying to load something that is not a text file will have ... interesting results. But we've made a start. It works well enough that you can use it. Now let's take it to the next level.

Open up the Teditor.yab file in the binder directory in Pe and take a look at the first few lines. Line 1 should read either

```
#!yab
```

or

```
#!yab4ybd
```

yab4ybd was a custom form of the yab executable that was bundled with early versions of Yabadabbadoo. At the time of writing it lives on as a symbolic link to yab, for backwards compatibility purposes,

This line is a special format in the UNIX world that tells a script file (which is what your program is at this point) where to look for its interpreter. If it is found, the script will look for that executable in the PATH (a standard set of directories: in Haiku they all tend to end with /bin). Yabadabbadoo sets this up for you already.

The other thing a script needs to be is *chmodded*: the system needs to be informed that this is a script, not just another piece of text. Again, Yabadabbadoo sets this up for you, but if you edit your program in another editor, you may need to know how this works. It's easy. In Terminal:

```
chmod +x Teditor.yab
```

If you run Teditor this way and then another yab app you have written, you will see something odd on the Deskbar: they share a single entry. There is a fix for this: insert a new line 2 in Teditor.yab:

```
mimetype "application/x-vnd.Teditor"
```

Well, that is supposed to work. I haven't had much success with it myself. In any case, this does not work when the yab app is bound, which is where we are going next. There is another fix for that.

You are very proud of your program. You want to put it up on your website or a Haiku software repository so other people can use it. Great! But how do you know if they have the yab executable that your script needs to run? You don't.

With Package Management, you could bundle a version of the yab executable (call it yab4teditor) and run your program against that. Better to "bind" your program. This means that you take the executable and your script and make a single file out of them. This is why yab apps are always so much bigger than other Haiku apps, by the way. But hard drives and RAM are cheap these days, so let's not worry about it.

You can do the binding from Terminal, but Yabadabbadoo had a built-in function that will bind your program and put it on the Desktop for you, ready for testing. Look for it in the menus.

When you have tested your new bound yab app exhaustively, how do we make it run in its own space in the Deskbar? Open the bound file in DiskProbe and search for *yab-app*. You now need to replace *yab-app* with a description of your own, and you have exactly those seven characters to work with! What do you know? There are seven characters in Teditor! Save the file and exit DiskProbe.

Now right-click on the bound file and select Add-ons, then FileTypes. If you bound your app from Yabadabbadoo, the signature should be all settled. Ignore the Application flags. Fill in the Version and description fields and drag in a nice icon. Save the settings. Your application is now ready to be packaged into a .zip or .hpkg file for distribution.

We have only been able to touch on a few of yab's many commands in this tutorial. For more, go to Yabadabbadoo's Utilities menu and play around with the yabAssistant.

4.4 Wrapping up your project - part two

What we have done up to now is the hard way. I made you go through it because it will help you understand what is going on. But there is an easier way: in the *yab* menu there is an entry called "Transfer to Official IDE" that will open the yab IDE and the binder folder where your complete yab script resides. You can now load this into the official IDE and use the BuildFactory facility. Now, you are no longer restricted to seven characters: you can give your program the signature *application/x-vnd.JoeBlogs_MyGreatTextEditor* if you like.

Keep one thing in mind, though: any changes you make while in the official IDE will NOT be transferred back to Yabadabbadoo. This is strictly a one-way trip.

The next version of Yabadabbadoo will likely have a direct connection to the BuildFactory.

But what is the difference between binding and the BuildFactory? From the perspective of the end user, not much. But from the programmer's perspective: a great deal. Not only are you not restricted to seven-character program names, but also the way BuildFactory operates is that it compiles a new version of the yab executable that contains your program, unlike binding where the two are just glued together. With a bound program, anyone with a hex editor can track down your code and extract it. With the Buildfactory this is not possible. The BuildFactory also removes unneeded lines, like blank lines and comments, before it starts, so your final product will typically be a little slimmer.

Whether you bind your program or use the BuildFactory, your user will still need the library libyab.so, which is bundled in the yab HPKG file. We have discussed this in earlier chapters.

4.5 Teditor - full code

```
#!/yab
#####
##### Prologue #####
#####

//Yabadabbadoo notification
#####DO NOT RENAME THIS FILE!#####
//Yabadabbadoo needs it to function.

##Fill in these fields with your own particulars.
##The variables will be used in the About Box and in naming the
program.

ProgramName$ = "Teditor"
AuthorName$ = "Joe Bloggs"
ProgramVersion$ = "V0.1"
ProgramBriefDescription$ = "My unbelievable first YBD program."
ProgramLicense$ = "Public Domain"
ProgramAcknowledgements$ = "With thanks to clasqm for creating
YBD!"

//*****
//*****Global Variables*****
//*****

## Technically, yab does not require you to declare global
variables,
##It just is a really, really good idea to do it anyway.
```

```

// set DEBUG = 1 to print out all messages on the console
DEBUG = 1
//change this to DEBUG = 0 when you are ready to bind the program
for distribution

#####
#####Preliminary Commands#####
#####

## Commands to run before the Main Loop come here.
## e.g. setting up a window with a menu.

OpenWindow()

#####End of Prologue#####

//Yabadabbadoo notification
#####DO NOT RENAME THIS FILE!#####
//Yabadabbadoo needs it to function.

//Main Message Loop
dim msg$(1)
while(not leavingLoop)
    nCommands = token(message$, msg$(), "|")
    for everyCommand = 1 to nCommands
        if(DEBUG and msg$(everyCommand)<>"") print msg$(
everyCommand)
        switch(msg$(everyCommand))
            case "_QuitRequested":
            case "MainWindow:_QuitRequested":
            case "MainWindow:File:Quit":
                leavingLoop = true
                break
            case "MainWindow:File:Save":
                SaveFile()
                break
            case "MainWindow:File:New":
                NewFile()
                break
            case "MainWindow:File:Open":
                OpenFile()
                break
            case "MainWindow:Edit:Undo/Redo":
                textedit set "EditSpace", "undo"
                break
            case "MainWindow:Edit:Cut":
                textedit set "EditSpace", "cut"
                break
            case "MainWindow:Edit:Copy":
                textedit set "EditSpace", "copy"
                break

```

```

        case "MainWindow:Edit:Paste":
            textedit set "EditSpace", "paste"
            break
        case "MainWindow:Edit:Select All":
            textedit set "EditSpace", "select-all"
            break
        case "MainWindow:Help:About":
            Alert ProgramName$ + " " + ProgramVersion$ + "\n"
+ "by " + AuthorName$ + "\n\n" + ProgramBriefDescription$ + "\n" +
ProgramLicense$ + "\n" + ProgramAcknowledgements$, "OK", "none"
            default:
                break
        end switch
    next everyCommand
wend

CloseWindow()

end

sub CloseWindow()
    //Close down the main window
    NewFile()
    window close "MainWindow"
end sub

sub MakeMenu()
    //Create menu in MainWindow
    menu "File", "Quit", "Q", "MainWindow"
    menu "File", "New", "N", "MainWindow"
    menu "File", "Open", "O", "MainWindow"
    menu "File", "Save", "S", "MainWindow"
    menu "Edit", "Undo/Redo", "Z", "MainWindow"
    menu "Edit", "Cut", "X", "MainWindow"
    menu "Edit", "Copy", "Z", "MainWindow"
    menu "Edit", "Paste", "Z", "MainWindow"
    menu "Edit", "Select All", "A", "MainWindow"
    menu "Help", "About", "", "MainWindow"
end sub

sub NewFile()
    local sos
    if textedit get$ "EditSpace" <> "" then
        sos = alert "Save Current Text First?", "Yes", "No", "",
"none"
        if sos = 1 SaveFile()
            textedit clear "EditSpace"
        endif
    end if
end sub

sub OpenFile()
    local file2open$, anewline$

```

```

        NewFile()
        file2open$ = filepanel "load-file", "Open Which File?",
"/boot/home"
        if file2open$ <> "" then
            open file2open$ for reading as #1
            while(not.eof(1))
                line input #1 anewline$
                textedit add "EditSpace", anewline$ + "\n"
            wend
            close #1
            textedit set "EditSpace", "gotoline", 1
        endif
    end sub

sub OpenWindow()
    //Setup the main window here
    window open 100,100 to 600,500, "MainWindow", "Teditor"
    textedit 0,20 to 499, 399, "EditSpace", 3, "MainWindow"
    textedit set "EditSpace", "wordwrap", 0
    MakeMenu()
end sub

sub SaveFile()
    local file2save$, filename$
    file2save$ = textedit get$ "EditSpace"
    if file2save$ <> "" then
        filename$ = filepanel "save-file", "Save File As ...", "/"
boot/home"
        if filename$= "" return
        open filename$ for writing as #1
        print #1 file2save$
        close #1
    endif
end sub

#####
###DATA statements, if any, come here###
#####

```

4.6 Yabadabbadoo FAQ

Like most lists of Frequently Asked Questions, these are largely made up! Send me some real questions and they will show up here in future releases.

Q: Can I bind a yabadabbadoo project from Terminal?

A: Yes, of course, as long as you have a yab executable in your PATH. We will use Yabadabbadoo itself as an example. Use the following bash script.

```
#!/bin/sh
```



```

rm -f /boot/home/Projects/Yabadabbadoo/binder/Yabadabbadoo.yab
touch /boot/home/Projects/Yabadabbadoo/binder/Yabadabbadoo.yab
cat /boot/home/Projects/Yabadabbadoo/prologue >
/boot/home/Projects/Yabadabbadoo/binder/Yabadabbadoo.yab
cat /boot/home/Projects/Yabadabbadoo/mainloop >>
/boot/home/Projects/Yabadabbadoo/binder/Yabadabbadoo.yab
cat /boot/home/Projects/Yabadabbadoo/sub/* >>
/boot/home/Projects/Yabadabbadoo/binder/Yabadabbadoo.yab
cat /boot/home/Projects/Yabadabbadoo/data >> /boot/home/Projects/
Yabadabbadoo/binder/Yabadabbadoo.yab
chmod +x /boot/home/Projects/Yabadabbadoo/binder/Yabadabbadoo.yab
mimeset -f
/boot/home/Projects/Yabadabbadoo/binder/Yabadabbadoo.yab
chmod +x /boot/home/Projects/Yabadabbadoo/binder/Yabadabbadoo.yab
yab -bind /boot/home/Desktop/Yabadabbadoo
/boot/home/Projects/Yabadabbadoo/binder/Yabadabbadoo.yab
chmod +x /boot/home/Desktop/Yabadabbadoo
mimeset -f /boot/home/Desktop/Yabadabbadoo
settype -t application/x-vnd.Yabadabbadoo
/boot/home/Desktop/Yabadabbadoo

```

Just change the directories and filenames to whatever you are using. A search-and-replace changing Yabadabbadoo to the name of your project should do the job. Then just change /boot/home/Projects to wherever you keep your projects.

Maybe it's just easier to create the yab file from the menus?

Q: Do you use Yabadabbadoo to work on Yabadabbadoo?

A: Yes, I have used Yabadabbadoo to work on itself, but using Run then throws a second copy on the screen, which gets confusing in a hurry ... I love IDEs. I wrote one. But if you can't get things done with just a terminal and a text editor then you are not programming the computer, the computer is programming YOU!

Q: I've created a bunch of subroutines and now they are all at the bottom of the list! How do I get them in alphabetical order?

A: Click the Refresh button.

Q: What about libraries? Do they still work?

A: Sure they still work. The problem is finding them. At the time of writing, yab looks for libraries in two locations. One is the same directory as the main yab file, which for us is the binder directory, not really very useful, but you could symlink your library there. The other is /boot/home/config/settings/yab. Older versions of yab used a location that is no longer writable under Package Management.

But consider this: if you need to re-use a subroutine, just use Tracker to copy or symlink it from one project's sub directory to another. The need for libraries just isn't what it used to be.

For example, I have two projects: Trope is a general purpose text editor, while Rondel is

a more specialised editor that will deal with Markdown annotations. The general purpose subroutines from Trope are symlinked into Rondel's subs folder. Whenever I improve Trope, Rondel automatically picks up those improvements. *and vice versa*.

Q: will Yabadabbadoo ever automate the creation of hpkg and/or zip containers?

A: No. There are too many intangibles, too many things I can't predict. My design philosophy is to have a single executable that creates its support files on the fly. But you might prefer to ship your program with lots of support files in hard-coded subdirectories. I think you're wrong, but that's a discussion for another day.

But I can't predict what you will put where, so you will just have to do it yourself. Sorry. Every Yabadabbadoo project contains a bare-bones PackagingFolder directory to get you started. That's as far as I will go with that.

Q: Will you create a framework to make Yabadabbadoo available in other languages?

A: No. But this is Public Domain software. If you really want a Serbo-Croatian version, go ahead and make one. You have about 10 KLOCS of DATA statements to translate and one quotation mark out of place will crash the program. Good luck. Don't forget to escape quotation marks and real backslashes.

Chapter 5: Learning more about yab

At the time of writing, there are two resources where you can learn more about yab.

5.1 The BeSly database

This is a multilingual resource, not only for yab, but on a number of Haiku-related issues. Find it at <http://besly.de>. In the left sidebar, click on *Development* and then on *yab*. There is also an application called the BeSly Tutorial Finder that accesses the site from your desktop. Find it on the BeSly repo or [here](#).



The only drawback is that a lot of the articles here were originally written in German and are then translated into other languages by various people with various levels of translation skill (and Google translate seems to have helped out too). Your best experience will be if you can read the original German. But things are improving and Besly is already a very important yab resource.

A compendium of BeSly material in German can be found [here](#) or in PDF format [here](#).

BeSly member *lorglas* has also written a guide to yab in German. Download it [here](#) or [here](#).

5.2 The yab discussion forum

This is an English-language discussion group where yab programmers can announce their latest projects, ask for help with their code, daydream about future enhancements to the language and generally just hang out. Find it here:

<http://yab.orgfree.com/forum/index.php>

The screenshot shows a web browser window titled "yab | yet another Basic for HAIKU - WebPositive". The address bar displays "http://yab.orgfree.com/forum/index.php". The page header includes a navigation menu with "Search", "Member List", "Calendar", and "Help". A greeting "Hello There, Guest! (Login — Register)" and the current time "12-17-2016, 05:03 PM" are shown. The main content area features two forum sections. The first section, "Programming in yab", lists two forums: "Help with programs" (6 threads, 25 posts) and "Snippets" (18 threads, 62 posts). The second section, "installing / troubleshooting / feature requests", lists two forums: "Installation" (4 threads, 8 posts) and "Fixing yab" (5 threads, 28 posts). The status bar at the bottom indicates "http://yab.orgfree.com/forum/index.php finished".

yab | yet another Basic for HAIKU

Search Member List Calendar Help

Hello There, Guest! (Login — Register) Current time: 12-17-2016, 05:03 PM

yab | yet another Basic for HAIKU

Programming in yab

| Forum | Threads | Posts | Last Post |
|---|---------|-------|--|
| Help with programs This is a place to get help with programming problems. | 6 | 25 | 2 array compare 03-27-2016 10:12 PM by clasqm |
| Snippets Small chunks of code that may help others. | 18 | 62 | More spinner stuff. 05-20-2016 12:03 AM by clasqm |

installing / troubleshooting / feature requests
This is where to get help with installation and fixing problems with yab / yab-IDE installs.

| Forum | Threads | Posts | Last Post |
|---|---------|-------|---|
| Installation help with trouble installs | 4 | 8 | yab 1.7.5.3 04-29-2016 03:27 AM by bbjimmy |
| Fixing yab | 5 | 28 | Error in hrev 50028 |

http://yab.orgfree.com/forum/index.php finished

You will need to register with the site. I check it regularly, so feel free to start a new thread complaining about this e-book! That way, the next edition will be better.

There is a yab Facebook group, but it has not been very active lately.

About the author

Michel Clasquin-Johnson is the guy down the corridor whose actual job title has nothing to do with computers, but who always gets called upon when things break. He is the author of several books, e-books and academic articles, but this is the first that is programming-related.

"Programming with yab" was written over four weeks in the Christmas holidays while Michel was minding the otherwise deserted office. Yes, I know, it shows. Sorry.

For more Haiku-related activity by Michel Clasquin-Johnson, follow his dispatches from the depths of Haikustan at <http://clasquin-johnson.co.za/michel/haiku/blog/>

About this e-book

"Programming with yab" is published at Smashwords as a FREE e-book, and always will remain free. I encourage you to share copies with your friends.

However, if you have obtained your copy from someone else and enjoyed it, please consider signing up with Smashwords and downloading an official copy. That way, you will be notified by email whenever a new edition comes out. This e-book is intended to be a living document. As Haiku and yab grow and develop, the e-book will also be augmented and rewritten.

And if you contribute some cool idea on the yab forum (see chapter 5), I may ask your for permission to use it in the next edition!

This e-book was written in markdown format using the Rondel text editor on Haiku.

Appendix 1: Database-like functions in yab

Level: Advanced

by Jim Saxton

Originally published on <http://yab.orgfree.com/forum/> - 2 February 2016, reprinted with permission

The *fileblock* library brings database-like functions to yab. This allows one to read data from a large file and replace just one portion of the file, a record or even just one field of a record.

fileblock.yab information - Exported subroutines:

openfile (fil\$, recordlength, numberoffields)

Opens a file for block access, one must plan ahead.

- *fil\$*... The name of the file to open.
- *recordlength*... The size of the block or record.
- *numberoffields*... The number of data elements that each block contains.
Returns the filehandle number to identify this file for future actions.

Note: One may open the same file, *fil\$* more than once and use different number of fields for each instance. Thus allowing one to have different types of data elements stored in the same data file. The *recordlength* must remain the same for all instances.

Field(filehandle,fieldnumber, fieldname\$, fieldlength)

Specifies the length in bytes and name of each data element.

- *filehandle*. .. the number that identifies the file.
- *fieldnumber* ... the number of the field starting at 1
- *fieldname\$* ... the name to identify this field with.
- *fieldlength* ... The number of bytes allocated to this data element.
- Returns 1, ok or 0, bad field number

Note: One must have a field statement for each data element in the block.

lset(filehandle,fieldname,data)

rset(filehandle,fieldname,data)

Complimentary statements for placing data into a field buffer for writing to a record. lset places the data on the left end of the field and rset places it on the right end. if the data passed in is larger than the field, it is truncated to fit. If it is shorter than the field, the extra spaces are filled with the space character, " ". either after, lset or before, rset the data.

- *filehandle* ... The number that identifies the file.
- *fieldname\$* ... The name of the data element.
- *data\$* ... The information to be stored.

write_block(recordnumber, filehandle)

Writes or over-writes a block (record) to the data file with the data in the buiffer that was set with lset, rset or read_block.

- *filehandle* ... the file identifier.
- *recordnumber* ... The number of the record in the file (the first record is record 0, the next record is one etc.) if recordnumber is -1 the record is appended to the end of the file.

read_block(recordnumber, filehandle)

Reads the data of the record *recordnumber* into the data buffer.

- *recordnumber* ... the record to read
- *filehandle* ... the identifier for the file.
Returns the number of the next record.

readfield(*File*, *fieldname*) Reads the data held in the buffer for *fieldname*

- *File* ... The file identifier
- *fieldname* ... The name of the field to read.

lof(filehandle) numberofrecords(filehandle) Returns the number of records in the file.

filehandle ... The file identifier.

Appendix 2: Preparing yab apps for distribution.

Level: Intermediate

by Jim Saxton

Originally published on <http://yab.orgfree.com/forum/> - 10 July 2015, reprinted with permission.

Give your app a signature

When packaging a yab-app for distribution it is best to give your app its own app Application Signature.

From the IDE, replace *application/x-vnd.yab-app* with *application/x-vnd.yourappname* when using the Buildfactory.

From the command-line., use:

```
BuildFactory.yab outputfile inputfile application/x-  
vnd.yourappname
```

This will allow your app to not be hidden in the Deskbar application listing by another yab-app.

Apply an RDEF file to your app.

This is a plain text file that defines resources for an application. In this case, we want to add an icon and application version information as shown in the FileType tracker add-on applet.

An example rdef file:

```
resource vector_icon {  
$"6E6369660E0500020006023C43C6B9E5E23A85A83CEE414268F44A445900C6D  
7"  
$"F5FF6B94DD03EC66660200060238C5F1BB105D3DFDC23B9CD045487847B5070  
0"  
$"FFFFFFFFC1CCFF020006023B3049396B0ABA90833C646E4A101543299500FFF  
F"  
$"FFFFEBEFFFF020006023C71E33A0C78BA15E43C7D2149055549455700E3EDFFF  
F"  
$"9EC2FF03FFACAC0200060239D53438FFCBBC1973C666F4ADC3246DC6C00C1C  
C"  
$"FFFFFFFFF03003CB0020006023C0AE63B3927BC611E3D03FF4C25624A1A960  
0"  
$"A3043CFFFF90AF03C93B3B030D296402000602BD498B3E1159BF219BBE7D2F4  
C"  
$"1B8F4A331300BD0F0FFFE98484040174100A08325E385E40564E5E545E60505  
8"  
$"4C3E510A062E2C2E3E3E454A3C4A2A3A250A042E2C2E3E3E453E320A042E2C3  
E"  
$"324A2A3A250A043E323E454A3C4A2A0A0338423C4D3C440A0622422254325C3  
E"  
$"513E402E3A0A0422422254325C32490A04224232493E402E3A0A043249325C3  
E"  
$"513E400A063E423E544E5C5A505A3F4A390A04C222C20F4E495A3F523C0A043  
E"  
$"42C222C20F523C4A390A054151C08BC8834E5C4E49C22AC2130A053E423E54C  
0"  
$"8BC8834151C22AC2130A044E494E5C5A505A3F110A0D0100000A0001061815F  
F"  
$"01178400040A00010618001501178600040A010107000A080109000A0B01052  
0"
```



```
$"20210A050108000A00010A1001178400040A02010D000A0A010E000A0902040
F"
$"000A06010B000A0C010C000A0001011001178400040A030102000A040103000
A"
$"07010400"
};
```

```
resource app_signature "application/x-vnd.yab-IDE";
resource app_version {
major   = 2,
middle  = 2,
minor   = 5,
variety = B_APPV_FINAL,
internal = 0,
short_info = "yab IDE",
long_info = "An integrated development environment for yab."
};
resource app_flags 1;
```

- The vector icon info is generated by Icon-O-Matic when an icon is exported as "HVIF RDef".
- Replace *"resource() #VICN' array"* with *"resource vector_icon"*.
- The resource *app_signature* should match your *"application/x-vnd.yourappname"*.
- *resource app_version* is the version number information.
- *resource app_flags* is the launch flag, Single launch, Multiple launch, etc.

Use the *RdefApply* yab script included in the */boot/home/yab_work/BuildFactory* directory.

Usage:

```
RdefApply <RdefFilename> <yourappbinary>
```

This will apply the Rdef to your compiled binary. If this is done, it doesn't matter if the file is unzipped or copied to a file system that doesn't support attributes, the attributes, version launch flag and icon, are resources of the binary and applied to it by the OS when the file is launched. That is not the case if you apply these qualities with the FileType Tracker add-on.

Appendix 3: Why you should be writing apps for Haiku

Michel Clasquin-Johnson

Based on a blog post from 2014 on <http://clasquin-johnson.com/michel/haiku/> and reprinted with permission (from myself)

Level: Beginner

You're not going to get rich out of this. Let's just get that out of the way so we can move on to saner ground. If you are hoping to retire at the age of twenty-five, go forth and write iPhone games, and good luck.

But if you are a computer hobbyist, if you simply enjoy playing with bits and bytes, if you just like to write a program for the thrill of seeing it work, and putting it online in case someone else finds it useful, then you should seriously consider writing apps for Haiku.

Hai-what? Glad you asked. Haiku is an open-source x86 operating system that tries to recreate the look and feel of the BeOS that you may remember from the 90s. Think of it as "like Linux, but back when Linux was still fun, before the men in the suits took over." It is largely POSIX-compatible, but not based on a Linux or BSD kernel. The file system has flat-file database built in. It does not have a "please wait" cursor built in, because it runs fast and smooth, and app sizes tend to be small. I'm not going to sing Haiku's praises in this post. Just go to <http://haiku-os.org> and read up on it.

Compared to Linux, this is a tiny community, where everybody knows everybody else. On the other hand, it is pretty much the last man standing out of what was once a flourishing field of hobbyist operating systems. Once-promising OSs like AtheOS are barely surviving. The Haiku community, like your favourite bickering relatives, keeps going on doggedly.

So why should you be writing apps for Haiku? For one simple reason: you can be the first. Does Windows need another flat-file database? Does MacOS need another wall-paper utility? Does Linux need another text editor? Of course not. Those systems have been around for decades and it is really, really hard to think of a new sort of app for which there aren't already a dozen well-established alternatives.

Haiku needs **everything**. Can you write a word processor? This is your chance to be a hero. Write something that will read and write the common document formats and do the basic word processing functions on Haiku, and everybody in the community will install it. You have the chance to be the first, to be the author of that program that all the other programmers have to compete against.

You like to write tools and utilities? Have I got a project for you. Python runs on Haiku, but only in text mode. There was an app called Bethon from the BeOS days that gave Python hooks into the API, so you could write GUI apps with Python. Bethon desperately needs an overhaul. People are trying to create API hooks for Lua and FreePascal. Those people need your help.

You don't want to start from scratch, but you do like to take something that already

exists, see how it works and fix it? There are hundreds of old BeOS programs, with source code available, that just need a little attention to become functioning Haiku citizens again. Or you can port QT4 and Java apps - more about that later.

You can do this on many levels. If you like to be all indie and just put up your stuff on your own website, do that. You can put your app on one of the third-party software repositories for wider distribution. You can join haikuporter and get your apps into the official repository. Or you can start working on the OS itself. Sadly this is where we tend to lose app developers - they get sucked into the main project.

OK, what do you get to work with? Let's start with writing small command-line utilities. You get C, C++ and Python out of the box. Perl, Lua and Ruby are easily available. The command line, by the way is the same bash shell you already know from Linux, BSD and Mac OSX. FreePascal can be installed. The Go language is being ported. There are a variety of BASIC interpreters and if you really insist, you can install oddities like Cactus and Gwydion Dylan. If you can write in Assembly Language, you probably don't need to read this article, but yes, assemblers can be installed.

Things are more restricted if you want to write Graphical User Interface (GUI) programs. The official way is to write in C++. That is what the Haiku API is really aimed at. And we're talking about programming the old-fashioned way, with a text editor and a terminal. There are half a dozen graphical IDEs from the BeOS era in need of porting. You want one, pick one up and fix it. Please?

There is an excellent tutorial available, called *Learning to Program with Haiku*, which takes you from absolute beginner to C++ Haiku programmer. Actually the course and the never-officially-published sequel are freely downloadable, but just go to <http://lulu.com> and buy the thing, OK?

Your other option is yab. This is a version of Yabasic with Haiku API hooks, and it is entirely addictive. Hey you, stop turning up your nose. Yes, it is BASIC, but it is a modern BASIC: no line numbers, no GOTOs. yab has only recently been accepted into the official repositories. Try it, you'll be surprised at the way you can throw up a bare-bones, but fully functioning graphical text editor with 130 lines of code.

With C++ you can write any kind of program. yab is a little more restricted. You shouldn't try to write a graphics-heavy game with it. But the yab community is continually stretching the boundaries of what this language can do. Both the official yab IDE and my own alternative IDE were written entirely in yab.

As I've mentioned before, the possibilities of writing graphical apps for Haiku in Python, Pascal and Lua can best be described as "work in progress". But there are two other possibilities.

QT4 has been ported to Haiku, and it is possible to port QT apps to run on Haiku. This is controversial within the community. Ports that have simply been recompiled without closer attention don't look right on a Haiku system, and they ignore the features like extended attributes that make Haiku different from other operating systems. But in time, QT may well become entrenched as Haiku's second API and people may start writing QT

apps specifically for Haiku, not just porting existing ones.

More recently, OpenJDK (i.e. Java) was ported, and the Haiku community is still cautiously feeling its way into the world of Java porting and programming. Right now, Java apps look even less Haiku-like than those ported from QT. But that too may change.

And that is why you should write apps for Haiku: a small but fanatically devoted community that will snap up your program, the satisfaction of being the first to bring a new kind of app to the platform, and mostly, well, it's just more fun.

See you on the Haiku forums.

Appendix 4: A note about arrays

Level: Beginner

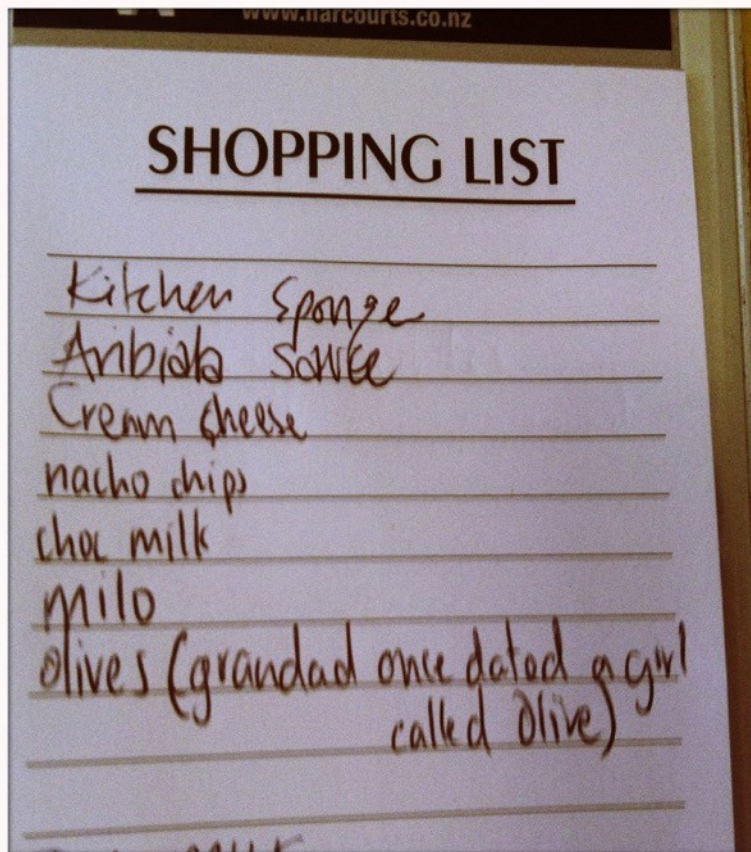
Arrays are a very useful way to store your data. The standard yab main loop uses one to store messages received, as we saw in Chapter 2.

An array has *dimensions*. The following command creates a one-dimensional array of numbers:

```
dim a(10)
```

That would be like a long list of numbers, one after the other. A one-dimensional string array would be like a shopping list:

```
dim a$(10)
```



An array can also be two-dimensional, like a spreadsheet:

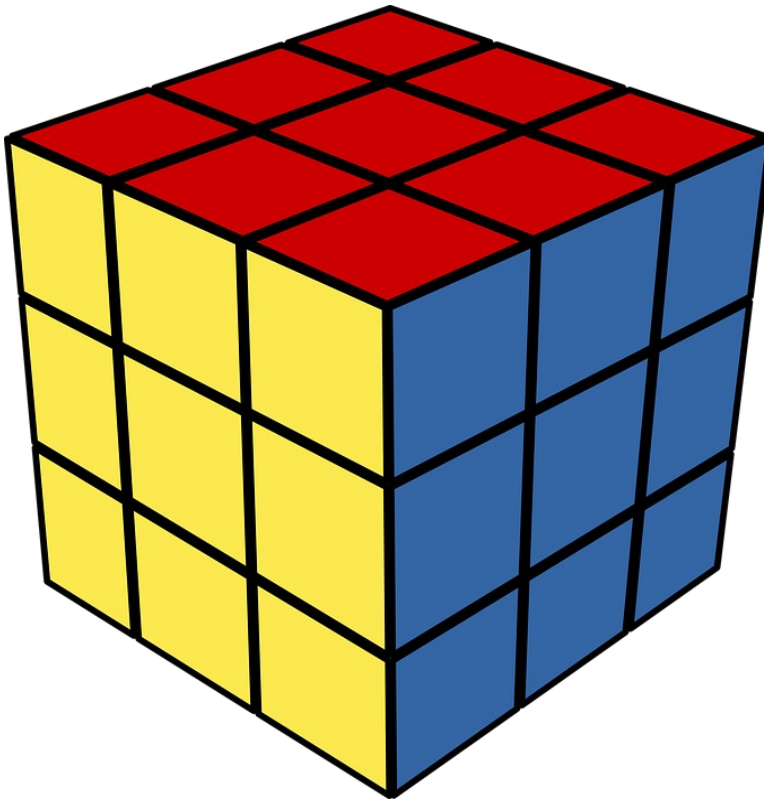
```
dim a(10,10)
```

| | A | B | C | D | |
|----|--------------------|------------------------|-----------------|---------------|--|
| 1 | Date | Income | Expenses | Profit | |
| 2 | 2005-12-17 | 235 € | 128 € | 107 € | |
| 3 | 2005-12-18 | 311 € | 124 € | 187 € | |
| 4 | 2005-12-19 | 457 € | 466 € | -9 € | |
| 5 | 2005-12-20 | 232 € | 132 € | 100 € | |
| 6 | 2005-12-21 | 122 € | 134 € | -12 € | |
| 7 | 2005-12-22 | 128 € | 223 € | -95 € | |
| 8 | 2005-12-23 | 432 € | 218 € | 214 € | |
| 9 | 2005-12-24 | 256 € | 121 € | 135 € | |
| 10 | | 2.173 € | 1.546 € | 627 € | |
| 11 | | | | | |
| 12 | Avg. Profit | =AVERAGE(D2:D9) | | | |

Actually, the spreadsheet includes text, numbers and formulas in a way that the array won't, but you get the idea, I'm sure. A two-dimensional array can be a fantastic way to hold your data.

Or the array can even be three-dimensional like, I don't know, a Borg cube? Or perhaps a Rubik's Cube if it was blocks all the way inside instead of a twisting mechanism.

```
dim a(10,10,10)
```



Now what would happen if you specified a fourth dimension?

```
dim a(10,10, 10, 10)
```

Yab will happily keep creating new dimensions until you run out of memory. DO NOT DO THIS! Unless you are working in quantum physics, three dimensions are as far as you should go.

The reason for this is simple: the human brain is not set up to visualise more than three dimensions. If you can't visualise it, you can't program it. Trying to work with more than three dimensions will lead to endless debugging. Trust me, I've tried. If you really need to store four sets of data, try four one-dimensional arrays, or a pair of two-dimensional ones.

Appendix 5: Using a treebox

Level: Intermediate

by Jim Saxton

Originally published on <http://yab.orgfree.com/forum/> - 8 March 2017, reprinted with permission

Treeboxes are the yab version of the Outline List View. They allow the user to select an item like a listbox while allowing for sub items for each listed item. This takes some special handling to set-up.

First, add the treebox :

```
TREEBOX 0,0 to 300,400, "TB", 1, View$
```

Next add all the top level items in the order to be shown:

```
TREEBOX ADD "TB", "First Item"
TREEBOX ADD "TB", "Second Item"
TREEBOX ADD "TB", "Third Item"
```

Now add the sub-items grouped together and in the reverse order:

```
TREEBOX ADD "TB", "First Item", "Sub item2",0
TREEBOX ADD "TB", "First Item", "Sub item1",0
TREEBOX ADD "TB", "Sub item1", "Detail2", 0
TREEBOX ADD "TB", "Sub item1", "Detail1", 0
TREEBOX ADD "TB", "Second Item", "Sub item2,2",0
TREEBOX ADD "TB", "Second Item", "Sub item2,1",0
```

Haiku has issues with the outlinelist view therefore yab has the same issue. If one collapsed a menu and later expands the menu and attempts to make a selection, the wrong item is selected. The following is a work-around that will always make the proper selection, although it adds a minor visual glitch.

In the program loop do the following:

```
if left$(msg$(everyCommand),11)="TB:_Select:" then
tmp=treebox get "TB"
TREEBOX SELECT "TB", tmp
// now process the selection
endif
```

The magic here is:

```
TREEBOX SELECT "TB", tmp
```

this selects the item, but the loop will come around soon enough to re-select the sub-item that was really intended. otherwise outlinelistview will continue to return the wrong

message and the user will see his selection not take effect.

EXAMPLE PROGRAM

```
#!/yab
```

```
// set DEBUG = 1 to print out all messages on the console
DEBUG = 1
dim tbitem$(10)
OpenWindow()
for x=1 to treebox count "TB"
tbitem$(x) = "This should be \"\"+(treebox get$ "TB",x)+".\"\"": print tbitem$(x)
next

// Main Message Loop
dim msg$(1)
while(not leavingLoop)
  nCommands = token(message$, msg$(), "|")
  for everyCommand = 1 to nCommands
    if(DEBUG and msg$(everyCommand)<>"") print msg$(everyCommand)
    switch(msg$(everyCommand))
      case "_QuitRequested"
      case "MainWindow:_QuitRequested"
        leavingLoop = true
        break
      default

      end switch
      if left$(msg$(everyCommand),11)="TB:_Select:" then
        tmp=treebox get "TB"
        TREEBOX SELECT "TB", tmp
        TEXTEDIT CLEAR "TE"
        TEXTEDIT ADD "TE", "Message: "+ msg$(everyCommand) +"\n\n"+tbitem$
(tmp)
        endif
      next everyCommand
    wend
  CloseWindow()
end

// Setup the main window here
sub OpenWindow()
  window open 100,100 to 600,500, "MainWindow", "Main Window"
  Window set "MainWindow","flags", "Not-Zoomable Not-Resizable"
```

```

SPLITVIEW 0,0 TO 500,400, "SplitView", 1, 1, "MainWindow"
SPLITVIEW SET "SplitView", "Divider", 100
SPLITVIEW SET "SplitView", "MinimumSizes", 100, 200
TEXTEDIT 0,0 TO 500,400, "TE", 0, "SplitView2"
TREEBOX 0,0 to 300,400, "TB", 1, "SplitView1"
TREEBOX ADD "TB", "First Item"
TREEBOX ADD "TB", "Second Item"
TREEBOX ADD "TB", "Third Item"
TREEBOX ADD "TB", "First Item", "Sub item2",0
TREEBOX ADD "TB", "First Item", "Sub item1",0
TREEBOX ADD "TB", "Sub item1", "Detail2", 0
TREEBOX ADD "TB", "Sub item1", "Detail1", 0
TREEBOX ADD "TB", "Second Item", "Sub item2,2",0
TREEBOX ADD "TB", "Second Item", "Sub item2,1",0
end sub

//close down the main window
sub CloseWindow()
    window close "MainWindow"
    return
end sub

```

Appendix 6: Full-screen programs

By Michel Clasquin-Johnson

Based on a post from 2017 on <http://yab.orgfree.com/forum/> and reprinted with permission (from myself)

Level: Beginner

yab does not have a built-in facility to write apps that run in a formal full-screen mode. But we can fake it! Writing such a program is not very difficult. It boils down to the following steps:

- Find out the dimensions of the screen using the PEEK command
- Create a window from 0, 0 to those dimensions, in other words, cover the entire screen with a window.

- Remove all the window decor with WINDOW SET
- Continue writing the rest of the program.

That's all there is to it, really! The following little demo shows how to do this.

```
#!yab
doc Fullscreen demo
OpenWindow()

// Main Message Loop
dim msg$(1)
while(not leavingLoop)
//This is a dummy loop just to stop the demo from closing
wend
end

// Setup the main window here
sub OpenWindow()
local xw, yh
xw = peek("desktopwidth")
yh = peek ("desktopheight")
    window open 0,0 to xw,yh, "MainWindow", "Main Window"
    window set "MainWindow", "look", "no-border"
end sub
```

In an actual program, though, I would do one thing differently. Those values for *hw* and *yh* - I am going to need them all over the place once I start placing widgets on this window. So just this once, I would consider making them global rather than local variables. Global variables are not intrinsically evil: if you know what you are doing, a good mixture of global and local variables is perfectly acceptable.

This is of course just a regular application that happens to take up the entire screen. Other

apps can hide behind it or come up in front of it. A true full-screen mode is provided by the operating system and prevents that from happening.

There is a way to specify that a yab app should always remain on top (see the documentation for WINDOW SET), but that is normally a bad idea when you are working full-screen. You would prevent your user from getting to Tracker or the Deskbar, for one thing. If you are going to do that, make sure to provide an obvious way to minimize or close your app. Yes, you will need to do it - you can't just leave it to the window tab, because we removed it!

But what if you want your program to toggle between a window and full screen? WINDOW SET can help you there too: it has a range of options for moving and resizing your window.

Glossary of yab commands

This quick reference to all the commands and functions supported by yab under Haiku uses the convention that commands in lowercase form part of the command-line yabasic substrate, while those in uppercase are part of the GUI-related enhancements introduced by yab. A program that only uses the commands listed here in lowercase should run seamlessly in yabasic under Windows or Linux. I have removed the commands relating to the obsolete yabasic graphic-window, and the problematic (in my experience) sound-related commands. If your program needs beeps and boops, find a CLI utility to do that and access it with SYSTEM(), or if you are up to it, remote-control a GUI app with SYSTEM("hey ..."). Good luck.

Almost all these commands take further parameters - please read the documentation that comes with yab for these, either in the yab IDE or in the documents */boot/system/apps/yab-IDE/Documentation/yabasic.html* and */boot/system/apps/yab-IDE/Documentation/yab-Commands*. The commands and functions listed here are the ones you should regard as your toolkit.

- `;` - suppress the implicit newline after a print-statement
- `:` - separate commands from each other

- ****@**** - synonymous to at
- ****** or **^** - raise its first argument to the power of its second
- **//** - starts a comment
- **#** - either a comment or a marker for a file-number
- **abs()** - returns the absolute value of its numeric argument
- **acos()** - returns the arcus cosine of its numeric argument
- **ALERT** - Opens an alert window
- **and** - logical and, used in conditions
- **and()** - the bitwise arithmetic and
- **arraydim()** - returns the dimension of the array, which is passed as an array reference
- **arraysize()** - returns the size of a dimension of an array
- **asc()** - accepts a string and returns the position of its first character within the ascii charset
- **asin()** - returns the arcus sine of its numeric argument
- **at()** - can be used in the print-command to place the output at a specified position
- **atan()** - returns the arcus tangens of its numeric argument
- **ATTRIBUTE CLEAR** - Delete an attribute
- **ATTRIBUTE GET** - Get the number value of an attribute
- **ATTRIBUTE GETS** - Get the string value of an attribute
- **ATTRIBUTE SET** - Set an attribute
- **bin\$()** - converts a number into a sequence of binary digits
- **bind()** - Binds a yabasic-program and the yabasic-interpreter together into a standalone program.
- **BITMAP** - Creates a new bitmap in memory, you can draw on it
- **BITMAP COLOR** - Returns the color of a pixel
- **BITMAP GET** - Copies specified area of a bitmap to a target_bitmap.
- **BITMAP GET** - Copies the icon, shown in Tracker, of a file onto a bitmap.
- **BITMAP GET** - delivers height and width of a bitmap
- **BITMAP IMAGE** - Load image file into a new Bitmap
- **BITMAP REMOVE** - Removes Bitmap\$ from memory
- **BITMAP SAVE** - Saves a bitmap
- **BOXVIEW** - Adds a box
- **BOXVIEW SET** - Change box decoration
- **break** - breaks out of a switch statement or a loop
- **BUTTON** - Sets a button
- **BUTTON IMAGE** - Create an image button
- **CALENDAR** - Open a calendar widget
- **CALENDAR GETS** - Get the current date selected in a calendar.

- **CALENDAR SET** - Set the date according to the current format
- **CANVAS** - Create a canvas
- **case** - mark the different cases within a switch-statement
- **CHECKBOX** - Display a checkbox
- **CHECKBOX IMAGE** - Create an image checkbox
- **CHECKBOX SET** - (De-)Activate the check box
- **chr\$()** - accepts a number and returns the character at this position within the ascii charset
- **clear screen** - erases the text window
- **CLIPBOARD COPY** - Copy text to the system clipboard.
- **CLIPBOARD PASTE\$** - Paste ASCII text from the system clipboard
- **close** - close a file, which has been opened before
- **color** - print with color
- **COLORCONTROL** - Create a color control widget
- **COLORCONTROL SET** - Set the color control to the color r,g,b.
- **colour** - see color
- **COLUMNBOX** - open a columnbox
- **COLUMNBOX ADD** - add a column
- **COLUMNBOX CLEAR** - Removes all entries of ColumnBox\$.
- **COLUMNBOX COLOR** - change colour of columnbox
- **COLUMNBOX COLUMN** - adjust columns
- **COLUMNBOX COUNT** - Returns the number of entries
- **COLUMNBOX GET\$** - Returns a specific entry
- **COLUMNBOX REMOVE** - Removes the entries in Row
- **COLUMNBOX SELECT** - Selects Row.
- **compile** - compile a string with yabasic-code on the fly
- **continue** - start the next iteration of a for-, do-, repeat- or while-loop
- **cos()** - return the cosine of its single argument
- **date\$** - returns a string with various components of the current date
- **dec()** - convert a base 2 or base 16 number into decimal form
- **default** - mark the default-branch within a switch-statement
- **dim** - create an array prior to its first use
- **do** - start a (conditionless) do-loop
- **doc** - special comment, which might be retrieved by the program itself
- **docu\$** - special array, containing the contents of all docu-statement within the program
- **dot** - draw a dot in the graphic-window
- **DRAW BITMAP** - Draw a bitmap on a view, another bitmap or a canvas.
- **DRAW CIRCLE**

- **DRAW CURVE**
- **DRAW DOT**
- **DRAW ELLIPSE**
- **DRAW FLUSH** - Deletes all former drawing commands from the drawing queue
- **DRAW GET** - Return the width/height of a string in the current font
- **DRAW GET\$** - Returns a list of all installed fonts and styles.
- **DRAW IMAGE** - Draws and scales an image file
- **DRAW LINE**
- **DRAW RECT** - Draws a rectangle
- **DRAW SET** - Fill Or Stroke
- **DRAW SET** - set a colour
- **DRAW SET** - Set the drawing font
- **DRAW SET** - Sets the Alpha-Channel for the transparency of the drawing colors.
- **DRAW TEXT** - Draws text
- **DROPBOX** - Add an empty dropbox
- **DROPBOX ADD** - Add Item to the dropbox.
- **DROPBOX CLEAR** - Clear the drop box from all items.
- **DROPBOX COUNT** - Count the number of items.
- **DROPBOX GET\$** - Get the item at a position.
- **DROPBOX REMOVE** - Removes an item
- **DROPBOX SELECT** - Select the item at a position
- **else** - mark an alternative within an if-statement
- **elsif** - starts an alternate condition within an if-statement
- **end** - terminate your program
- **end sub** - ends a subroutine definition
- **endif** - ends an if-statement
- **eof** - check, if an open file contains data
- **eor()** - compute the bitwise exclusive or of its two arguments
- **error** - raise an error and terminate your program
- **euler** - another name for the constant 2.71828182864
- **execute()** - execute a user defined subroutine, which must return a number
- **execute\$()** - execute a user defined subroutine, which must return a **string**
- **exit** - terminate your program
- **exp()** - compute the exponential function of its single argument
- **export** - mark a function as globally visible
- **false** - a constant with the value of 0
- **fi** - another name for endif
- **FILEPANEL** - Opens a filepanel

- **for** - starts a for-loop
- **frac()** - return the fractional part of its numeric argument
- **getscreen\$()** - returns a string representing a rectangular section of the text terminal
- **glob()** - check if a string matches a simple pattern
- **gosub** - continue execution at another point within your program (and return later)
- **goto** - continue execution at another point within your program (and never come back)
- **hex\$()** - convert a number into hexadecimal
- **if** - evaluate a condition and execute statements or not, depending on the result
- **import** - import a library
- **inkey\$** - wait, until a key is pressed
- **input** - read input from the user (or from a file) and assign it to a variable
- **instr()** - searches its second argument within the first; returns its position if found
- **int()** - return the integer part of its single numeric argument
- **ISMOUSEIN()** - sees if the mouse cursor is in a view
- ****KEYBOARD MESSAGE**** - ~~Works like INKEY~~ on the command line (well, nearly; it does not wait for input as inkey\$ does).
- **label** - mark a specific location within your program for goto, gosub or restore
- **LAUNCH** - Launch a program or file
- **LAYOUT** - Set the layout for all views on the window
- **left\$()** - return (or change) left end of a string
- **len()** - return the length of a string
- **line** - draw a line
- **line input** - read in a whole line of text and assign it to a variable
- **LISTBOX** - Adds an empty listbox
- **LISTBOX ADD** - Add an item to the listbox
- **LISTBOX CLEAR** - removes all entries
- **LISTBOX COUNT** - Count the number of entries in the list box
- ****LISTBOX GET**** - ~~Return the Entry~~ of Row in ListBox\$.
- **LISTBOX REMOVE** - Remove the item at a position
- **LISTBOX REMOVE** - Removes an item from the Listbox.
- **LISTBOX SELECT** - Select the item at a position.
- **LISTBOX SORT** - Sort the entries of a ListBox.
- **local** - mark a variable as local to a subroutine
- **LOCALIZE** - Initialize automatic translation and use the locale mimetype.
- **LOCALIZE** - Turn automatic translation on again.
- **LOCALIZE STOP** - Turn automatic translation off.
- **log()** - compute the natural logarithm
- **logical or** - logical or, used in conditions

- **loop** - marks the end of an infinite loop
- **lower\$()** - convert a string to lower case
- **ltrim\$()** - trim spaces at the left end of a string
- **max()** - return the larger of its two arguments
- **MENU** - Add a menu to the menubar.
- **MENU SET** - Enable/Disable or mark/unmark or remove a menu item.
- **MENU SET** - Put the menu in radio mode, so up to one item is marked.
- **MENU TRANSLATE\$** - translate menu commands.
- **MESSAGE SEND** - Send a string to another yab application
- **MESSAGES\$** - Collects the messages generated by the GUI elements.
- **mid\$()** - return (or change) characters from within a string
- **min()** - return the smaller of its two arguments
- **mod()** - compute the remainder of a division
- **MOUSE MESSAGES\$** - Returns the state of the mouse
- **MOUSE SET** - Hide or show the mouse cursor for this application.
- **mouseb** - extract the state of the mouse buttons from a string returned by inkey\$
- **mousemod** - return the state of the modifier keys during a mouse click
- **next** - mark the end of a for loop
- **not** - negate an expression; can be written as !
- **numparams** - return the number of parameters, that have been passed to a subroutine
- **on gosub** - jump to one of multiple gosub-targets
- **on goto** - jump to one of many goto-targets
- **on interrupt** - change reaction on keyboard interrupts
- **open** - open a file
- **open printer** - open printer for printing graphics
- **open window** - open a graphic window
- **OPTION COLOR** - Set the background colour of any view
- **OPTION SET** - Set/remove the focus of a view\$
- **OPTION SET** - Automatically resize a view
- **OPTION SET** - Enable/disable a control
- **OPTION SET** - Give a Control a new label.
- **OPTION SET** - Set a control invisible or visible
- **or()** - arithmetic or, used for bit-operations
- **pause** - pause, sleep, wait for the specified number of seconds
- **peek** - retrieve various internal informations
- **peek\$** - retrieve various internal string-informations
- **pi** - a constant with the value 3.14159

- **poke** - change selected internals of yabasic
- **POPUPMENU** - Pop up a menu at a position
- **print** - Write to terminal or file
- **PRINTER** - Load the printer settings from SetupFile\$ and print a view
- **PRINTER SETUP** - Setup the printing environment and store it
- **putbit** - draw a rectangle of pixels into the graphic window
- **putscreen** - draw a rectangle of characters into the text terminal
- **RADIOBUTTON** - Add a radio button
- **RADIOBUTTON SET** - (De-)Activate a radio button
- **ran()** - return a random number
- **read** - read data from data-statements
- **redim** - create an array prior to its first use. A synonym for dim
- **rem** - start a comment
- **repeat** - start a repeat-loop
- **restore** - reposition the data-pointer
- **return** - return from a subroutine or a gosub
- **reverse** - print reverse (background and foreground colours exchanged)
- **right\$()** - return (or change) the right end of a string
- **rinstr()** - find the rightmost occurrence of one string within the other
- **rtrim\$()** - trim spaces at the right end of a string
- **screen** - as clear screen clears the text window
- **SCREENSHOT** - Takes a screenshot and copies the specified region of the desktop onto a bitmap
- **SCROLLBAR** - Make View scrollable.
- **SCROLLBAR GET** - Get the current position of the scrollbars.
- **SCROLLBAR SET** - adjust scrollbar properties
- **seek()** - change the position within an open file
- **SHORTCUT** - Set a key shortcut
- **sig()** - return the sign of its argument
- **sin()** - return the sine of its single argument
- **sleep** - pause, sleep, wait for the specified number of seconds
- **SLIDER** - Add a slider with a minimum and a maximum value.
- **SLIDER COLOR** - Set slider colours
- **SLIDER GET** - Get the currently selected value of the slider.
- **SLIDER LABEL** - Set start and end limit labels
- **SLIDER SET** - Set the slider hashmarks.
- **SLIDER SET** - Set the slider to a value.
- **SPINCONTROL** - Set a spin control

- **SPINCONTROL GET** - Get the current spin control value.
- **SPINCONTROL SET** - Set the spin control to a value.
- **split()** - split a string into many strings
- **SPLITVIEW** - Set up a new view and split it into two new views
- **SPLITVIEW GET SplitView\$** - Get the current position of the divider.
- **SPLITVIEW SET** - Set the minimum sizes of the left (or top) view and the right (or bottom) view.
- **SPLITVIEW SET** - Set the position of the divider, default is the half of the total split view.
- **sqr()** - compute the square of its argument
- **sqrt()** - compute the square root of its argument
- **STACKVIEW** - Set up a stack of views where only one is always visible.
- **STACKVIEW GET** - Get the current number of the visible view.
- **STACKVIEW SET** - Set a view as the visible view.
- **static** - preserves the value of a variable between calls to a subroutine
- **STATUSBAR** - Creates a status bar with ID\$ and label(s) on View\$.
- **STATUSBAR SET** - set properties of the status bar
- **step** - specifies the increment step in a for-loop
- **str\$()** - convert a number into a string
- **sub** - declare a user defined subroutine
- **SUBMENU** - This is the same as MENU, it just adds a submenu instead.
- **SUBMENU SET** - Enable/Disable or mark/remove a submenu item.
- **SUBMENU SET** - Put the submenu in radio mode, so up to one item is marked.
- **switch** - select one of many alternatives depending on a value
- **system()** - hand a statement over to your operating system and return its exit code
- **system\$()** - hand a statement over to your operating system and return its output
- **TABVIEW** - Create a tabbed view
- **TABVIEW ADD** - Create a new tab
- **TABVIEW GET** - Get the current opened tab.
- **TABVIEW SET** - Open a tab
- **tan()** - return the tangens of its argument
- **tell** - get the current position within an open file
- **TEXT** - Displays Text. This cannot be flushed like DRAW TEXT and is meant for permanent labels. Furthermore you can set the alignment for this command with the TEXT SET command.
- **TEXT SET** - Set the alignment of Text.
- **TEXTCONTROL** - Opens a textcontrol
- **TEXTCONTROL CLEAR** - Delete the text of the text control.
- **TEXTCONTROL GET\$** - Get the entry of a text control

- **TEXTCONTROL SET** - Set the text control's options.
- **TEXTEDIT** - Opens a text editor
- **TEXTEDIT ADD** - Insert text in a textedit
- **TEXTEDIT CLEAR** - Clears text from a textedit
- **TEXTEDIT COLOR** - change textedit colours
- **TEXTEDIT GET** - Get the current line number in a textedit or other textedit properties. Also used to find text
- **TEXTEDIT GET\$** - get the text of a textedit, the current line, or the current selection
- **TEXTEDIT SET** - Applies an option to a textedit
- **TEXTURL** - Create a texturl widget
- **TEXTURL COLOR** - Set the colours for the URL
- **then** - tell the long from the short form of the if-statement
- **THREAD GET** - Get thread information
- **THREAD REMOVE** - kill a thread. You can crash your system with this command! If you don't know what this command is meant for, then don't use it!
- **time\$** - return a string containing the current time
- **to** - this keyword appears as part of other statements
- **token()** - split a string into multiple strings
- **TOOLTIP** - set a tooltip
- **TRANSLATE\$** - get translations
- **TREEBOX** - Adds a tree box.
- **TREEBOX ADD** - Add item to the treebox.
- **TREEBOX CLEAR** - Clear the tree.
- **TREEBOX COLLAPSE** - Collapses items in a treebox.
- **TREEBOX COUNT** - Returns the number of entries in a treeBox
- **TREEBOX EXPAND** - Expands item in a treebox.
- **TREEBOX GET** - Get the current treebox position.
- **TREEBOX GET\$** - Returns the Item at a position in a treebox.
- **TREEBOX REMOVE** - Removes a list item from the tree. Note: this also removes all subitems!
- **TREEBOX REMOVE** - Removes the entry at a position in a treebox.
- **TREEBOX SELECT** - Selects the item at a position in a treebox.
- **TREEBOX SORT** - Sorts the entries of TreeBox\$ alphabetically.
- **trim\$()** - remove leading and trailing spaces from its argument
- **true** - a constant with the value of 1
- **until** - end a repeat-loop
- **upper\$()** - convert a string to upper case
- **using** - Specify the format for printing a number

- **val()** - converts a string to a number
- **VIEW** - Adds a view
- **VIEW DROPZONE** - Define view as a drop zone that accepts dropped files.
- **VIEW GET** - Returns the requested property of a view or if used with Exists/Focused returns 1 if so and 0 if not.
- **VIEW REMOVE** - Remove view. The window view cannot be removed. Warning: Currently clears all internal information about menus and drop boxes. It is only safe to use it, when you don't have any menus or drop boxes on views that will be removed. Never remove menus with shortcuts, otherwise yab will crash when the shortcut key is pressed!
- **wait** - pause, sleep, wait for the specified number of seconds
- **wend** - end a while-loop
- **while** - start a while-loop
- **WINDOW CLOSE** - Closes window Warning: this might destabilize your program if you try
- **WINDOW COUNT** - Returns the number of open windows
- **WINDOW GET** - Returns the requested property or if used with Exists returns 1 if found and 0 if not.
- **WINDOW OPEN** - Open a window
- **WINDOW SET** - Set window properties

xor() - compute the exclusive or

Glossary of handy PEEKs

The *peek* and *peek\$* commands are an incredibly useful way to get information about the system on which your application is running. For example, if you can use PEEK to determine that your app is running on a 1024 x 786 screen, you can stop your program from creating a window bigger than that.

This quick reference to all the peeks supported by yab under Haiku uses the convention that commands in lowercase form part of the command-line yabasic substrate, while

those in uppercase are part of the GUI-related enhancements introduced by yab.

Please read the documentation that comes with yab for these, either in the yab IDE or in the documents */boot/system/apps/yab-IDE/Documentation/yabasic.html* and */boot/system/apps/yab-IDE/Documentation/yab-Commands*. But note that the yabasic documentation file, in particular, has not been purged of the old "graphic-window" commands.

- **peek("argument")** - Return the number of arguments, that have been passed to yab at invocation time. E.g. if yab has been called like this: yab foo.yab bar baz, then peek("argument") will return 2. This is because foo.yab is treated as the name of the program to run, whereas bar and baz are considered arguments to the program, which are passed on the command line. The function peek("argument") can be written as peek("arguments") too. You will want to check out the corresponding function peek\$("argument") to actually retrieve the arguments. Note, that each call to peek\$("argument") reduces the number returned by peek("argument").
- **PEEK("Deskbar-x")** - Returns the position of the left side of the Deskbar
- **PEEK("Deskbar-y")** - Returns the position of the top side of the Deskbar
- **PEEK("Deskbarexpanded")** - Returns true if the Deskbar is expanded (only possible in position 1 and 3) and false if not.
- **PEEK("DeskbarHeight")** - Returns the width of the Deskbar
- **PEEK("Deskbarposition")** - Returns the position of the Deskbar as follows (clockwise): 1 = top-left 2 = top 3 = top-right 4 = bottom-right 5 = bottom 6 = bottom-left
- **PEEK("DeskbarWidth")** - Returns the height of the Deskbar
- **PEEK("DesktopHeight")** - Returns the current Y-resolution of the current desktop
- **PEEK("DesktopWidth")** - Returns the current X-resolution of the current desktop
- **peek("error")** - Return a number specifying the nature of the last error in an open-or seek-statement. Normally an error within an open-statement immediately terminates your program with an appropriate error-message, so there is no chance and no need to learn more about the nature of the error. However, if you use open as a condition (e.g. if (open(#1,"foo")) &€!) the outcome (success or failure) of the open-operation will determine, if the condition evaluates to true or false. If now such an operation fails, your program will not be terminated and you might want to learn the reason for failure. This reason will be returned by peek("error") (as a number) or by peek\$("error") (as a string)
- **peek("isbound")** - Return true, if the executing yab-program is part of a standalone program
- **PEEK("menuheight")** - Returns the height of the menu (which is related to the user's font settings). This returns the height only when any window already has a menu, otherwise it returns -1.
- **peek("screenheight")** - Return the height in characters of the window, wherein yab runs. If you have not called clear screen yet, this peek will return 0, regardless of the

size of your terminal.

- **peek("screenwidth")** - Return the width in characters of the window, wherein yab runs. If you have not called clear screen yet, this peek will return 0, regardless of the size of your terminal.
- **PEEK("scrollbarwidth")** - Returns the width of the scrollbars. Different to "menuheight", it returns the correct width even if no scrollbars are used.
- **PEEK("tabheight")** - Returns the height of the tabfield.
- **peek("version")** - Return the version number of yab (e.g. 2.72).
- **peek(a), peek(#a)** - Read a single character from the file a (which must be open of course).
- ****peek\$("argument")**** - Return one of the arguments, that have been passed to yab at invocation time (the next call will return the the second argument, and so on). E.g. if yab has been called like this: yab foo.yab bar baz, then the first call to peek\$("argument") will return bar. This is because foo.yab is treated as the name of the program to run, whereas bar and baz are considered arguments to this program, which are passed on the command line. The second call to peek\$("argument") will return baz. Note, that for windows-users, who tend to click on the icon (as opposed to starting yab on the command line), this peek will mostly return the empty string. Note, that peek\$("argument") can be written as peek\$("arguments").
- **PEEK\$("directory")** - Returns application directory.
- ****peek\$("env", "NAME")**** - Return the environment variable specified by NAME (which may be any string expression). Which kind of environment variables are available on your system depends, as well as their meaning, on your system; however typing env on the command line will produce a list (for Windows and Unix alike). Note, that peek\$("env",...) can be written as peek\$("environment",...) too.
- **peek\$("error")** - Return a string describing the nature of the last error in an open- or seek-statement. See the corresponding peek("error") for a detailed description.
- **peek\$("infolevel")** - Returns either "debug", "note", "warning", "error" or "fatal", depending on the current infolevel. This value can be specified with an option on the command line or changed during the execution of the program with the corresponding poke; however, normally only the author of yab (me!) would want to change this from its default value "warning"
- **peek\$("library")** - Return the name of the library, this statement is contained in.
- **peek\$("os")** - This peek returns the name of the operating system, where your program executes.
- **PEEK\$("refsreceived")** - Returns TrackerItem which you used 'Open with...' your application on.