Basic Programming

with the Casio ClassPad

First Edition

written by **Marty Schmude** 2008

Table of Contents

TABLE OF CONTENTS 2

FROM THE AUTHOR 3

LESSON 1 - STARTING A NEW PROGRAM 4

LESSON 2 – DISPLAYING TEXT 6

LESSON 2 (EXTENSION) – A MESSAGE TO YOUR USER 10

LESSON 3 – RENAMING A PROGRAM 11

LESSON 4 – PROMPTING THE USER FOR AN INPUT 12

LESSON 5 – USING THE INPUT DATA 15

LESSON 6 – MEET THE `LBL` AND `GOTO` COMMANDS 17

LESSON 7 – MEET THE `INT` FUNCTION 19

LESSON 8 – MEET THE `IF` STATEMENT 20

LESSON 9 – MEET THE `RAND` FUNCTION 24

LESSON 10 – START THE *SUMDICE* PROGRAM 26

LESSON 11 – CREATE A GRAPH 28

CONCLUSION 30

EXTENSION – LOCKING A PROGRAM 31

Styles used in this booklet

Please take note of the formatting style and what each style means.

Italic – Represents the *name of a file*

Bold – Represents a **menu** option

`Courier font` – Represents a `command` of the ClassPad and the `code` written in a file

From the author

Learning to program the Casio® ClassPad offers you the ability to create simple but powerful programs. With a little bit of knowledge, you can write pieces of code that can make your life easier, help your students learn, and even help you better understand mathematics.

For the writing of this book, it has been assumed that you at least know the basics on using the ClassPad, such as navigating around the ClassPad's interface (like entering menus, entering and deleting text) and how to draw basic graphs, like the histogram. If you do not feel like you know these basics, there are some excellent help documents you can download for free at www.casioed.net.au.

It has also been assumed that you have **no experience** in the area of programming.

This booklet has been structured as a continual sequence of lessons, each one building on the previous one. It revolves around the creation of one program, which in the end, will hopefully be of some use. As well as the major program, you will be given other programs to write, if you choose, that will use the knowledge you have learnt, and apply it to a different situation.

Any feedback or comments are welcomed. I hope you enjoy the learning experience.

Marty Schmude

Email: mjschmude@hotmail.com

LESSON 1 - Starting a new program

Entering the PROGRAM application

The first part of this lesson is to navigate our way into the ClassPad's PROGRAM application. To do this from the MAIN menu, simply scroll down and tap on the PROGRAM icon. You will be taken directly to the Program Loader page (seen at the bottom of the CP's screen). The Program Loader lists programs according to the folders they are saved in. The default folder is the **main** folder.

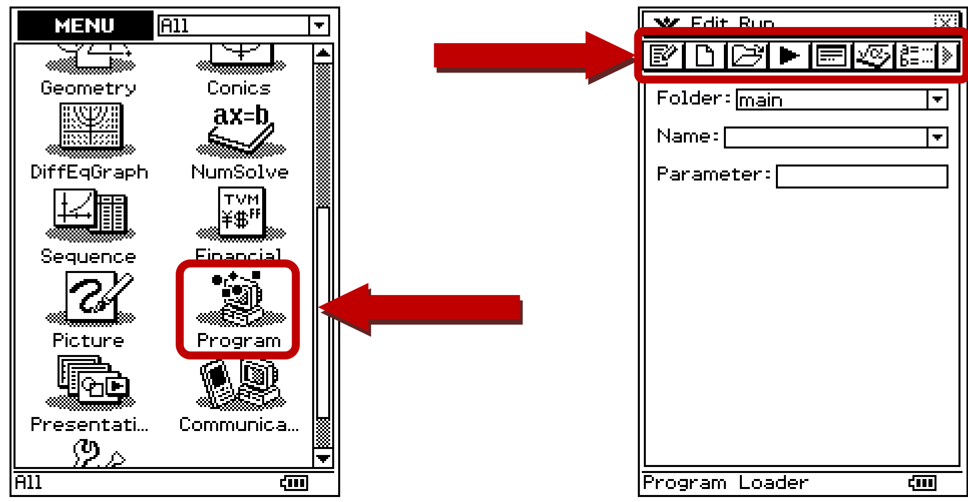
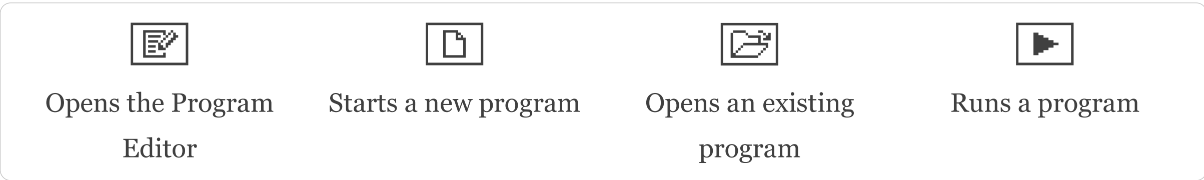


Figure 1 - MAIN menu

Figure 2 - Program Loader

Getting to know the environment

Across the top of the Program Loader screen, you will see different icons. The first four icons will be the most important to us initially. A description of them is listed below.



Opens the Program Editor	Starts a new program	Opens an existing program	Runs a program

Create and name a new program

Let's start a new program (to be saved in the **main** folder) by tapping on the icon, or pressing **Edit** ⇨ `New File`.

You will be prompted to give the ClassPad some details about your new program. Namely,

- What **type** of program is it?
- Which **folder** would you like to save the program into?
- What **name** will you give the program?

Figure 3 - Program Loader window

Marty Schmude 2008

Both the defaults for **Type** and **Folder** are just what we want, so we will leave them and simply enter in a name. For the moment, let's call our new file *hello*. The 'soft' keyboard should already be showing at the bottom of the screen, so go ahead and enter *hello*, then tap OK (figure 4).

Figure 4 - New file window

You will be immediately taken to the Program Editor (name shown at the bottom of the screen). This is the area where the actual program writing takes place (figure 5).

Saving and exiting a program

Once you are in the Program Editor, you can always save your file tapping the icon, or pressing **Edit** ⇨ `Save File`. Go ahead and save it now.

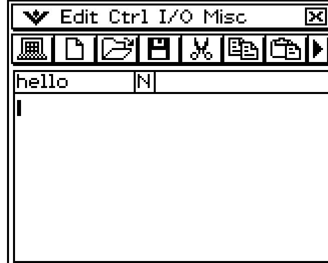


Figure 5 - Program Editor

To return to the Program Loader (where you would need to go to run your program), simply tap on the icon.

Please take note, if you try to exit the Program Editor before saving the file, a prompt will appear, asking you if you would like to save the changes (figure 6).

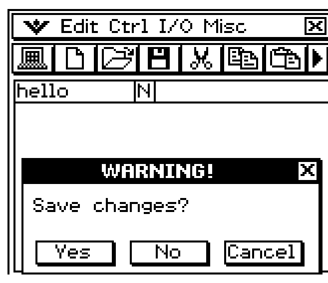


Figure 6 - prompt to save changes

Tips

- The name of a file can only be 8 characters long, including spaces.
- There are some words that are reserved solely for the ClassPad and are not allowed as file names, such as *text*.

LESSON 2 – Displaying text

The need for text

Nearly every program you will ever write will need to display text of some sort. Whether it is to give instructions to the user or information about the data on the screen, it is a necessary skill to have. The good news is that it is very simple and nice place to start. As this is the first time entering code, the instructions will be a little more detailed than they will be further on in the book.

Entering the *hello* program

To be able to write the code, you need to be in the *hello* file Program Editor. If you are not there already, please select the *hello* file from list in the Program Loader and tap on the icon (figure 7). You should then be in the Program Editor (figure 2).

Figure 1 - Program Loader

Writing our first code

To place text on screen, we simply use the `Print` command. You can print text, which is often called a 'string'. You can also show the value of variables, which we will talk about a bit later. For now, let's write some code.

Believe it or not, programming has its own traditions when it comes to learning to program, and one of them is the first piece of text you should write. The words should be 'Hello World' (http://en.wikipedia.org/wiki/Hello_world_program). It has happened for decades, and so we will follow in the steps of thousands and thousands of people who have gone before us.

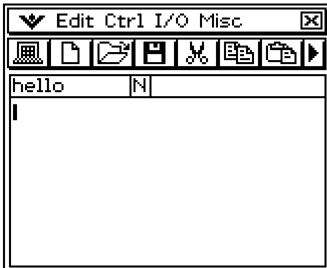


Figure 2 - Program Editor for *hello*

To make our program print the text, 'Hello World', we simply enter the line,

```
Print "Hello World"
```

Now on the ClassPad, entering commands, like `Print`, is so simple! You will notice up the very top of the screen, the name of the menus we will be using extensively. Namely, **Edit**, **Crtl**, **I/O** (which stands for Input/Output) and **Misc** (for Miscellaneous).

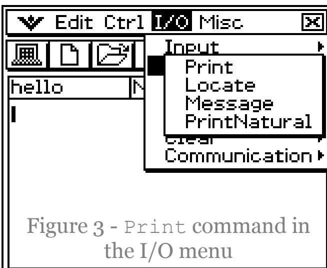


Figure 3 - `Print` command in the I/O menu

Since printing `"Hello World"` is an output, we will find the `Print` command in the **I/O** menu. So go ahead and tap on the **I/O** menu, then **Output** and lastly `Print`.

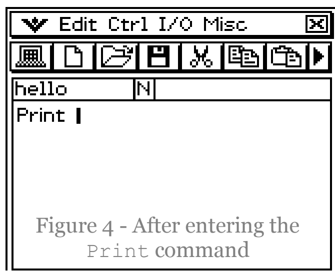


Figure 4 - After entering the `Print` command

From now on, to make things more succinct and easy to read, instructions will be written like this,

I/O ⇨ **Output** ⇨ `Print`

The next part is writing the text, "`Hello world`". You will notice that the text string is in double inverted commas. This is the way a text string needs to be entered into the ClassPad.

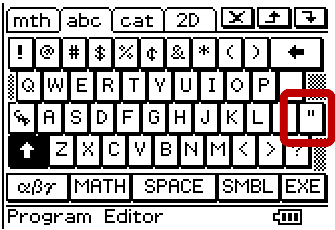


Figure 5 - Keyboard with Caps Lock on

The " symbol is found on the 'soft' keyboard. You should be already be seeing the keyboard, but if not, simply press (Keyboard) on the keypad. The " symbol is found when Shift is pressed (🠅 symbol), and " is on the right side of the keypad (figure 5).

Next just type in the words, `Hello World`, and finish with the double inverted commas again (figure 6).

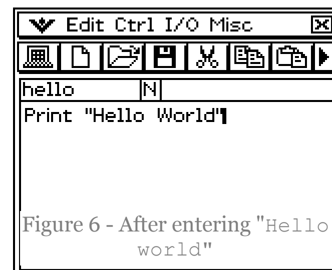


Figure 6 - After entering "`Hello world`"

Running the program

Save your file by tapping on the [💾] icon. It is now ready to run! Let's get back to the Program Loader by tapping on the [▦] icon and then run *hello* by tapping the play button, [▶]. You should see this in the Output Screen.

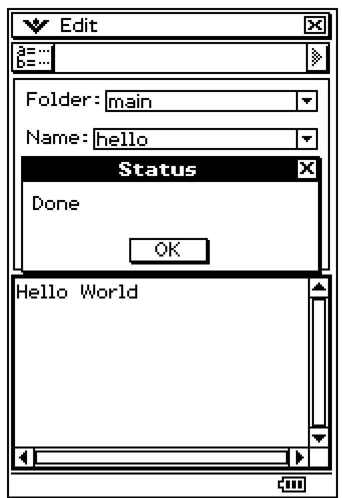


Figure 7 - after running hello

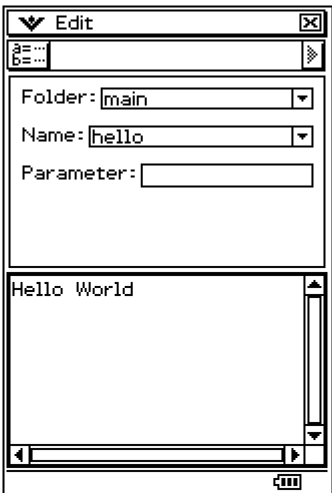


Figure 8 - split screen

You will see a Status window letting you know that program is 'Done'. Tap OK to get rid of the message.

You will also notice that the screen is split in two – the Program loader at the top and the Output Window on the bottom. The bold line around the Output Window means it is the active window (figure 8).

You could close the Output Window by simply tapping on it the in the [☒] top right corner, but we'll keep it open for the moment.

The program has done exactly what we expected. It has printed the text, "`Hello World`".

Now there is one important point to be made when writing programs. To make this point, tap in the Program Loader screen window in the top half of the screen, and run the *hello* program again (by pressing [▶]) and see what happens. You should get the result shown in figure 9.

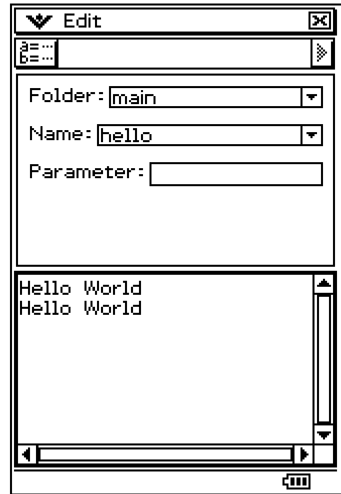


Figure 9 - need to clear the screen

You may think that our *hello* program has made a mistake and printed "Hello World" twice, but it hasn't. What it has done, is simply print the text, "Hello World", again, but the problem is that the message from the first time we ran the *hello* program is still in the output window.

So the point of the task is to realise that when you start a program, it is a good idea to write in some things at the start of the program that will ensure you have a 'clean slate' on which to work (in this case, with text).

Enter the *hello* Program Editor

Let's get back into the *hello* Program Editor. This will be the only time you will be told directly how to do this (because you'll probably get sick of hearing it ☺), but if you forget down the track, you can always come back to here.

Tap on the top half of the screen, which is the Program Loader window. You will see the box around it go bold, which means it is the active window. You will notice the menu change at the very top too. Tap on the [✎] icon to get back into the *hello* Program Editor.

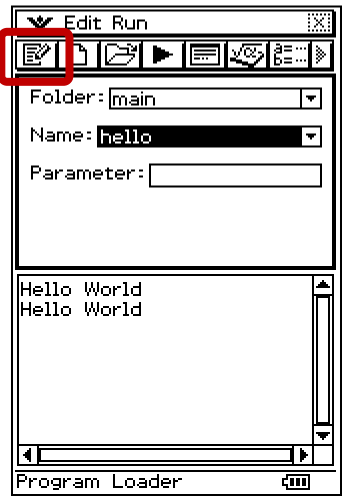


Figure 10 - duplicate copies

Clearing the text

To clear the text from the Output Window, we need to place the command, `ClrText`, in the code before we want to display the text i.e. before the `Print "Hello World"` line.

Place the cursor at the start of the `Print "Hello World"` sentence by tapping it with your stylus.

The `ClrText` command can be found here,

I/O ⇨ **Clear** ⇨ `ClrText`

Now press (EXE) to insert a line break. You should have your code looking like figure 11.

Figure 11 – code for lesson 2

That should do it! Save your file and give it a try. Remember to run it you need to go back to the Program Loader ([▦]) and press [▶].

You should be able to run it over and over again and never see duplicate "`Hello World`" s.

Quick recap

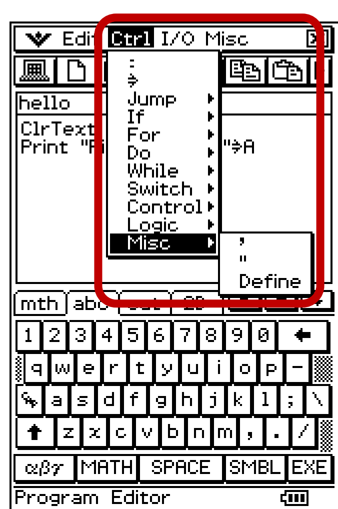
In this lesson, we've learnt two main things,

1. How to command the *hello* program to display text, and
2. How to prepare the Output Window so that any text that was previously displayed is cleared.

The key points to remember when entering text is,

- The text string needs to be encased in double inverted commas, such as,

  "`Hello World`"
- To display text, we use the `Print` command in the code.
- When a program is started, the displayed text remains in the Output Window until cleared by a `ClrText` command.
- The " symbol can also be found in the following place,



Give it a try

Here is a task for you to try on your own. There aren't any answers, but you should be able to tell if you've done it correctly yourself.

How about you create a new program file (**normal** type, in the **main** folder) called *name* and have it display your full name.

LESSON 2 (extension) – A message to your user

Please note...

This particular lesson is not going to be used in our program at this stage, but it deals with displaying text, so probably best fits into this lesson. It is a nice feature and is well worth learning.

A user can't miss it!

The last lesson displayed text to the user in the Output Window, but it is possible to create a more prominent message to the user. This message is shown in a big pop-up style box, and to continue, the user must tap on an OK button. There is another option of Cancel, which stops the program completely. During the message's presentation, a program is paused.

Some possible uses for a message box are:

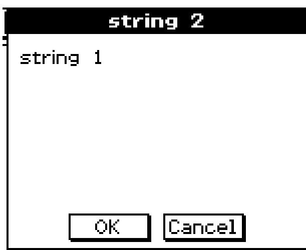
- A welcome message
- A very important message that is crucial to the user
- A copyright message

The code

The syntax for the `Message` command is,

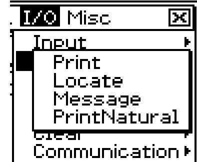
```
Message "string 1","string 2"
```

String 1 is the text you wish to place in the message box and string 2 is the title of the message box. A picture is shown below for you. String 2 is optional and does not need to be included.



The command can be found here,

I/O ⇨ **Output** ⇨ `Message`



Give it a try

Create a new program and call it *note*. Then, enter a message to your users. You may like to try it with only string 1, and then again using both string 1 and 2.

Marty Schmude 2008

LESSON 3 – Renaming a program

Changing the *hello* name

There might be occasions when you have to change the name of your program. We are going change the name of our *hello* program to *multiply* (the reason will become apparent later on).

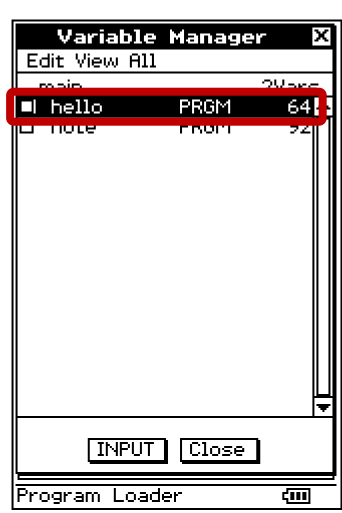
There a few taps you'll have to do to rename the *hello* program. To start with, let's get back into the Program Loader, then tap on the icon, which takes you to the **Variable Manager**.

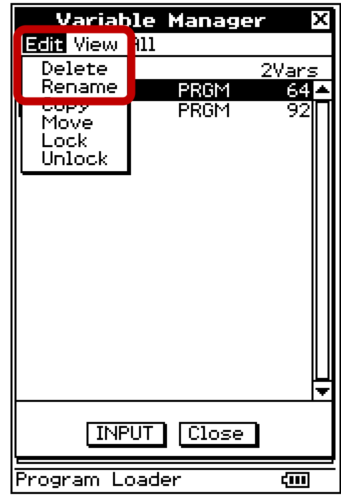
Figure 1 - entering the Variable Manager

Now, we have to enter the folder that holds our *hello* program, which is the **main** folder. Tap on this to enter the folder. You should see the *hello* program, as well as any other programs you've created so far.

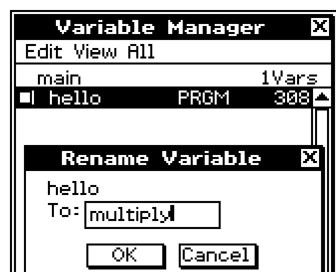
Tap on the *hello* program to select it (shown by a black box on the row), then tap,

Edit ⇨ Rename





A dialog box will appear for you to enter the new name. We will call it, *multiply*, then tap OK.



To exit the Variable Manager, simply tap Close twice.

LESSON 4 – Prompting the user for an input

Make your programs flexible

Imagine you wish to write a program that multiplies two numbers together and then displays the answer. It would be pointless to write a program that already had the two numbers in it, and simply showed you the answer. What if you wanted to multiply two different numbers? You would have to go back into the code and change those two numbers.

As an example, let's say you wrote the piece of code shown below. Just be aware that the letters, `A` and `B` are called variables, and simply serve as holders of numbers.

```
6⇨A
10⇨B
Print A×B
```

The way to read the code above is like this, *'let 6 be stored as* `A` *and 10 be stored as* `B`*, then print the answer of* `A×B`*'*. The ⇨ button is found at **Ctrl** ⇨ ⇨.

The result in the output window would be equal to 60. In fact, it would only ever equal 60! You could run this code a million times and it would never display a different answer. This is known as *hard coding* as there is no flexibility in the code to handle different values. Generally speaking, 'hard coding' is NOT a good thing.

What would be better is to ask the user to enter the two numbers they would like to multiply, and have the program dynamically handle the user's numbers – a gorgeous use of algebra. Well this lesson deals with the first part of this process – asking the user for a number.

Asking the user

To get the program to ask the user to enter a number is very easy. All you have to do is use the `Input` command and assign the entered value to a variable (a letter), just like we did above with `A` and `B`.

Let's get back into our *multiply* program. You will see the code we had before.

Our first step is to delete the line `Print "Hello World"`, as we don't need that anymore. Instead, on that same line we will enter the following,

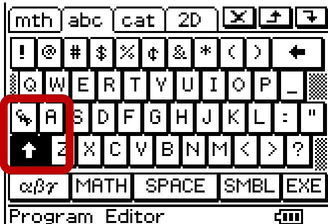
```
Input A
```

The `Input` command is found here,

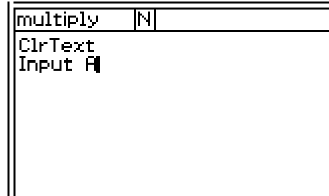
I/O ⇨ **Input** ⇨ `Input`

Marty Schmude 2008

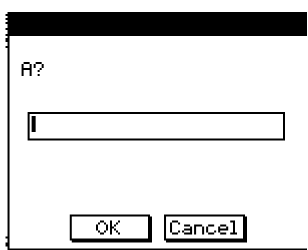
The letter A just comes from the keyboard. You will have to press ⬆ to access the capital A. This means that whatever number the user enters, it will be stored in variable A.



Your new code should look like this,



Save your file and run it. You will see a pop-up window where the user can enter a number.



You may notice that the text displayed to the user is the variable we assigned it to, which was A. Just to say it again, this means that whatever number the user enters, will be stored as variable A.

We have a problem

There is a major user-friendly problem with the program at the moment. The user probably has no idea what it is that the program is asking for. What on earth does A mean?

It would be better to have a small message to the user that gives them some idea of what we want them to enter.

Message to the user

We have to adjust the Input command by adding some extra detail. If you did LESSON 2 (extension), this will sound very familiar.

The actual input syntax is,

```
Input <variable>,”string 1”,”string 2”
```

String 1 refers to the message to the user, and string 2 refers to the title of the message box (which is optional, and we will leave blank at the moment).

So let's get back into our *multiply* Program Editor and adjust the `Input` command. Place the cursor after the variable A, and enter the extra strings so it is the same as the code shown below. You can find the comma (`,`) on the keypad.

```
--------- multiply --------
ClrText
Input A, "Please enter your
first number"
```

Again, save your file and run it. You should see the same box below. That looks much better!

Quick recap

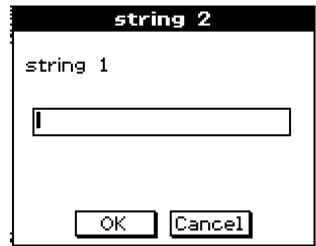
This lesson has covered two main points. Firstly, you were introduced to the basic `Input` command, which is used to get the calculator to stop and ask the user to enter a number. Secondly, we took the `Input` command a step further and included some extra text, so that the user had more of an idea of what was being asked of them.

The `Input` command always needs to have a variable assigned to it, which it uses to store the value of the user's entry.

There are some extra text strings you can include with the `Input` command. String 1 is the text in the box, and arguably the most noticeable, and string 2 is the text used in the title bar of the message box.

This line of code below produces the window below, and would store the number entered in the variable `A`.

```
Input A,"string 1","string 2"
```



Just beware that both string 1 and 2 cannot be just any length. If they are too long, there is a chance that some of the text will be cut off by the box. The box will not expand to show all of the text in the strings if they are too long. Try a few different sentences to get a feel for an acceptable length.

LESSON 5 – Using the input data

Let's use some numbers

It is no good asking for numbers from the user if you're not going to do anything with them. This lesson is going to focus on using two numbers given by the user to multiply them together, and then show the user the result.

Please note, the actual skills used in this lesson have already been covered, so the instructions will be succinct.

Ask the user for two numbers

Make sure you are in the *multiply* Program Editor.

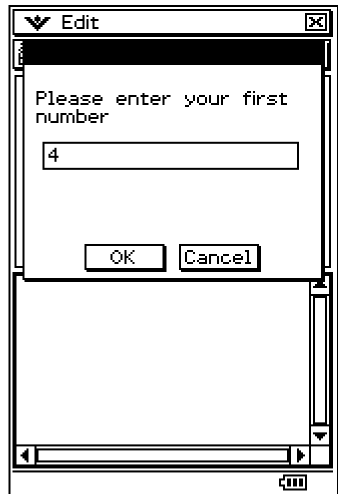
We need to ask the user to enter in a second number too, and then we will show the product of the two numbers to the user.

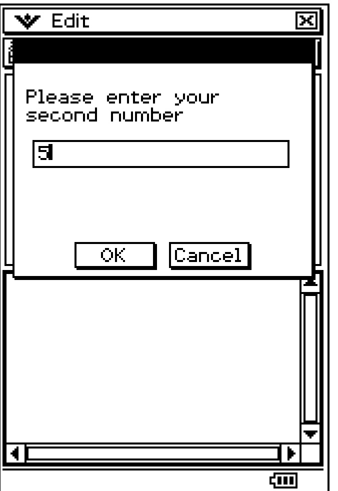
The code is very straight forward and looks like this,

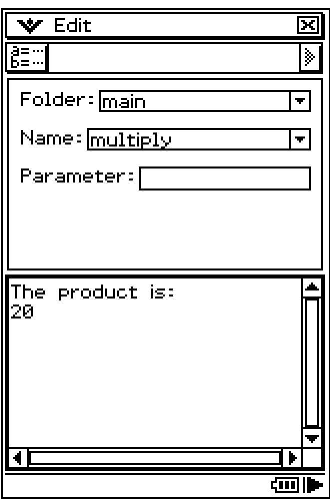
```
--------- multiply --------
ClrText
Input A, "Please enter your
first number"
Input B, "Please enter your
second number"
Print "The product is:"
Print A×B
```

Please note, there are no inverted commas around the operation, A×B. The reason will be explained later.

Save your file and give it a try. You should have something similar to what is shown below.







How it works

This lesson used the skills that we have learnt in the previous lessons, so nothing to add on that front.

What we added to the program though, was another input box for the user to enter a second number, and then we used both of those numbers to print the product.

Note how powerful the program is now. It is very flexible for what we want it to do, and that is to multiply two numbers together and show the result. When we say flexible, we mean that user can input any two numbers (decimals, negative numbers…) and it will always work. There is no hard coding with regards to the numbers. The only hard coding is the that it is always multiplication.

You will have noticed something different when we wrote the line shown below.

```
Print A×B
```

You will have noticed that the `A×B` is not in inverted commas. The reason for this is that `A` and `B` are variables, and so to differentiate them from a regular text string, they are do not have inverted commas. If we did put them in inverted commas, like this,

```
Print "A×B"
```

The result is that it simply would have printed `A×B`, instead of the product of `A×B`, which is what we want.

Give it a try

Write a similar program called *subtract*. Have the user enter two numbers to find the difference. Use the message box to direct the user to enter the larger number first.

LESSON 6 – Meet the `Lbl` and `Goto` commands

A downfall with the current program

One problem with the program as it is at the moment, is that you can only run it once and then the program stops. If you have more than one lot of numbers you wish to multiply, then you have to go back into the Program Loader and run the program again. It would be nice to have this program on a loop, where it jumps back to the start straight after it shows the results. To do this, we will use a `Lbl` (short for `label`) and a `Goto` command.

They work in pairs

The `Lbl` and `Goto` commands work in pairs. You simply give each of them the same name and place the `Lbl` where you want the program to jump to when it reaches the `Goto` command. We will use these commands to make the program jump back to the start after it has shown the result of the first pair of numbers.

Now when it was said that they must have the same name, what was meant was that you need to give the commands a unique identifier so that the ClassPad knows where to jump to. For example, you could call them `Lbl start` and `Goto start`. You can only ever have one `Lbl` with a certain name, but you can have many `Goto` commands with the same name. In other words, a program could have only one `Lbl` called `start`, but could have many `Goto` commands called `start`.

The code

We are going to place the `Lbl` at the very start of program and the `Goto` at the very end of the program.

Both the `Lbl` and `Goto` commands are found here,

Ctrl ⇨ **Jump** ⇨ `Lbl` and `Goto`

You will need to use your keyboard to enter the word, `start`, as the name for both commands.

```
--------- multiply --------
Lbl start
ClrText
Input A, "Please enter your
first number"
Input B, "Please enter your
second number"
Print "The product is:"
Print A×B
Goto start
```

Save your file and give it a try. **Just a quick warning in advance, there'll be a problem**. The next part of the lesson will be adding an extra line to fix our little problem.

It doesn't stop before jumping

Did you notice what the problem was? You might have found that after entering the second number and pressing OK, the program instantly went back to asking you for the first number again! It doesn't even show the answer to the first lot of numbers you entered.

The problem is that, after entering the second number, the program goes on to print the result of `A×B`, then instantly jumps to the top of the program, which clears the text and prompts the user for the first number again. It simply goes through the 'print the answer for the user' part far too quickly.

What we want is to 'pause' the program, in order for us to have a look at the answer of `A×B`, and then we'll tell the program to continue when **we** are ready!

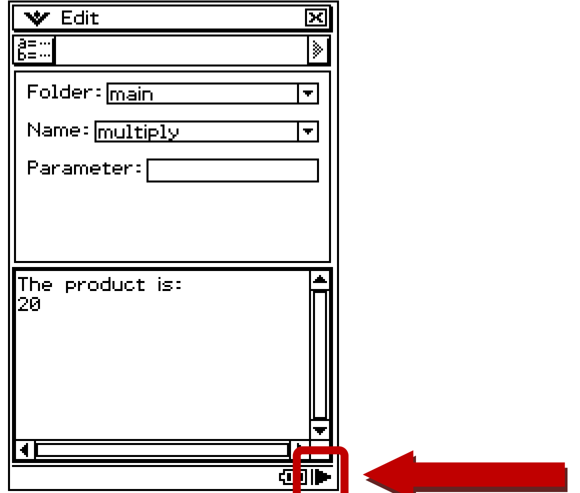
Pausing the program

We'll place a pause command in between showing the answer and the `Goto` command. Adding a `Pause` command is very simple. You will find it here, **Ctrl** ⇨ **Control** ⇨ `Pause`

```
--------- multiply --------
Lbl start
ClrText
Input A, "Please enter your
first number"
Input B, "Please enter your
second number"
Print "The product is:"
Print A×B
Pause
Goto start
```

Resume running the program

There is one really important thing to be aware of, and that is how to get the program to resume running. It is a fairly subtle way that users may not be aware of. When a program pauses, you will see in the bottom right corner of the screen, a dark arrow. **You need to tap the arrow to continue!**



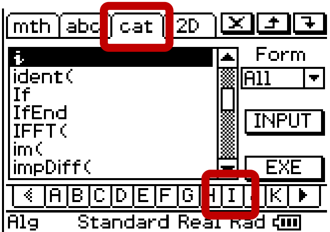
LESSON 7 – Meet the `int` function

Cutting off the decimals

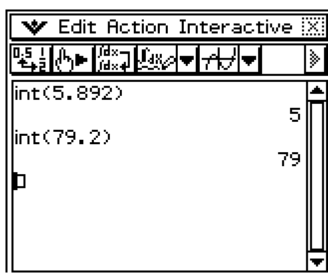
We are going to investigate the `int` function on the calculator, as we will be using it in our next lesson. The `int` function simply takes the integer (whole number) part of a number. It doesn't round the number up or down, it just simply truncates the decimal places.

To see it in action, let's have a break from the PROGRAM application for a moment, and head on into the MAIN application, by tapping of the Main icon.

The `int` function can be found in the **catalogue**, which is found on your 'soft' keyboard. Bring up your keyboard by pressing Keyboard on your keypad. Next, tap on **cat** (which stands for catalogue), and then the letter I.



Scroll down a little until you see `int(`. Tap on it and then tap on `INPUT` to insert the function. You will see it inserted into the **Main** window. Please beware not to select a very similar function, which is `intg(`.



Enter in a number with decimals and then press `EXE`. You will see that the `int` function simply ignores the decimal value completely, and only returns the whole number part.

Give it a try

Use you calculator to find the values of these questions.

`int (1.846)`   `int (19.365)`   `int (-8.74)`   `int (0.69)`

LESSON 8 – Meet the `if` statement

Meeting a certain condition

The `if` statement is one of the most widely used tools in programming, and arguably one of the most important because of the enormous control and flexibility it offers. An `if` statement is called a conditional statement. That means if a certain condition is met, then a certain bit of code is executed. If the condition is not met, the program moves on to the next section of code.

The condition we will add to our *multiply* program is going to restrict the user to only being allowed to enter whole numbers (integers) when prompted for the first and second number. To do this, we are going to use the `int` function to test the numbers. In other words, after the user enters a number, we are going to test to see if it is a whole number. If the number is a whole number, let's call it `A`, than `int(A)=A`. But if `A` is not a whole number, than `int(A)≠A`. This is where the `if` statement is used.

Always think of the user

So what should happen if they enter a number that is not a whole number? Well, a couple of things 'have' to happen to be considered a user-friendly program.

> **Issue 1 - The user should be warned at the prompt that the number needs to be a whole number (prevention is better than cure ☺).**
>
> **Issue 2 - If the user didn't read the warning properly, they need to be told why it is incorrect.**
>
> **Issue 3 - The user needs to be redirected back to the section they were up to so they can have another go.**

We will refer to these 'issues' later on when we write the code.

Like the `Lbl` and `Goto` commands, an `If` statement is always partnered with a `Then` and an `IfEnd` statement. The `If` statement structure looks like this,

```
If 'enter condition here'
Then
'enter code here'
IfEnd
```

Please take note of the line break after the `Then`.

Now since an `If` statement controls the program, you will find it in,

Ctrl ⇨ **If** ⇨ `If` and `Then` and `IfEnd`

Marty Schmude 2008

Solutions to our 'issues'

Issue 1 - The user should be warned at the prompt that the number needs to be a whole number.

This is a real easy one to fix. We will simply add the word 'whole' to each of the text strings when we prompt the user to enter a number.

```
--------- multiply --------
Lbl start
ClrText
Input A, "Please enter your
first whole number"
Input B, "Please enter your
second whole number"
Print "The product is:"
```

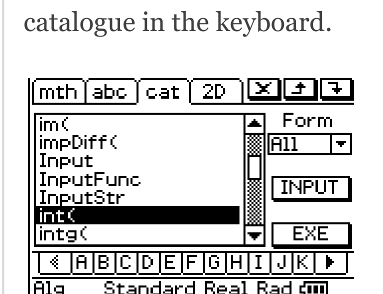
Issue 2 - If the user didn't read the warning properly, they need to be told why it is incorrect.

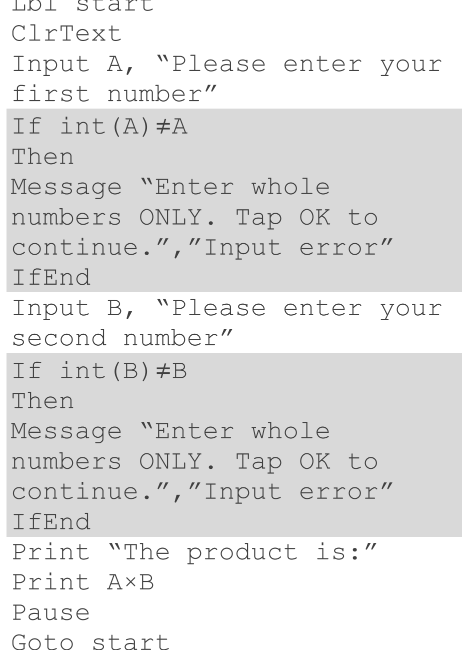
If the user enters a number that is not an integer (whole number), then we will show a message to let the user know this condition. The code that creates the message will be in the `if` statement. Colloquially, the code we are about to write says, *'if the user enters a number that is not a whole number, then pop up a message that tells them, then send them back to where they were up to'.*

The 'sending them back' part will be done when we talk about Issue 3.

The `if` statement needs to go after the user inputs a number. That way we can do our test with the number. Recall that the test is if `int(A)≠A`.

Remember that the `int` command comes from the catalogue in the keyboard.





```
--------- multiply --------
Lbl start
ClrText
Input A, "Please enter your
first number"
If int(A)≠A
Then
Message "Enter whole
numbers ONLY. Tap OK to
continue.","Input error"
IfEnd
Input B, "Please enter your
second number"
If int(B)≠B
Then
Message "Enter whole
numbers ONLY. Tap OK to
continue.","Input error"
IfEnd
Print "The product is:"
Print A×B
Pause
Goto start
```

The ≠ sign can be found on the keyboard too, by pressing **MTH ⇨ OPTN**.

Issue 3 - The user needs to be redirected back to the section they were up to so they can have another go.

This is another pretty easy one to fix too. We are going to 'send the user back to where they were up to' by using two new `Lbl` and `Goto` commands.

Assuming the user needs to be sent back to enter the number again, the `Lbl` needs to go before the `Input` prompt, so the program will run the `Input` code again for them. The names for the `Lbl`'s are arbitrary, but we will call them `first` and `second` if you like, referring to the number to be entered.

The `Goto` command needs to only run if they have entered an incorrect number. That means that it should be in the `Then` section of the code, thereby only running after the error message has been shown.

```
--------- multiply --------
Lbl start
ClrText
Lbl first
Input A, "Please enter your
first whole number"
If int(A)≠A
Then
Message "Enter a whole
number ONLY. Tap OK to
continue.","Input error"
Goto first
IfEnd
Lbl second
Input B, "Please enter your
second whole number"
If int(B)≠B
Then
Message "Enter a whole
number ONLY. Tap OK to
continue.","Input error"
Goto second
IfEnd
Print "The product is:"
Print A×B
Pause
Goto start
```

Make sure you go through the code carefully, and read each line slowly, and try to understand what each line is doing.

Marty Schmude 2008

How does it work

Let's just look at the first number, `A`, because the second number, `B`, is just doing the same thing.

```
Lbl first
Input A, "Please enter your
first whole number"
If int(A)≠A
Then
Message "Enter a whole
number ONLY. Tap OK to
continue.","Input error"
Goto first
IfEnd
```

Perhaps to best understand the code above is to look at two different scenarios.

Scenario 1 – The user enters a whole number

Let's assume the user enters a whole number after being prompted. The value of `A` is a whole number, so `int(A)=A`, which does not satisfy the `If` condition. The result is that the program completely skips the `If` statement and executes the very first thing after the `IfEnd`.

```
Lbl first
Input A, "Please enter your
first whole number"
If int(A)≠A
Then
Message "Enter a whole
number ONLY. Tap OK to
continue.","Input error"
Goto first
IfEnd
```

No problem here. It will skip this entire `Then` section.

Scenario 2 – The user enters a number that is not a whole number

If the number entered is not a whole number, then `int(A)≠A`, which satisfies the `If` statement's condition. The result is that the program will now execute the `Then` section of code.

The first thing it does is pause the program and shows a popup a message telling the user that the number needs to be a whole number. They are then instructed to tap OK to continue (tapping Cancel will stop the program completely).

Once they tap OK, the program resumes running. It reads the `Goto` command that sends the program to the `Lbl` with the same name.

You will notice that because of where we placed the `Lbl`, the first thing that the program does after jumping to the `Lbl` is to prompt the user again for the number. **From the user's point of view, it is a completely seamless and well-informed journey.**

LESSON 9 – Meet the `rand` function

Pseudo random generators

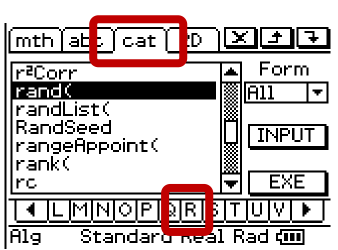
Pseudo random number generators are quite common in technology nowadays – for example, your iPod would use one when it ‘shuffles’ your playlist. You might be wondering why it is called pseudo random numbers. The reason is that the numbers generated are not random at all. The generators use a number called the ‘seed’, which it puts into an algorithm that then churns out a massively long number. The technology then chops the number up to produce the ‘random’ numbers. You can find out more at Wikipedia.

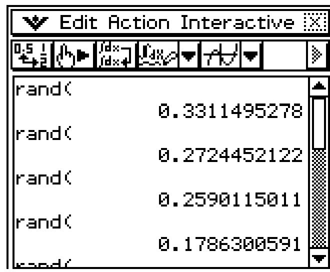
More info: http://wikipedia.org/wiki/Pseudorandom_number_generator

For people in the mathematics world, the ability to generate random numbers is an extremely useful tool. We are going to start a whole new program in our next lesson, but before we do that, we need to learn about the `rand` function on the calculator as our new program will be using it to generate numbers on a die.

What does `rand` do?

Let’s head into the MAIN application (Main) again. The `rand` function is found the same way we found the `int` function. You have to bring up your keyboard and look in the catalogue (**cat**) for `rand`. Tap R, then on `rand` and finally INPUT. Press EXE repeatedly and take note of the numbers that are produced.





You might have to change your settings to get a decimal answer by visiting,

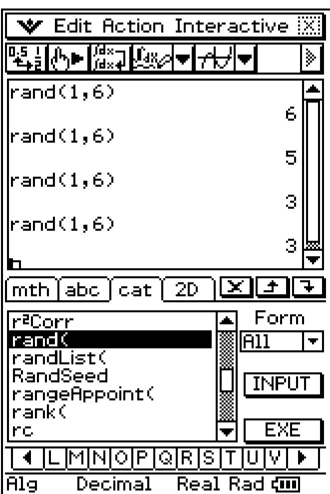
Settings ⇨ **Basic format** ⇨ `Decimal calculation`

The `rand` function simply generates numbers from 0 to 1, not inclusive.

Generating numbers 1 to 6

Generating numbers 1 to 6 on the ClassPad is completely simple! You simply use the `rand` function and state the lowest and highest values you want it to generate between (in this case, 1 and 6 respectively).

```
rand(1,6)
```



Now the one catch is that it only produces whole number values, so if you want to do in between values, you will need to do some tricks like generate large numbers and then divide by 10, 100, 1000…

Give it a try

1. Experiment with a few different values and see if the results are what you expect. The numbers in the brackets are called the function's *arguments*. Here are some arguments to try in your `rand` function.
   a) 0 and 10
   b) -5 and 5
   c) 0.1 and 4 (problem?)

2. Generate numbers 0 to 10, to 1 decimal place? As a hint, try starting with `rand(0,100)` and **change the answers with another operation and the number 10**.

The `randList` function

This function is fantastic! Talk about making things easy! If you want to create a bulk set of random numbers, this is the bomb ☺. The syntax is simple, you just have to add one more argument by telling the ClassPad how many numbers you want it to generate.

```
randList(number of trials, lowest number, highest number)
```

So for our die scenario, if we wanted the result of 4 rolls, it would look like this: `randList(4,1,6)`, with one result looking something like `{3, 5, 6, 2}`. Give it a try yourself.

LESSON 10 – Start the *sumdice* program

A more useful program

So far, the programs we have done have not been all that useful, but hopefully have been helpful for you to learn about programming in a familiar situation. We are now going to start a new program, and something that will give a more interesting result.

Our new program will add two dice together and analyse the outcomes.

You may like to have a go at building the start of this program by yourself, so below is a list of the initial structure of the program. Just remember that there is no one-way to code. The way you might write something is different to how another programmer might. Obviously there are some things in common, and there are some ways that are better than others too, but overall, everyone has their own style.

1. Start a new program called *sumdice* (LESSON 1).
2. Clear any text from the output window (LESSON 1).
3. Optional – a message to the user telling them about the program (LESSON 2).
4. Prompt the user to enter the number of rolls they would like and insist on only having whole number values (LESSON 4).
5. Shown later – Use the `randList` function twice to simulate two dice and store them in list 1 and list 2 (uses part of LESSON 9).
6. Shown later – Add the results of the list 1 and list 2 and store it in list 3
7. Shown later – Produce a histogram graph of the data in list 3.

Here is the code for the first part (up to point 6 written above). You could probably understand most of it now, but we'll go through it anyway.

```
--------- sumdice ---------
Lbl start
ClrText
Input A,"How many rolls
would you like?"
If int(A)≠A
Then
Message "Please enter whole
numbers ONLY. Tap OK to
continue.","Input error"
Goto start
IfEnd
randList(A,1,6)⇨list1
randList(A,1,6)⇨list2
list1+list2⇨list3
Print list3
```

Optional line of code

Save your program and run it.

We will now look at the section of this that we have not seen before, namely,

```
randList(A,1,6)⇨list1
randList(A,1,6)⇨list2
list1+list2⇨list3
Print list3
```

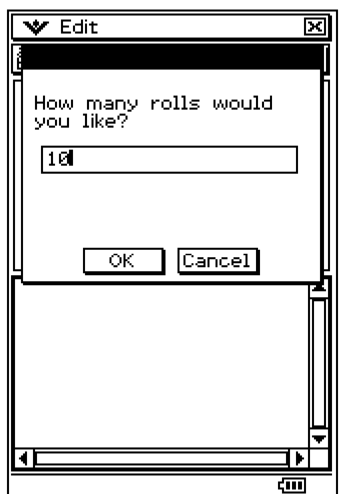
How it works

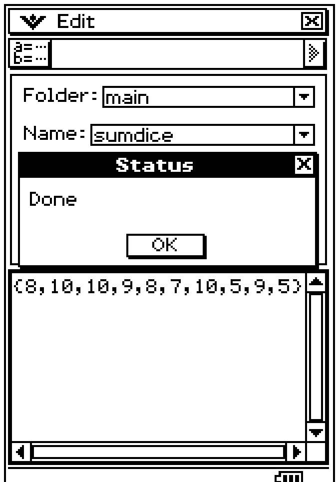
```
randList(A,1,6)
```

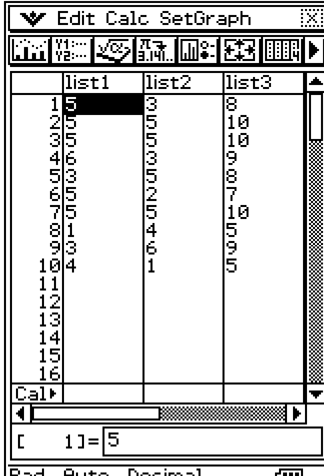
Recall that `A` is the variable that stores the input from the user when asked to enter the number of rolls they would like. This piece of code is creating a list of random results (numbers 1 to 6) with the number of elements in the list being however many rolls the user wanted – if the user wanted 10 rolls, then `A=10` and the list will have 10 elements to it.

```
randList(A,1,6)⇨list1
```

The rest of the line says, *'store the results of `randList` in list 1 of the STATISTICS application'*. This is done so we can more easily view and analyse the data later on.








```
list1+list2⇨list3
```

This line adds the results of the dice together (list 1 and list 2) and puts the data in list 3.

```
Print list3
```

This line simply prints our results in the Output Window for us to see immediately the outcomes of the rolls. This line is really only good for small values of `A`. It would be pointless if you do 200 rolls of the die, as you could always just visit the STATISTICS application to see the results. We will be removing it from the code later on.

The next lesson is going to involve drawing a histogram of the data in list 3.

LESSON 11 – Create a graph

Learning the syntax

To code a program to draw a graph requires some specific knowledge of the syntax used by the technology. In our case, the ClassPad is capable of drawing so many graphs, that it is not possible to cover all of them in this booklet. Instead, we will restrict ourselves to one type of graph – the histogram – and leave the others for you to lookup in the manual, if you want (Section 12-6-32).

It is assumed that you have some idea about creating graphs on the ClassPad.

The syntax to draw our histogram (other graphs may have slightly different syntax) is as follows (with an explanation of each part below),

```
StatGraph 1,On,Histogram,list3,1
```

`StatGraph 1` – You can setup up to 9 graphs, but we will only be looking at a single graph.

Misc ⇨ **Statistics (1)** ⇨ `StatGraph`

`On` – This turns the graph 'on', which just means that it is active and will draw when we say 'draw'.

Misc ⇨ **Setup (1)** ⇨ `On`

`Histogram` – This just tells the ClassPad what type of graph we want.

Misc ⇨ **Statistics (1)** ⇨ `Histogram`

`List3` – This is the list that holds the data we want to graph

Cat ⇨ **L** ⇨ `list3`

`1` – This is the frequency of each element in List 3. If we had a frequency list, which we don't, we would have the list here.

Other graph features

It is not a good idea to just 'hope' that certain settings on the calculator will be the way you want them to be when you write programs. For example, are the axes on or off, is the view-window the right size you want it to be? Perhaps the user has adjusted the settings at another time, which would make your graph look incorrect or useless.

In order to not leave these things to chance, it is very important to include some extra lines of code to set things up the way **you, the programmer**, want them. How do you know which settings matter? Easy, experience and mistakes... The two settings we will insist on are:

- Axes are off. The syntax is: `SetAxes Off`

Misc ⇨ **Setup (2)** ⇨ `SetAxes`

Misc ⇨ **Setup (1)** ⇨ `Off`

- Statwind is set to auto (this means the ClassPad will automatically determine the size of the view-window). The syntax is: `SetStatWinAuto On`

Misc ⇨ **Setup (3)** ⇨ `SetStatWinAuto`

Misc ⇨ **Setup (1)** ⇨ `On`

To tell the ClassPad to draw the graph, you will find the `DrawStat` command here,

I/O ⇨ **Draw** ⇨ `DrawStat`

THE FINAL CODE:

```
--------- sumdice ---------
Lbl start
ClrText
Input A,"How many rolls
would you like?"
If int(A)≠A
Then
Message "Please enter a
whole number ONLY. Tap OK
to continue.","Input error"
Goto start
IfEnd
randList(A,1,6)⇨list1
randList(A,1,6)⇨list2
list1+list2⇨list3
StatGraph 1,On
,Histogram,list3,1
SetStatWinAuto On
SetAxes Off
DrawStat
```

The `Print list 3` line has been deleted

DO NOT enter line breaks in the StatGraph statement.

Give it a go! Enter around 500 trials and enjoy what you see!
Use Analysis ⇨ `Trace` **to analyse the results.**
When prompted, use Start=2 and Step=1

Conclusion

You've done it!

Wow, what a great effort you've made. Look how far you've come and the amount of stuff you've had to learn. You should give yourself a big pat on the back. I certainly think you've done a fantastic job!

You have visited many of the basic ideas of programming that are common to many other programming languages.

I've included a small extension section that will cover how to 'lock' your programs. This is particularly useful if you are a school teacher, and wish to distribute the program to your class. – nothing worse than students accidentally messing with your code!

I hope you've found this booklet helpful and please feel free to get in touch at mjschmude@hotmail.com.

Kind regards,

Marty

Extension – Locking a program

Benefits of locking programs

It is generally a good idea to make your code unavailable for others to edit. You may think it is to protect all your hard work and intellectual property, which is fine, but there is also a useability issue on the ClassPad.

You may have noticed when you finished running a program and closed the Output Window, it would take you back into the Program Editor. Now this is not a section where you want people to be fiddling if they don't know what they are doing (especially if you're a teacher and your class is running the program on their calculators). It is not hard to accidentally change the code with a bump of a button and then try to run it again and ... they get an error! So...

How to lock our *sumdice* program

1. Open the sumdice program
2. Tap **Edit** ⇨ `Compress`
3. You will prompted to enter a new file name. The calculator is about to create a backup copy of the file. This is so you can still have an editable version. It does mean that the new name cannot be the same, so we will call our new program something like, *sumdice2*.

   Once you tap OK, the ClassPad has created the following files:
   - *sumdice* (locked)
   - *sumdice2* (editable)