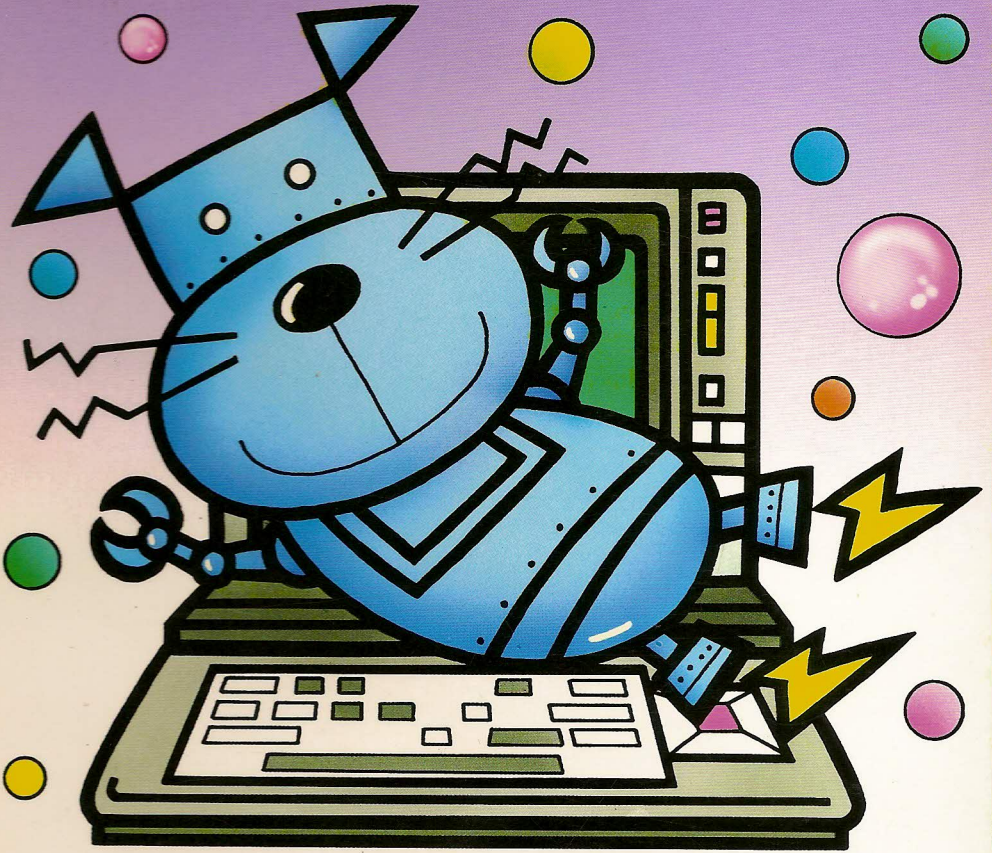


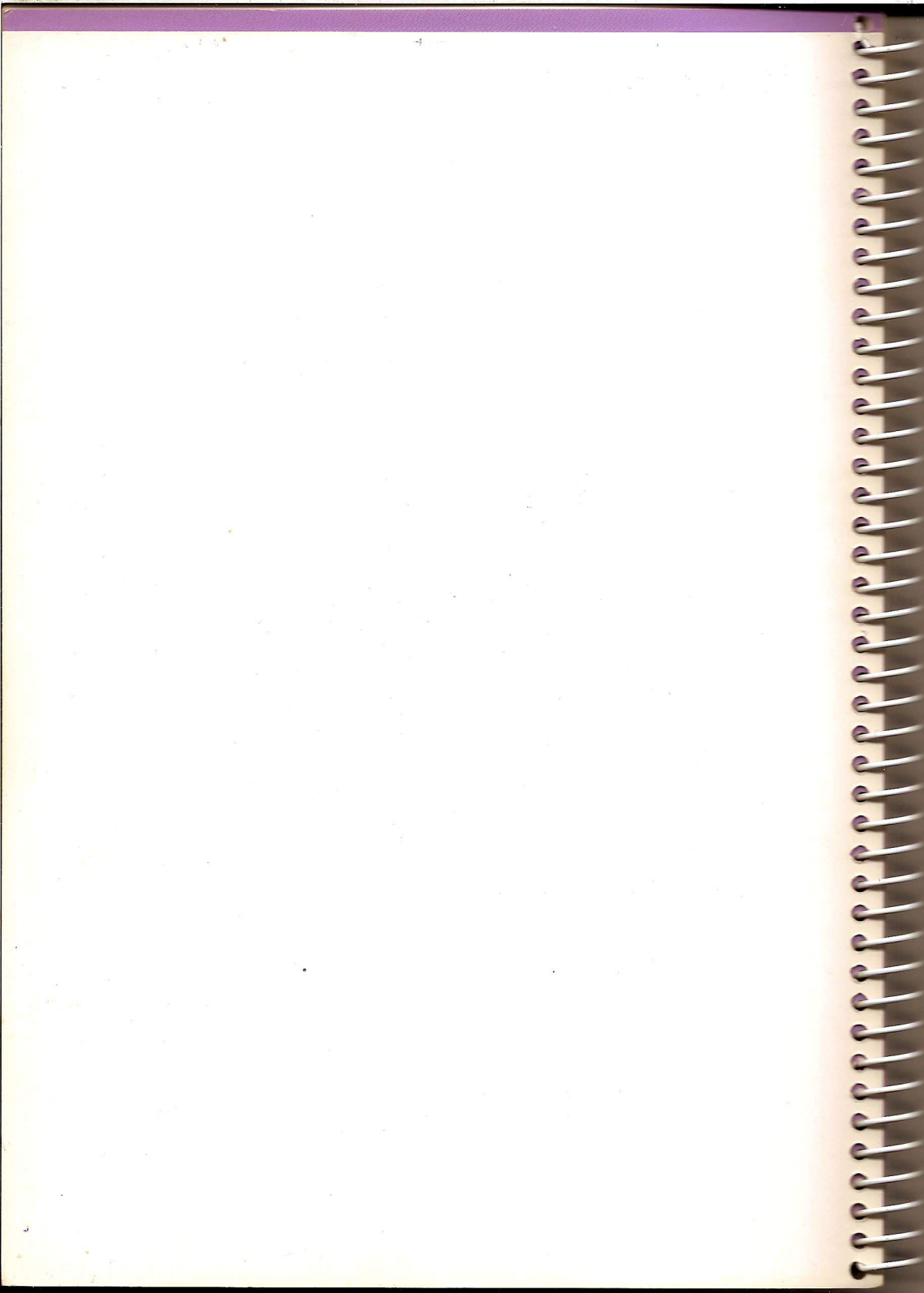
SONY®

MSX

# Introduction to MSX-BASIC



HIT BIT





Introduction to  
**MSX-BASIC**

# TABLE OF CONTENTS

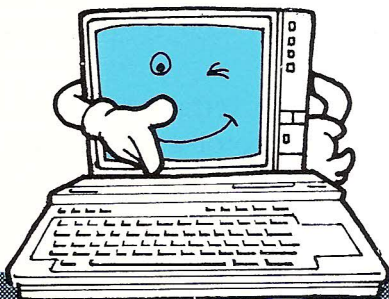
<b>Starting out</b> .....	4
About this book .....	4
Fido the smart dog .....	5
Your Sony computer .....	6
<b>Putting your computer to work</b> .....	7
Commands and inputs .....	7
Some tricks Fido can't do .....	10
Print the answer .....	14
<b>Numbers, letters, variables</b> .....	17
Numbers or letters: both the same .....	18
A letter becomes a variable .....	19
<b>Making your first program</b> .....	27
Planning a program .....	28
Writing a program: as easy as one, two, three .....	31
Correcting a program: check for bugs .....	32
Running your program .....	34
<b>Graphics: twinkling stars</b> .....	37
A graphic program .....	37
Random numbers .....	42
Put color in your programs .....	45
<b>Saving your programs</b> .....	52
Connecting the tape recorder .....	52
Making the computer talk .....	53
Did the tape really save? .....	55
Loading a program .....	56
<b>Decisions and games</b> .....	57
Hit or miss—the computer decides .....	59
Making the program simpler .....	60



<b>Graphics that move</b> .....	63
Display, erase, display, erase.....	66
<b>Putting it all together</b> .....	70
Adding color and sound.....	70
Guessing game + sound + video .....	72
<b>The slot machine game</b> .....	76
What is an array variable? .....	77
Making a slot machine.....	78
String variables.....	83
How to stop a loop with INKEY .....	86
What's the score? .....	88
The finishing touches.....	93
For easier program reading .....	95
Putting titles in the program .....	96
You've come a long way!.....	99
<b>BASIC command practice</b> .....	100
PRINT .....	100
INPUT .....	103
FOR—NEXT.....	105
IF—THEN and IF—THEN—ELSE .....	107
DIM (Array Variables).....	109
<b>Index</b> .....	113

# STARTING OUT

- About This Book
- Fido the Smart Dog
- Your Sony Computer



## ABOUT THIS BOOK

This book is going to introduce you to BASIC.

What's BASIC? BASIC is an easy language for "talking" to computers. The name of the language is BASIC because it is the simplest, most **basic** way to talk to a computer.

But why do we want to talk to a computer? Because your Sony Computer can help you do so many things—play games, solve problems, practice mathematics, store information, and much more. This book shows you how to begin having fun with your Sony Computer in only an hour or two.

First we will learn how your computer listens and talks to you (in BASIC language). Then we will use this special language to make your computer do some interesting tricks—simple ones first, and complex things soon after. Before the book is finished, you'll be making your Sony Computer operate a slot machine game such as you might see in a game arcade.

After you have used this book to become familiar with BASIC, you will be able to read and use programs in computer magazines, and share your fun with your friends who also have computers. If you want to learn even more about BASIC, there are longer, advanced books available.

There are many special words and commands that we use to talk about computers and computer programs. They are all listed in the **Index** at the back of this book. When you want to check the meaning of a word or a command, the Index will tell you where to find it.



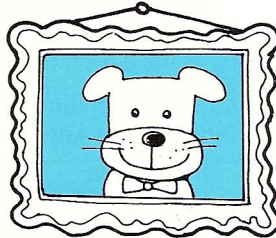
And there's one last, important bit of information you should know: computers really don't have bad tempers. They're both simple and **fun**, and they don't care how many mistakes you make as you learn how to use them. Nothing can "go wrong" with your Sony Computer if you happen to hit a "wrong" key—or any key. So have no fear. Just as with any new activity, it's natural that you'll have to correct things and back up to repeat steps, to get it right. Your computer will have all the patience in the world as you learn.

In fact, the only really important thing to remember as you begin this book is that in your Sony computer you've got a friend, who's just waiting for you to give computing a try.

## FIDO THE SMART DOG

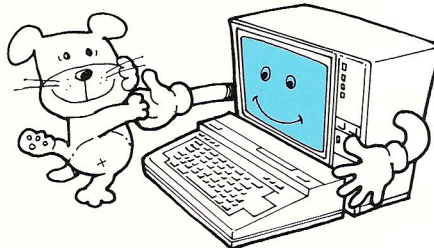
---

As we start learning BASIC, we will have a friend with us. This is our family dog, Fido. Like most dogs, he's friendly and faithful. Fido is just an ordinary kind of dog, but he's a rather smart one. He can do dozens of tricks whenever he is asked.



Fido

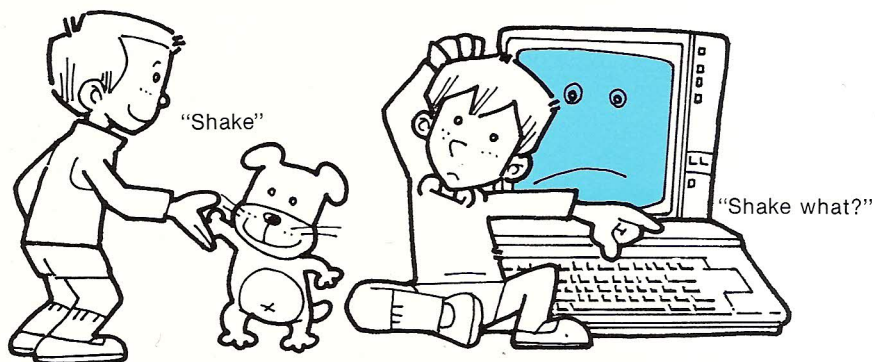
Why is Fido in a book about BASIC and computers? He is here because dogs are very good at following **commands** and doing tricks. And because Fido and your Sony Computer are both quite friendly. You will soon see that your computer follows **commands** and does tricks, too, but not quite in the same way. When we think about how Fido does things and how your Sony Computer does them, it will be easier to understand how the computer listens, talks and thinks in BASIC.



Friendly pair

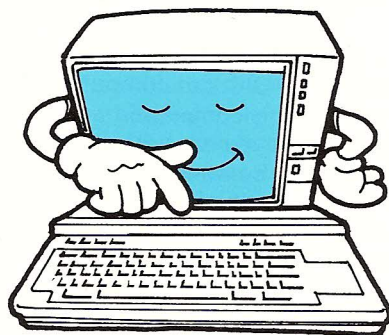
## YOUR SONY COMPUTER

Fido, of course, is good at tricks. "Sit," "Roll Over," "Find," "Bring"...these are pretty easy for Fido. What about your Sony Computer? Try "shake hands."



Fido understood, and gave you his front paw. But the computer has no hands. It doesn't even have ears!

Then, how do you tell a computer to do things? The computer's "ear"—its way of hearing your commands—is the **keyboard**.



Talk with your fingers

Before we learn BASIC, and before you go on to the next page, please read your Sony Computer's **Operating Instructions** to learn about the keyboard. It's not necessary to memorize all the keys. But it's very important to understand the **cursor**—what it is, and how it moves. You should also try typing some **letters**, **numbers** and **graphics** on the screen.



# ■ PUTTING YOUR COMPUTER TO ■ WORK

How to...

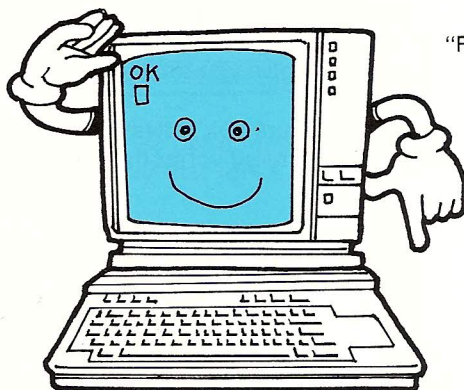
- Stop Your Computer
- Give Commands
- Type in Color
- Use the Shift Key
- Make Your Computer Do Arithmetic



Now that you have read the Operating Instructions and know about the keyboard, let's see what your Sony Computer can do.

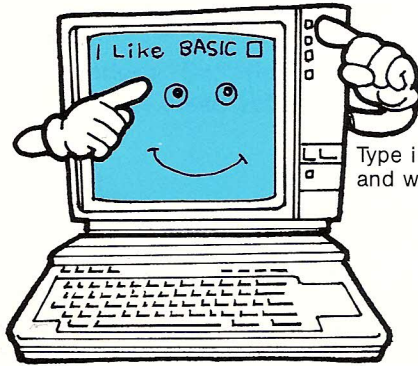
## COMMANDS AND INPUTS \_\_\_\_\_

We said earlier that the keyboard is the “ear” that the computer uses to hear our messages. Now, how do we know when the computer is listening to us? It tells us, with a friendly “Ok,” when it’s ready.



“Ready anytime!”

If you have been typing on the keyboard, the screen shows all the letters and numbers you typed. The cursor is in the middle of the screen—and the “Ok” message is left behind.

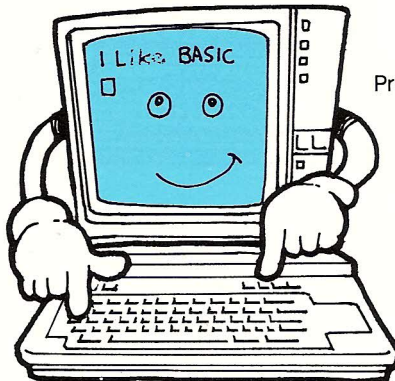


Type in a few words,  
and watch the cursor move.

If that's so, then we need to get the computer's attention. Press the **CTRL** key with one finger, and then press the **STOP** key with another. When **CTRL** and **STOP** are pressed **at the same time**, the computer stops listening to the old commands, and gives full attention to the new ones.

Commands are different from **input**. Input is the letters and numbers that make up the information you put into the computer—like all the letters it takes to write "I Like BASIC" on the screen. The input doesn't directly tell the computer to compute anything. We need **commands**, which are given with certain keys or typed words, to tell the computer what to do with the input. On a pocket calculator, for instance, in the sum  $2+2=4$ , think of "2", "+", and "2" as input. The "=" key is a **command** to the calculator, telling it to add 2 and 2 and give you the result, or output: 4. That's just how a computer works.

One more important point: the letters "Ok" **always** appear above the cursor when a **command** has been entered correctly: that means everything you have done is OK.



Press **CTRL** and **STOP**.

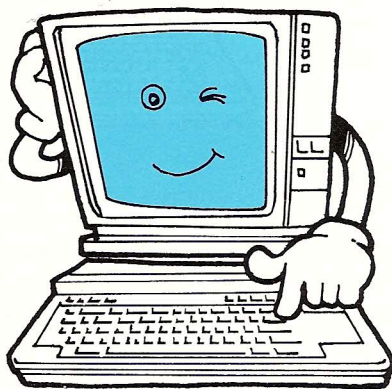


Now the computer is ready for your commands. You can enter commands in either small letters or capital letters—it makes no difference. Let's tell it to shake hands, by typing in "SHAKEHANDS".

## SHAKEHANDS

There is no response yet.

That's because we must tell the computer when to execute the command. The key for this is on the right, and is marked "RETURN". **Every time you give the computer a command**, you'll execute it with the **RETURN** key. It is a little bigger than the other keys, because it is used so often.

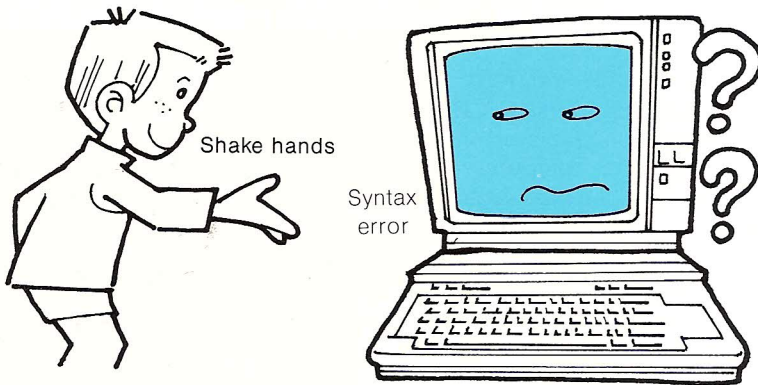


The **RETURN** key gives the commands.

Push **RETURN**, and now we get a response. Not a handshake, however, but a "beep!"—and a reply that looks like this:



That is an **error message**. It means the computer heard your command—but doesn't understand it. That's natural in this case, since computers don't usually know how to shake hands with people. This message, "Syntax error," means there is a mistake in language—the BASIC language, of course.



Our good dog Fido can understand "Sit" and "Shake" and many other words besides, but that's because he learned them from somebody. Our computer had a different kind of training, so the commands it understands are different. These commands are in the language called BASIC. And as you learn them you'll find that it is very easy to talk to the computer. Let's try a few simple ones right now.

## SOME TRICKS FIDO CAN'T DO

Here's an easy one to start with—though Fido wouldn't think so.

WIDTH 10

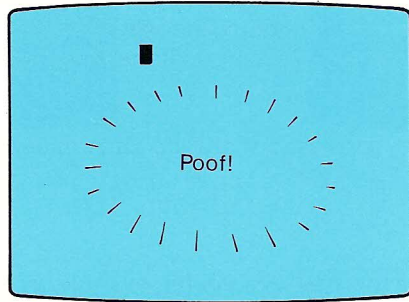
To type that into the computer is not hard, but be sure you tap the space bar once to put a **space** between the word WIDTH and the figure 10.

W I D T H    1 0

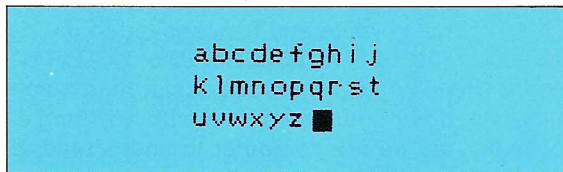
Don't worry if you make a mistake. Just use the directions in the **Operating Instructions** to move the cursor and correct any errors. The Sony Computer forgets all your mistakes as soon as you correct them. When your screen looks like this,

WIDTH 10 ■

you're ready to give the command. Push the **RETURN** key—and what happens?



The letters have disappeared, and the cursor is near the middle of the screen. But what did the **WIDTH 10** command mean? Let's type something, and we'll soon find out. Back to some **input**: let's type in the alphabet, with no spaces and no **RETURN**.

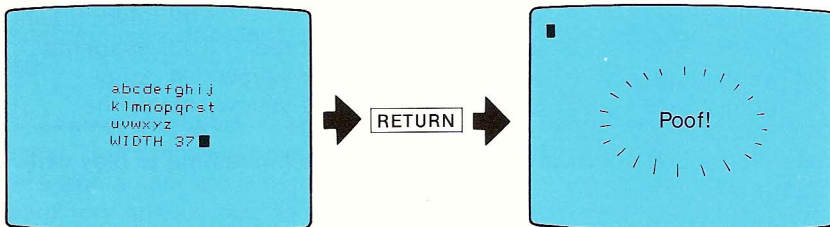


How many letters on each line? Ten. Without any more commands, our computer has understood that we want every line to be only **ten characters** wide. **WIDTH 10** is one of the commands it knows, and when it hears that command, it makes each line on the screen exactly ten characters wide.

Let's try another one—but first, press the **CTRL** and **STOP** buttons together, to “erase” the last inputs and prepare the computer for the **next** command:

**WIDTH 37**

Now press the **RETURN** key.





What now? Again the screen is empty, with the cursor back at the upper left side. You told the computer to make the line 37 characters long, and everything you type from now on, until you change that command, will come out in 37-character lines. (By the way, whenever you turn on the machine, 37 is automatically the length of the lines on the screen, unless you tell it to change.)

Now let's try another command that Fido can't understand. Hit `CTRL` and `STOP` first, and then type:

```
COLOR 8
```

(Don't forget to press `RETURN`.) Suddenly every letter on the screen is red! Type a few more words or letters. All are in the same color.

How about another color? First return the cursor to the left to the screen by pressing `CTRL` and `STOP`. Then type `COLOR 15` and hit the `RETURN` key—and you're typing in white again.

It's easy to see that 15 means white and 8 means red to the computer—but wouldn't it be easier just to type `COLOR RED` or `COLOR WHITE`? For people, and even for friendly Fido, words are easier than numbers. Computers, through, are better with numbers than with anything else, so we use numbers for things we want a computer to understand. BASIC is a language that often uses numbers to replace words.

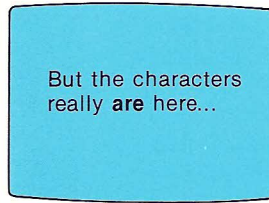
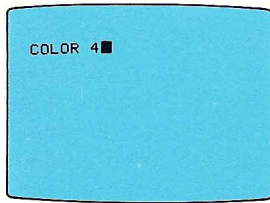
When you get used to giving commands to your computer, you'll be using numbers to say names, places, colors or sentences you want the computer to remember. It's not very different from using numbers for different people's automobiles or houses—and for the computer it's the most natural way to talk. Soon it will be very easy for you to use numbers as well as letters in your BASIC programs.

Now we know that 8 means red and 15 means white. How many colors are there?

code	color	code	color	code	color	code	color
0	transparent	4	dark blue	8	red	12	dark green
1	black	5	light blue	9	light red	13	magenta
2	medium green	6	dark red	10	dark yellow	14	gray
3	light green	7	sky blue	11	light yellow	15	white

There are sixteen colors in all, and they can be interesting to play with. Just for fun, try this:

```
COLOR 4
```



Dark blue characters on a dark blue background. Hmm, the evidence seems to have vanished.



Use the color code chart to try the different colors on the screen. Feel free to change your computer's colors—to suit your mood!

Now how about something a bit more tricky?

```
COLOR 1000
```

(Remember `RETURN`.) There is no number `1000` in the list of colors, and the computer knows that. A quick beep tells you the computer has a reply for you. It says "Illegal function call," which is the computer's way of saying "You can't fool me."

In fact, your Sony Computer can never be fooled on commands. Suppose you want to type

```
COLOR 6
```

but you make a mistake and type

```
CILOR 6
```

Without noticing the mistake, you press the `RETURN` key, and the computer will have another beep for you. This time the message is "Syntax error."

Of course, everybody makes some typing mistakes—even the pros at Sony! That's why we have the `BS` (backspace), `DEL` (delete), `INS` (insert) and cursor movement keys (see the Operating Instructions). You'll learn very quickly how to correct any mistakes.

### How many BASIC words are there?

Now you've learned your first two BASIC commands, WIDTH and COLOR. And you've seen how faithfully your Sony Computer reacts to commands entered correctly. It does as it's told when you spell them right, and it beeps and explains if you should get them wrong or use one that isn't on the list. How many BASIC command words are there altogether? There are about 100, and they tell your computer how to play music, how to draw pictures, how to solve mathematical problems and lots more besides. The MSX-BASIC Programming Reference Manual explains all of them. But you only need to know a few to get started in computing, so don't worry if 100 seems at first like an awful lot.

Your Sony Computer will also be teaching **you** some things as it goes along: it knows about 35 different **error messages**, like the "Syntax error" and "Illegal function call" that we just saw. It will usually tell you what is wrong if you give a command that it doesn't understand.

### PRINT THE ANSWER

---

Fido has his favorite tricks, and shaking hands is one of them. Your Sony Computer likes different tricks. Mathematics is one of the things it loves to do (and can do very well).

Let's type in

```
PRINT 3+5
```

Did you find the plus sign (+)? It's above the "=" mark, and it comes on the screen when you push the **SHIFT** key and the "=" **at the same time**.

Now push **RETURN** to enter this command into the computer.

```
PRINT 3+5
```

```
8
```

```
OK
```



You've done it! You've used the computer to solve a problem and print the answer. Let's try another:

```
PRINT 100-10
```

The minus sign, you'll notice, does not use the **SHIFT** key.

Now use **RETURN** to enter the command.



```
PRINT 3+5
8
OK
PRINT 100-10
90
OK
■
```

Of course the computer gives 90 as the answer. These are simple problems, but if you want to add or subtract trillions and billions, your computer will do it instantly.

How about multiplication? No problem at all. We just need to remember that the computer has a **special sign** for it. Instead of  $7 \times 9$ , it says  $7 * 9$ .

Type in PRINT  $7 * 9$ , and push **RETURN**.

```
PRINT 7*9
63
OK
■
```

To divide, we use the **/** key. Instead of PRINT  $120 \div 40$ , we type

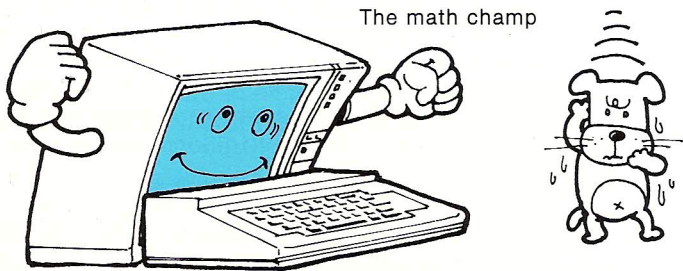
```
PRINT 120/40
```

And of course the computer knows the right answer when you push **RETURN**.

```
PRINT 120/40
3
OK
■
```

Are you ready to try large numbers ( $7654321 \times 18$ )?  
Complex problems  $(2+9) \times (6-8)$ ?

Go ahead and try whatever you like. Sometimes the computer may give an answer you don't understand, but don't let that worry you. Go step by step, think about the results, and you'll be doing marvelous things in just a few minutes.



# NUMBERS, LETTERS, VARIABLES



How to...

- Print Words and Phrases
- Create a Variable (a symbol with a changing value)
- Use Variable in Mathematics
- Give Different Names to Variables

We've been using the PRINT command to have the computer do math problems, and so far it's been acting just like a calculator. Now it's time to see some other kinds of calculation this machine can do. First, let's find out some more about PRINT, and get some typing practice at the same time by typing in

```
PRINT "JOHN AND MARY"
```

The **quotation mark (")** is up above the ' mark, and so you press the **SHIFT** key at the same time. Then, when everything is entered, push the **RETURN** key.

```
PRINT "JOHN AND MARY"  
JOHN AND MARY  
OK  
■
```

Clearly, the computer "reads" the quotation marks, and understands the PRINT command this way: it must print what is inside the quotation marks, but not the quotation marks themselves. Here are some more examples:

```
PRINT "ABC TO XYZ"  
ABC TO XYZ
```

```
PRINT "How are you?"  
How are you?
```



Of course you have to push the **RETURN** key each time. That should be easy to remember by now, so we won't be reminding you of it from now on: just tap the **RETURN** key each time you want to enter a command.



## NUMBERS OR LETTERS: BOTH THE SAME \_\_\_\_\_

Now let's try something a little different.

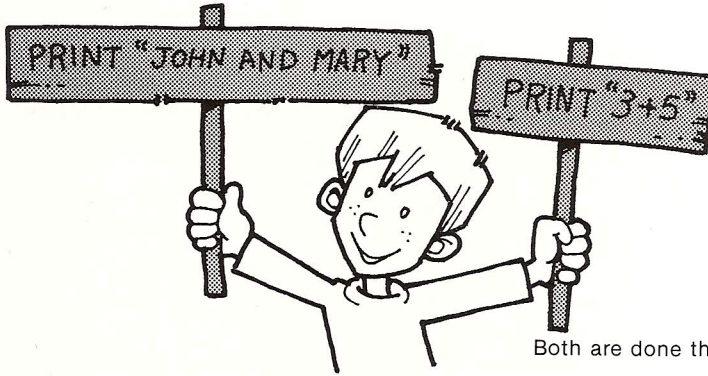
```
PRINT "3+5"
```

This looks a little bit like what we did a few pages back. At that time, we asked the computer to **PRINT 3+5**, and it kindly printed 8—because there were no quotation marks.

With quotation marks, this is what we get:

```
PRINT "3+5"  
3+5  
OK  
■
```

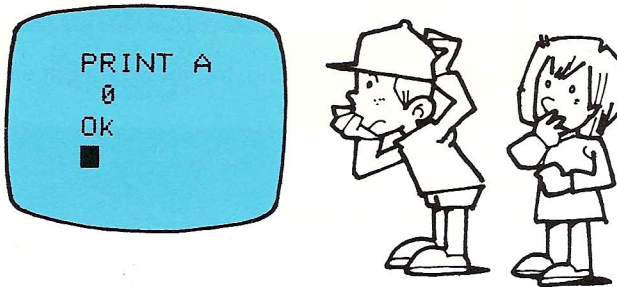
The **PRINT** command in this case—used together **with quotation marks**—tells your computer to read what is inside the quotation marks, and put it on the screen just as it is. Because of the quotation marks, the computer reads "3+5" as three symbols to be printed—not as a math problem to be answered.



That's what happens when a number is inside quotation marks.

## A LETTER BECOMES A VARIABLE \_\_\_\_\_

Now, we're going to learn another interesting thing about the PRINT command. What happens when a letter with no quotation marks comes after PRINT?



What's going on here? No beep! No "syntax error"! Instead we have only a 0, and then the Ok, telling us the computer has carried out our command, and is waiting for the next order. But what **was** our command?

If there's no error message, it must mean that PRINT A is a perfectly acceptable command. What does it tell the computer to do? To find out, do this:

```
LET A=3
```

When we push the **RETURN** key there's no error message, but also not much action, just that friendly Ok. The computer accepted our command, so something must be happening inside.



What's happening inside the computer?

Now let's put these two commands together:

```
LET A=3
OK
PRINT A
3
OK
■
```

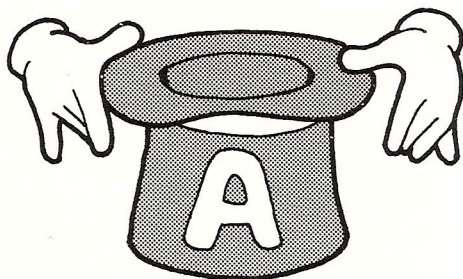
You probably have an idea of what's happening, but let's try it once more:

```
LET A=379
OK
PRINT A
379
OK
■
```

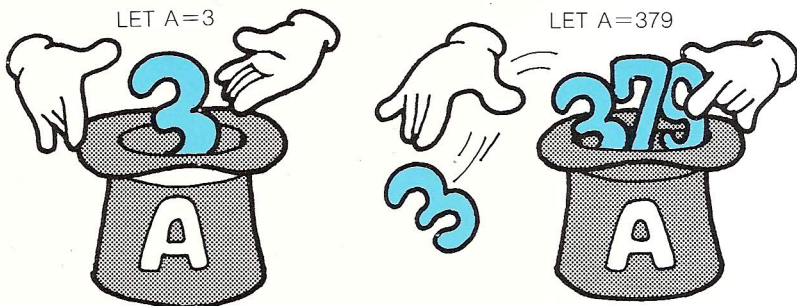
**A** was **3** a moment ago, but now it's **379**. If we want to, we could vary it again, just by writing **LET A=** and putting in any other number. That's why we call it a **variable**.

This kind of **A** is not really a letter of the alphabet, then. Rather, when it appears after **PRINT** but without quotations, it's something like a container that can hold any number we **LET** it.





A magician's hat is a good image for variables, because we can put values in the hat and pull them out like magic when we want them.



If we type `LET A=3`, we've put a 3 in the hat, and we can pull it out just by typing `PRINT A`. We can change the value that's in the hat to 379 with just a single simple command (`LET A=379`), and that value will come up when we ask for it.

`LET`, then, is a command which gives a specific value to our variable. Variables have many uses in BASIC, and they will be used very often in this book. For convenience, though, we can write this command faster, by leaving out the word `LET`:

`A=3`

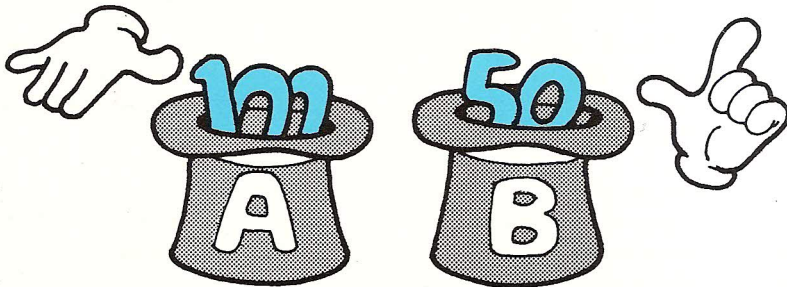
means the same thing as `LET A=3`.

Now, to make sure you remember everything, let's do a little review:

A=100	Give the value 100 to A
OK	
PRINT A	Display A
100	Here it is
OK	
PRINT "A"	Display the letter "A" (not the Variable A)
A	Here it is
OK	
B=50	Give a value to B
OK	
PRINT B	Display B
50	Here it is
OK	
■	

Earlier, we gave the command PRINT A, and the computer replied with 0. This means that any variable has the value of zero, until we give it a different value.

A variable can be represented on the screen by any letter. We have already used A and B, and given them different values.



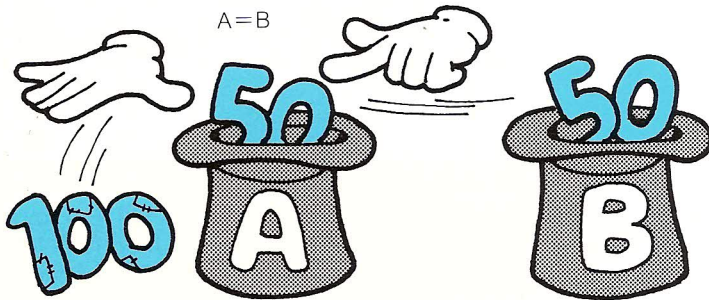
Now let's see what we can do with these variables.

We already told the computer that A=100 and B=50. Now give three commands: A=B, PRINT A and PRINT B. You can probably guess the result, but give it a try, for practice.

```
A=B
OK
PRINT A
  50
OK
PRINT B
  50
OK
■
```

Notice the first line. When we typed in  $A=B$ , we told the computer to give the value of B to A. Earlier, we changed the value of A from 3 to 379, and the old value just disappeared.

It's the same thing here again, changing A from 100 to B, or 50.

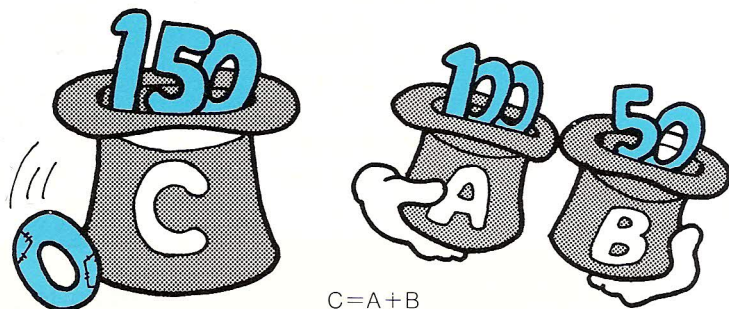


Now we can do some mathematics. Type in these four commands:

```
A=100
OK
B=50
OK
C=A+B
OK
PRINT C
  150
OK
■
```



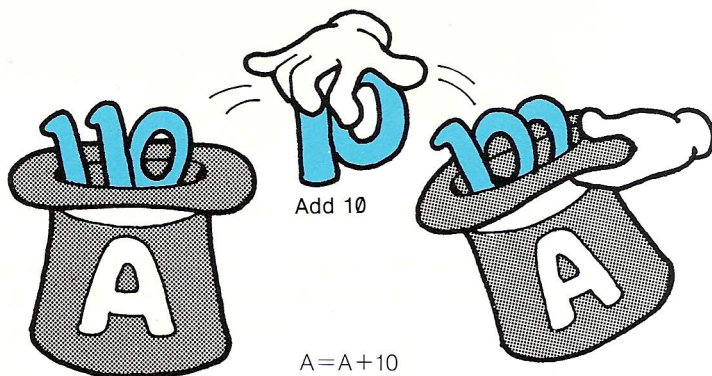
Here we have three containers, A, B and C, and the computer does the mathematics to find out how much to put in container C.



Now, here's a more interesting one:

```
A=A+10
OK
PRINT A
  110
OK
■
```

Does it seem confusing? A had an “assigned value” of 100. But, by reading this new command, the computer understood our directions to “LET A now have a value that is 10 more than it was before.” So, A is now 110.



You might like to try the same command again. And again. But before each time you do it, you should be able to figure out easily what the result is going to be. Your computer will **always remember** the amount assigned to the variable.

### More About Variables

Now you know what variables are, and how they work. Next we will see how they are used, and then we will use them in programs and games. Variables have many, many uses. One variable might keep the score of a game (a number which varies) each time points are added—or subtracted. Others might mean the changing position of a spaceship, or the speed of a racing car.

We have already used three letters—A, B and C—as variables. Besides these, we can use almost **any** letter or group of letters.

```
UFO=5
OK
PRINT UFO
5
OK
■
```

As you can see, a group of three letters without spaces between them, in this case UFO, can stand for a **single** number.



Also, it's important to remember that each different letter or group, such as A, B, or AB, has a different meaning.

```
A=3
OK
B=5
OK
PRINT A
  3
OK
PRINT B
  5
OK
PRINT AB
  0
OK
■
```

Here, we've assigned A the value of 3, and B is 5—but no value for variable AB has been entered, so the computer displays AB with a value of 0 (not 35!). A variable always has the value of 0 until we give it a different value.

Your name for a variable can have two, three, ten—or even 100—**characters**, and those characters can be **numbers** as well as **letters**. But **the computer will only read the first two characters and the first character should always be a letter**. This means that POA, POB, PO1 and PO2 will all be seen as the same, single value by the computer. So be a little careful when choosing names for your variables.

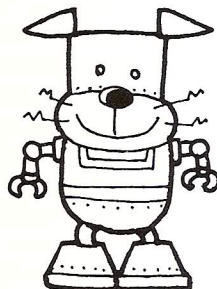
Any word that is part of the BASIC language **cannot** be used as the name of a variable. When your Sony Computer is given BASIC words, it reads them as **commands**, not variables. This means, for example, that LET, GOTO and PRINT cannot be variable names, and neither can GOTOA, BLET, or any other of the commands we will soon be learning.



# MAKING YOUR FIRST PROGRAM

How to...

- Plan a Program
- Write a Simple Program
- Make Your Computer Memorize Your Program
- Correct a Program
- Run Your Program
- Read Your Completed Programs on the Screen
- Erase a Program from Your Sony Computer's Memory



Until now, we've practiced the basics of using a computer: putting letters and numbers on the screen, using some of the symbols that the computer understands as part of its own "language," and giving simple instructions (commands). In other words, we've been telling the computer to **do** things on the screen, just the way you want them done. So we've already learned how people and computers communicate.

But these things are not much more than what a good pocket calculator can do. A calculator also adds and subtracts, and does other kinds of math, and shows us all the steps and results along the way, on its electronic display.

What makes a **computer** a truly exciting thing for you (and a truly valuable thing for scientists and many other people) is that it can go far beyond that. Your Sony Computer can:

- store a long series of **your special instructions** in its **memory**;
- use the instructions to perform many different tasks, games and tricks whenever you want it to;
- understand the result of each function, and carry the result on to the next function; and
- combine the results of all of the earlier steps as it continues to do the later steps.

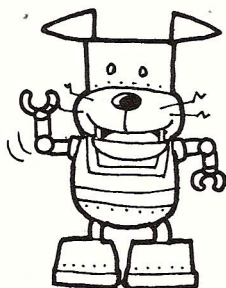
In other words, your computer can follow a complete **program** of functions automatically, and much more quickly than any calculator.

**Programming** is the process of putting instructions into the computer, to tell it which functions to perform and what order to follow. That's what lets you use a computer to create your own video games, draw your own video pictures, make your own electronic music. And that's what we're going to begin to learn in this chapter: we're going to make our first, simple program.

Making a computer program is not difficult. You already know how to give your computer some commands, and a program is just a series of commands. The important thing about a program is the **sequencing**: the **number** of commands and the **order** in which you give them.

We'll explain programs a little more clearly, by using another example of a performing dog. Fido is resting now, but we will try a few tricks with—**Superdog!**

Superdog is a different kind of friendly family pet: he's a robot! (Why not? Almost everything is automated nowadays, isn't it?)



I'm Superdog,  
and I follow programs!

Fido can do many tricks, as you know, but actually they are all very simple sequences. When you throw a ball, he will find it, pick it up in his teeth, bring it back to you, and release it. He will understand your command to “shake hands,” lift a paw, and let you shake it. We love Fido for his cleverness and many other reasons—but he can only do sequences of three or four commands at one time. To keep him performing, you have to go on to another trick—a new “program.”

## PLANNING A PROGRAM

---

Superdog is different. He has a microprocessor that remembers things and makes decisions, so he can perform many, many more steps than Fido can in a single trick, just as a calculator can do many more math problems than you can in the same amount of time. For example, we can tell Superdog to:

1. Find the ball and bring it back!
2. If the ball is white, turn around three times and bark!
3. If the ball is black, lie down for 10 seconds!

The first step is easy for Fido. But the other two are much more complex: the dog must be able to identify the ball as either black or white, then use that information to decide which trick to perform, and then do the trick correctly.

Also, each trick requires careful counting: the dog must know exactly how many times to turn around, or exactly how long to lie down. And to do the tricks correctly, he must always know, at each moment, exactly how many turns or seconds have been completed, and how many remain before he barks or brings back the ball. So Superdog is making judgments, decisions and computations all the time he is doing his tricks.

There is one more important difference between Fido and Superdog. Our family dog is a living animal, with a brain somewhat like our human brain, so he can do things by himself, without our commands. If Fido sees a cat, he will bark; if he sees a fire, he will run away. But Superdog is a machine. He can do very complex tricks, but he cannot “think” about them or remember what to do. Superdog does **only** what he is **programmed** to do, and he will never do anything until a human commands him to operate his parts—motors, sensors, components and the microprocessor—in a particular sequence.

So, when you write the program for Superdog’s trick, you will first have to think about everything you want the machine to do—**every** little step, and where it must be in the sequence. Something like this:

1. Receive and identify the command to “Find the ball and bring it back!”
2. Use image sensors to watch the moving ball and locate it.
3. Operate mechanical legs to move to the ball.
4. Use image sensors to identify the ball as “black” or “white”.
5. Operate body parts to pick up and return the ball.
6. Choose whether to “turn around” or “lie down”.
7. Operate body parts to do the required action.
8. Calculate the number of turns or seconds completed while performing.
9. Bark/Don’t bark.
10. Stop moving when the trick is finished.
11. Prepare to receive the next program command.



You must check all of these steps to be sure you haven't forgotten something, or made a mistake, and then enter them into Superdog's microprocessor. If you thought of everything—and listed it all in the right sequence—Superdog will perform perfectly, every time he does the trick. You can “run the program” once, ten times or ten thousand times, and he will always do the same thing.



So there are two main rules for writing a program: think of **everything** that the computer must do to finish exactly what you want, and put all of the steps in the **right order**. The only special “skill” that you need is normal, everyday logical thinking, because computers, just like any machines, work the same way your brain works—logically.

Now, let's write the first program on your Sony Computer, and see how simple it really is.

## WRITING A PROGRAM: AS EASY AS ONE, TWO, THREE

---



I know some fine tricks, too!

“Sequence” in a computer program means only that commands are written down in a one-two-three order. Remember our trick for Superdog: it had three separate parts, and we wrote a number in front of each.

1. Find the ball and bring it back!
2. If the ball is white, turn around three times and bark!
3. If the ball is black, lie down for ten seconds!

These steps are in a logical order: we want him to bring the ball back before he does the other things, so command 1 must come first. (Commands 2 and 3 could be reversed without changing the trick.)

We will enter our commands into the computer in the same way: we will write them on the screen in the same order in which they must be done, and we will give each command a number. Our first “job” for the computer—our program—will be to display the words “JOHN AND MARY” on the screen.

```
10 PRINT "JOHN"  
20 PRINT "AND"  
30 PRINT "MARY"
```

You already know the PRINT command, and here we use it three times—once for each of the three words we want printed. Of course, our sequence is the same as the word order. The only thing different from the way we wrote PRINT commands earlier is that each one starts with a number: 10, 20 and 30. This is very important: these numbers tell the computer **what command to do next** as it performs each step of the program. In other words, what sequence to follow.

Now, type in the first command: 10 PRINT "JOHN". Don't forget: at the end, hit **RETURN** once.

1 0 [ ] P R I N T [ ] " J O H N " RETURN

What's on the screen?

```
10 PRINT "JOHN"
█
```

Fine. Your first command is entered, the cursor has moved to the next line, and now you're ready to enter the next two commands, in exactly the same way:

```
10 PRINT "JOHN"
20 PRINT "AND"
30 PRINT "MARY"
█
```

There's your program—it's almost ready to be used.

## CORRECTING A PROGRAM: CHECK FOR BUGS —

Before we "run" our program, we should do one very important last step: let's **make sure** we've entered every part of the program correctly. Remember that your Sony Computer is just a machine, and it can only read commands that are in the BASIC language. If there is a mistake—just one character or space entered wrong—then the command doesn't look like a BASIC word (or looks like the wrong word). This will stop the whole program, or make the computer to do the wrong thing. For example:

```
10 PRINT "JOHN"
      ↑
      | Mistake
```

Mistake like this are called **bugs** in the program. The PRINT command must be spelled correctly—or the computer won't even understand that you are giving a command.



If you find a mistake in the program now, you should be happy: it's better to find it now than later, and it's very simple to correct it.

You already know how to move the cursor, using the four keyboard buttons with the arrows on them. The cursor is now at the bottom of the program, so first we move it up to the "10" with the button. Next, we move the cursor six spaces to the right with the button, so that it is on the mistaken letter M.

```
10 PRINT "JOHN"
      ↑
      | Place the cursor here
```

To correct it, just type the right letter.

```
10 PRINT "JOHN"
      ↑
      | Press [N].
```

Now press **[RETURN]** once ... and the bug is out!

```
10 PRINT "JOHN"
20 PRINT "AND"
      ↑
      | The cursor moves here after [RETURN] is pressed.
```

If there are other mistakes, correct them by moving the cursor in exactly the same way. (If you need to, you can check the **Operating Instructions** to see how to add a character or space that was forgotten, or take out a character or space that is not needed.) When all mistakes are corrected, use the arrow buttons to move the cursor back to the very bottom—so that your screen again looks like this:

```
10 PRINT "JOHN"
20 PRINT "AND"
30 PRINT "MARY"
■
```

Perhaps you didn't make any mistakes in this simple program. But it's still very important to **check** every time. Soon our programs will have more commands, and many of the commands will be longer. Any bug, large or small, can stop a program, and **everyone**—even the experts at Sony—makes spelling mistakes sometimes.

One thing is certain: if there is a mistake, your Sony Computer will tell you, sooner or later. An **error message** will appear on the screen, or the program will stop before you want it to, or the computer will do something different than what you wanted. Then you will have to look at the entire list of commands, locate the bug, and correct it. So it's better to check and "de-bug" your program at the beginning, when you are typing the commands.

#### Rules for programming

- Think out **every step your program will need** to make the computer do what you want it to do
- Make sure you list all commands—in the **logical sequence** that will give the right result
- Always use a **sequence number** at the **start** of each command line
- After you've entered your commands, **check each one carefully** for "bugs"—and correct any that you find

## RUNNING YOUR PROGRAM

Now that you've entered the commands and made the corrections, you're ready to **RUN** the program. That's the command we enter now on the screen:

```
RUN
```

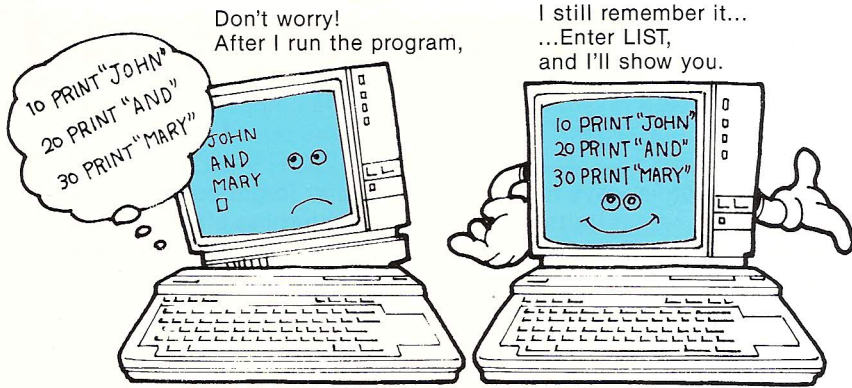
And that's what the computer does with your program when you've entered it, and pressed **RETURN**. Your screen now looks like this:

```
10 PRINT "JOHN"  
20 PRINT "AND"  
30 PRINT "MARY"  
RUN  
JOHN  
AND  
MARY  
OK  
■
```

It took only a few seconds for your Sony Computer to "read" the commands and print.

Type the RUN command again, just after the cursor, and press **RETURN** again. The result is the same, isn't it? The result for **this** program will always be the same, until you change or erase the commands. Try it several times more, and see for yourself. JOHN AND MARY will appear every time. That means that your Sony Computer **remembers** the program.

You can check the memory anytime by entering **LIST** (and then pressing **RETURN**). We will use the **LIST** command every time we write a program.



The most useful thing about your Sony Computer is that it will remember your commands. All you have to do is tell it to remember, by putting a **number before the command**.

```
10 PRINT "JOHN"
```

————— If a line number is entered, it's a program.

This command has a number, so the computer knows that it is part of a program, and it will memorize the command. 10 is the **line number**, and a line number can be any number from 0 to 65529.

How long will your computer remember your program commands? Forever!—or until you tell it to forget. The forget commands is

NEW

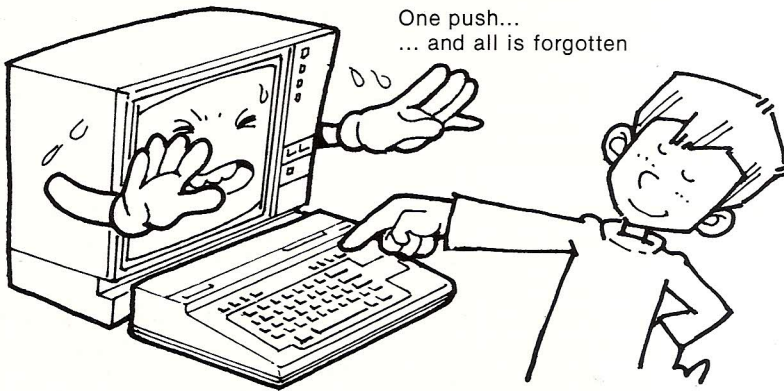
Let's make sure the computer is remembering, by entering some numbered commands, and then entering LIST. Did the computer list them? Good. Now enter NEW.



```
LIST
10 PRINT "JOHN"
20 PRINT "AND"
30 PRINT "MARY"
OK
NEW
OK
■
```

The screen still shows your commands, but the computer has forgotten them. Now try LIST again. The program is gone out of the memory.

There are two other ways to make the computer forget commands. One is pressing the **RESET** button, and the other is turning off the computer. **Don't do these two things** unless you are sure you want the computer to forget.



Later we will learn how to **save** programs, so you can use them again weeks or years later, by using a tape recorder. Until then, your computer will forget everything when you turn it off.

# GRAPHICS: TWINKLING STARS

How to...

- Use More BASIC Words
- Clear the Screen
- Draw Dots and Lines
- Make the Computer Repeat an Action
- Use a Variable
- Use a Random Number
- Use New COLOR Commands



## A GRAPHICS PROGRAM

Let's use some new BASIC words and learn what they do. Type these commands:

```
10 SCREEN 2
20 PSET (100,100)
30 PSET (150,100)
40 PSET (100,150)
50 PSET (150,150)
60 GOTO 20
```

This is a **program**—a set of steps that the computer can use to get something done. Now that the program is in the memory, give the RUN command, and we will see what this program does.

All the letters are gone, and four white dots have appeared on the screen. The cursor is gone, and if you push a key nothing appears on the screen, right? The computer is busy running the program. Why does it take so long?

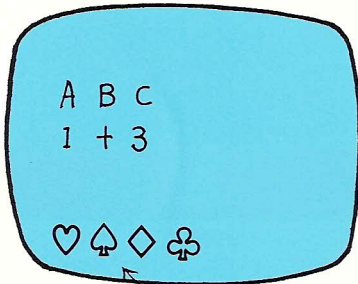
Let's see what this program means. First press **CTRL** and **STOP**, to turn off the program. The dots are gone and the cursor has returned. Now enter LIST, and we can look at the program line by line. First look at the line 10.

```
SCREEN 2
```

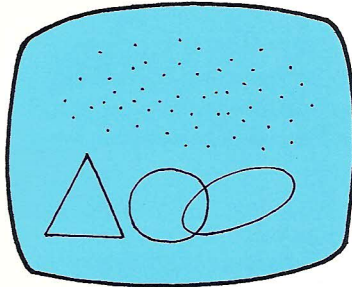
is the command which lets you start to draw **graphics** (pictures) on the screen. Graphics are dots, lines, circles and other figures or designs that are displayed on the screen. (Displays that are not graphics—numbers, letters and symbols—are called **characters**.)

SCREEN commands tell the computer in what form information will be displayed on the screen. There are two kinds of SCREEN commands.

SCREEN 0 or SCREEN 1 is for **characters**



SCREEN 2 is for **graphics**



(Symbols on the keyboard are characters)

SCREEN 2, then, makes the computer display graphics. When you gave the RUN command, the characters on the screen disappeared, because SCREEN 2 orders the computer to display graphics. When you pressed **CTRL** and **STOP**, you cancelled the graphics command, and the characters came back. Now that you understand the first line of the program, we can move on.

```
20 PSET (100,100)
30 PSET (150,100)
40 PSET (100,150)
50 PSET (150,150)
```

The explanation: PSET is the command of putting **dots** on the screen, and the numbers in brackets ( ) tell the computer where to put the dots.

### Repeat, Repeat, Repeat, Repeat

The last line of the program has a very important command:

```
60 GOTO 20
```

GOTO means "go to", so this command tells the computer **to go back** to line 20 and start over. When it gets to line 20,

```
20 PSET (100,100)
```

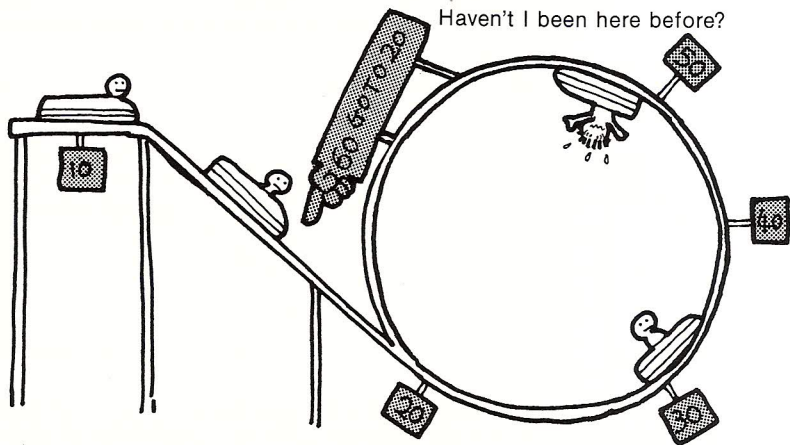
the computer naturally repeats the program. At line 60, we go again to line 20, repeat the program, go back to 20, repeat the program...



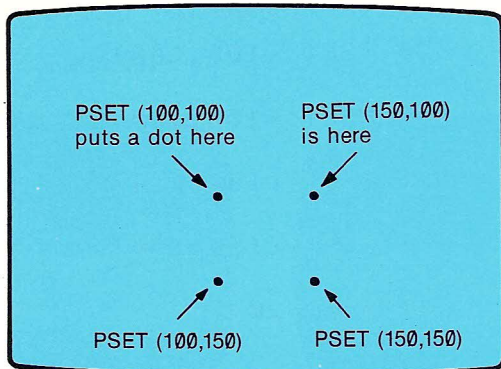
```
10 SCREEN 2
↓
20 PSET (100,100)
↓
30 PSET (150,100)
↓
40 PSET (100,150)
↓
50 PSET (150,150)
↓
60 GOTO 20
```

This part is endlessly repeated

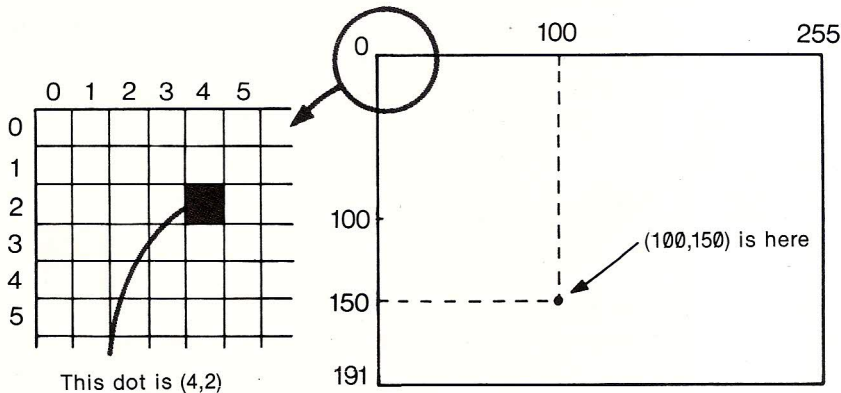
A repeating section in a program is a **loop**, and it is a common and useful tool for computer programming. (Fortunately, not all loops repeat endlessly.)



Let's see how those PSET commands put dots on the screen.



Do you see how the numbers work? For both dots on the left, the first number is 100, and for both dots on the right it's 150. Here is a magnified picture of the screen.



To tell your Sony Computer where to put a dot, you have to give it two position number, the first for left-right position, and the second for up-down. There are 256 different positions from left to right (0 to 255), and 192 from top to bottom (0 to 191). So we can put a dot anywhere on the screen, between (0,0) and (255,191).

## Graphics and Variables

You remember that any number can be replaced with a variable. That's true for PSET numbers, too. If we make the left-right number into variable X, and the top-bottom variable Y, our command becomes

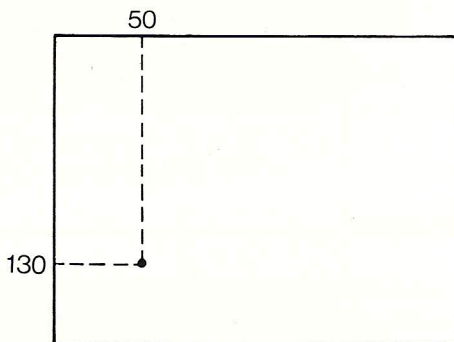
```
PSET (X,Y)
```

and we can use any numbers we like for X and Y, to make dots go wherever we want them.

Here's how to use the variables for a single dot. First use the NEW command to make your Sony computer forget the last program, and then enter this new program:

```
10 SCREEN 2
20 X=50:Y=130
30 PSET (X,Y)
40 GOTO 30
```

Put the program in the memory by pressing **RETURN**, then give the RUN command. Your computer starts with the command in line 10, which clears the screen for **graphics**. In line 20 it learns two variables, and in line 30 it uses them to put a dot on the screen.



Since X is 50 and Y is 130, there is a white dot on your screen at the (50,130) position. Next, the computer reads line 40—which sends it back to line 30. That means the only way to stop this loop program is to press **CTRL** and **STOP**.

Why did we use variables instead of just writing PSET (50,130)? And why did we make it repeat endlessly with line 40? These steps aren't needed if we only want to put a single dot on the screen for a very short time. But they are a good way to display many dots with a very short program. And that's what we will do next, after this note on punctuation and spelling.



**Note: Be careful with punctuation**

The program we are using now has three different kinds of **punctuation**—a , [comma] and a : [colon] and ( ), called brackets. These punctuation marks are **characters**, and they **must be put in the proper places**, just like the letters in the command words. Remember, your Sony Computer can't understand a command if there is a spelling mistake, and punctuation is part of the spelling.

Each punctuation mark has its own meaning in BASIC. The comma shows that one number has ended and another will begin. The colon means that one command is finished, and there will be another command on the same line. Brackets are necessary when two or more numbers, or two or more variables are used together—and brackets **always go in pairs**, first the left “ ( “ and later the right ” )”. Later, this book will show you some of the other punctuation marks that are used in BASIC commands.

As we said before, everybody makes spelling mistakes sometimes. When this happens, your Sony Computer usually cannot understand the command, so it displays an error message. For example,

```
Syntax error in 30
```

means that there may be a spelling mistake in line 30.

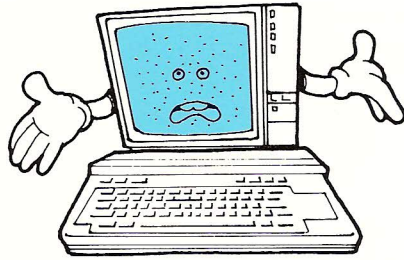
## RANDOM NUMBERS

---

Use the NEW command to clear the computer memory, then enter this program in the memory:

```
10 SCREEN 2
20 X=INT(RND(1)*256)
30 Y=INT(RND(1)*192)
40 PSET (X,Y)
50 GOTO 20
```

When you RUN it, the screen will fill up with more and more dots, until you stop it ... that's right, with **CTRL** and **STOP**.



What are you doing to my face?

Now let's explain the two new lines in the program.

```
20 X=INT(RND(1)*256)
30 Y=INT(RND(1)*192)
```

It's easy to see that these lines give values to variables X and Y. But what do those values mean? First, let's look at RND (1). RND means **random**, and it is one of the **functions** in BASIC. To understand this **random number function**, try this command.

```
X=RND(1):PRINT X
```

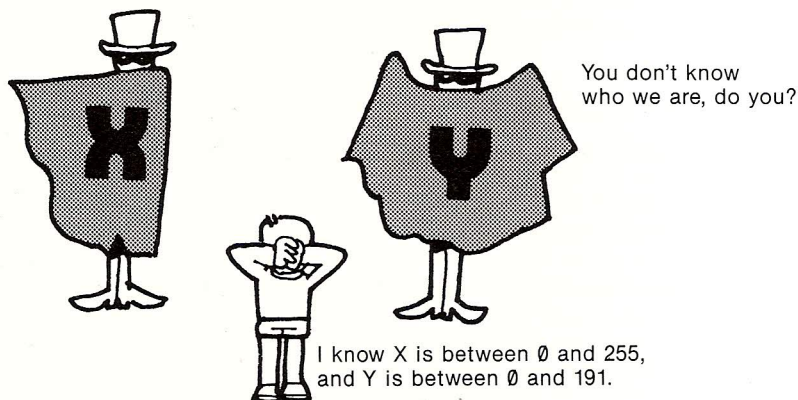
Press **RETURN** after entering this command: you will see a 14-digit fraction, a number that is less than 1 but more than zero, and begins with a **decimal point**. For example:

```
X=RND(1):PRINT X
.59521943994623
Ok
■
```

Try this command again. Did the number change? Try it again. Your computer has a long list of random numbers, and it is choosing one after another to use for X.

Going back to line 20 in the program, we have  $(\text{RND}(1) * 256)$ , which means that the random number is multiplied by 256. But if we multiply those fractions by 256, the results will usually also contain fractions—while our “dot” positions must be **whole** numbers, such as 1 or 30 or 192, because the location numbers on the screen are fixed in whole numbers only. That's why the command has **INT** at the beginning. **INT** means **integer**, or whole number, and it cuts off the decimal point and the numbers after it, to make every result a whole number. Now we can understand the value of X. It changes every time the program repeats the loop.

Every time your computer reads  $X = \text{INT}(\text{RND}(1) * 256)$ , it rounds out  $X$  to a whole number between 0 and 255—but you never know which number it will be. And every time it reads line 40, it makes  $Y$  a whole number, between 0 and 191, just the same way.



We already understand the next line,

```
40 PSET (X,Y)
```

even if we don't know what the values of  $X$  and  $Y$  will be. The computer won't tell us the values, because there is no PRINT command. Instead, it will keep displaying more dots somewhere on the screen, at the places where left-right and up-down integers tell it to.

Still, we have only explained how one dot actually gets on the screen. Where do the others keep coming from? This is where the loop goes to work. The next line,

```
50 GOTO 20
```

returns the program to

```
20 X=INT(RND(1)*256)
```

New numbers are then given to  $X$  and  $Y$  each time, so

```
40 PSET (X,Y)
```

puts a dot in a new place, and we see dot after dot after dot, all around the screen.



### **Programs: two kinds of commands**

A program, as we have said, is a **series of commands in a fixed, numbered order** that has been entered in the computer's memory. A program can have hundreds or thousands of lines—commands—but that's not necessary in most cases; we just made thousands of dots with only five lines in our program.

One of those five command lines—GOTO 20—didn't actually make the computer do anything, but rather told it **where to go next**. This was the line that made our short program do so much—by making a loop that repeats endlessly.

We can see now that there are two kinds of commands:

#### **1. Commands that tell the computer what to do, such as:**

PRINT    PSET    WIDTH    COLOR    etc.

and

#### **2. Commands that change the order of the program**

GOTO    IF—THEN    FOR—NEXT    etc.

There are only a few of the second kind of commands, and we will learn more about them later in the book. They are probably the most interesting tools in computing: Like GOTO, the one we used to make a loop, they are all used to make **small programs very powerful**.

## **PUT COLOR IN YOUR PROGRAMS** \_\_\_\_\_

We're going to add some excitement to the dot program—with color! To start, let's display the program that's in the computer's memory, with the LIST command. (If you turned off the power switch, you will have to enter the program again.)

```
LIST
10 SCREEN 2
20 X=INT(RND(1)*256)
30 Y=INT(RND(1)*192)
40 PSET (X,Y)
50 GOTO 20
```

Now change line 40 to: 40 PSET (X,Y),2

```
10 SCREEN 2
20 X=INT(RND(1)*256)
30 Y=INT(RND(1)*192)
40 PSET (X,Y)■
50 GOTO 20
```

Move the cursor to this position, and enter ,2. (Don't forget the comma—it's important.) Press **RETURN** once to enter, and again to move the cursor below the program. Now give the RUN command.

The dots have changed from white to medium green, because 2 is the code number for that color. Using the Color Chart on page 12, we can pick many different colors. Or, we can make the color a variable, and let the computer change the dots from color to color.

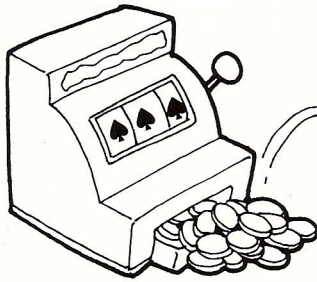
Go back to line 40 and change the color code to variable C.

```
10 SCREEN 2
20 X=INT(RND(1)*256)
30 Y=INT(RND(1)*192)
40 PSET (X,Y),C
50 GOTO 20
```

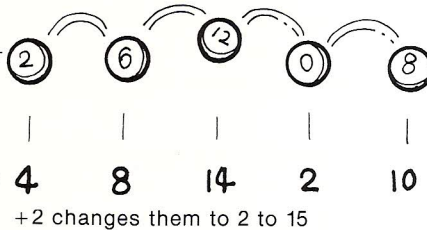
Now we must give a value to C, which is our command for what color we want. But it will be more fun if, again, we let the computer make the value, randomly. Here is the command line that will do that—but don't enter it just yet.

```
35 C=INT(RND(1)*14)+2
```

What does it mean? It is a random number command like those for X and Y, with one more step at the end. It chooses a number at random between 0 and 13, and then adds 2 to it (+2). That means the number is between 2 and 15. But there are 16 colors, from 0 to 15. Why does this command leave out colors 0 and 1? Because 0 is transparent and you can't see transparent dots, and 1 is black, which you can't see if the background becomes black, too.



Any number, 0 to 13



Where do we want to put this line in the program? It must be before the dots are displayed in line 40—so our new line becomes number 35. (And that's why our command lines are numbered 10, 20, 30... to leave space for any new commands that we decide to put in later.) Now, let's put it in the program.

```
10 SCREEN 2
20 X=INT(RND(1)*256)
30 Y=INT(RND(1)*192)
40 PSET (X,Y),C
50 GOTO 20
35 C=INT(RND(1)*14)+2
■
```

30, 40, 50, 35. Do you think the computer will understand? Of course it will, because it knows how to count, and it automatically arranges the lines in order by number. It's easy to check—just enter LIST command again.

```
LIST
10 SCREEN 2
20 X=INT(RND(1)*256)
30 Y=INT(RND(1)*192)
35 C=INT(RND(1)*14)+2
40 PSET (X,Y),C
50 GOTO 20
```

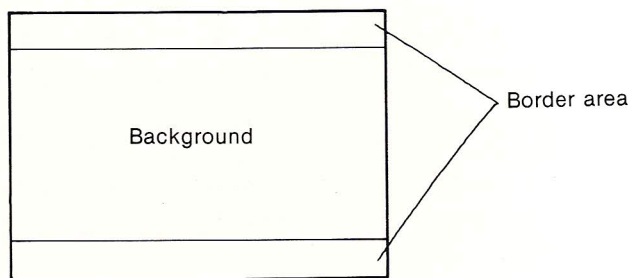
Now we're ready to turn our dots into **stars**. Stars can only be seen at night, so we want to make the screen black. Enter this new line:

```
5 COLOR 15,1,1
```



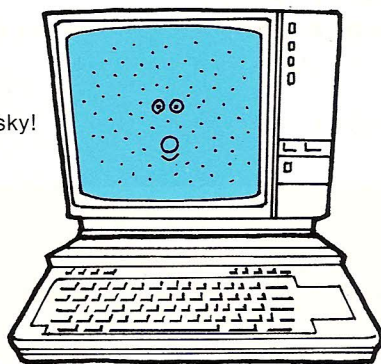
### New commands for color and lines

That line is a new way to use the COLOR command. It has three codes: 15 sets the **character** color (or foreground color), the first 1 sets the **background** color, and the last 1 sets the **border** area color. (This time, the background and the border are the same color. Any numbers will do, but let's try these.)

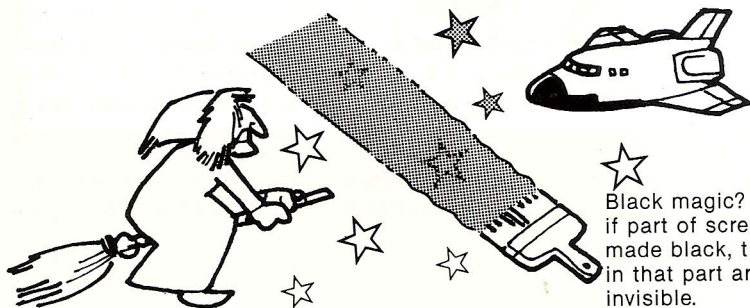


Now we're ready: RUN.

Stars in the night sky!



But we're not finished yet. We can make them look even more like natural stars. First we want to control the number of stars we're making—before we end up with a whole galaxy. Use **CTRL** and **STOP**, as usual.



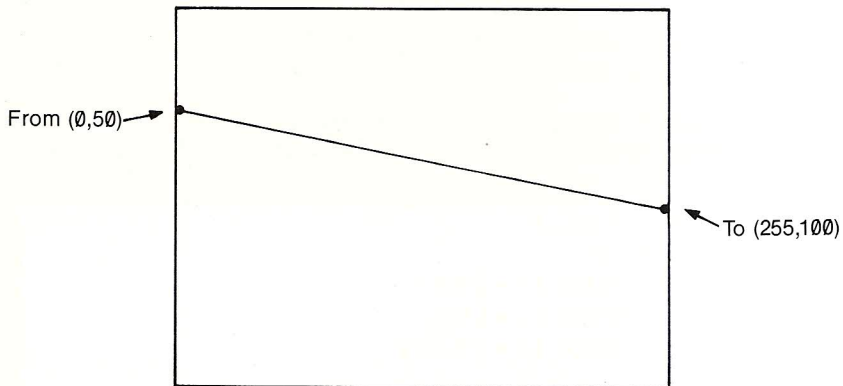
Black magic? No, but if part of screen is made black, the stars in that part are invisible.

Drawing **black lines** on the screen is one way to make a part of the screen black, of course. The PSET command is only for dots: for lines, we use **LINE**. Here is a command that tells the computer where to make a line, and what color to use.

```
LINE (0,50)-(255,100),2
```

For the [-] (hyphen sign) in the middle, by the way, use the minus sign.

The hyphen means “from/to,” and this command draws a line **from** position (0,50) **to** position (255,100). You know, of course that the last part of this command is the color medium green. If you use this LINE command in your program, it will make a green line like this:

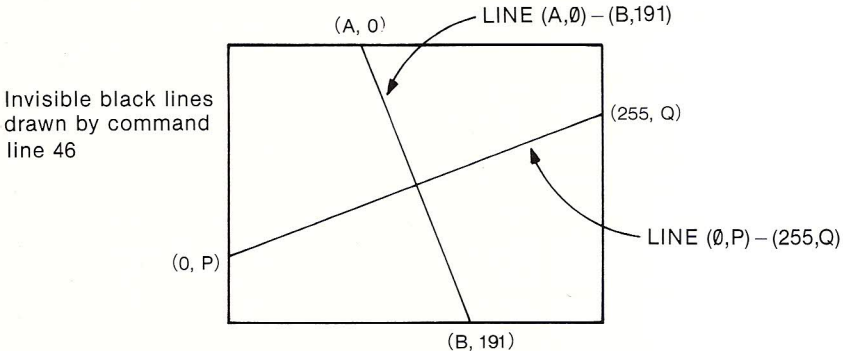


Now that you understand the LINE command, you can use it to improve the star program. We want to use a black line, to make some stars invisible, right? That means we will change the color code to 1. Also, we are going to use variables again, so that a random series of lines appear, along with the random series of dots. Enter these three new lines on your computer:

```
42 A=INT(RND(1)*256):B=INT(RND(1)*256)
44 P=INT(RND(1)*192):Q=INT(RND(1)*192)
46 LINE (A,0)-(B,191),1:LINE (0,P)-(255,Q),1
```

These three lines take more than one line each on the screen, but that is not a problem. The computer will automatically write each command on two lines, but it is still only one line, because there is only **one line number** (42, 44 and 46).

Line number 46 contains two LINE commands, using four new variables. Lines 42 and 44 are the random number commands that we already know, and they give values to the new variables A, B, P and Q. This is how one pair of random black lines would look:



Each time you have entered the new command lines 42, 44 and 46 in the computer, press **RETURN**, and then enter LIST to see the whole program.

```

5 COLOR 15,1,1
10 SCREEN 2
20 X=INT(RND(1)*256)
30 Y=INT(RND(1)*192)
35 C=INT(RND(1)*14)+2
40 PSET (X,Y),C
42 A=INT(RND(1)*256):B=INT(RND(1)*256
)
44 P=INT(RND(1)*192):Q=INT(RND(1)*192
)
46 LINE (A,0)-(B,191),1:LINE (0,P)-(2
55,Q),1
50 GOTO 20

```

Notice that the new command lines come before the GOTO loop in line 50. This means they will be repeated endlessly, just like the stars. Each time the computer runs from command line 20 to command line 46, it makes new dots in different places and then new lines in different places. And if a dot is located where a black line is drawn, the dot will become invisible. Now, give the RUN command ... and we have stars that **twinkle!**



If you want to actually see these lines, so you can understand them better, change the color code in command 46 from 1 to 2. Now you can see green lines appearing on the screen.

**NOTE:** If you stop this program, the screen will still be black. You can return to the original dark blue background and border with this command:

COLOR 15,4

15 means white letters and 4 is dark blue background and border.

And, though this is not yet explained before, the screen is automatically set in the SCREEN 0 condition when you first start the BASIC. In this condition, the border color is always set to the same color as the background color. If you want to have the background and border with different colors, try the SCREEN 1 command then give for example, COLOR 15, 4, 7 command.

To return to the original screen once again, give SCREEN 0 command.

## SAVING YOUR PROGRAMS

How to...

- Connect Your Computer to a Tape Recorder
- Transfer a Program to the Tape Recorder
- Check that the Program is Saved
- Load a Program from the Tape Recorder into the Computer



Have you ever seen a photograph of a large computer—one that is used to guide a rocket, or control a factory, or do medical research? You may have seen parts that look like big tape recorders. Actually, that is exactly what they are. On big machines they are called “tape drives,” and they record and store the information the computer uses.

Some of what they store is programs—the instructions for the machines, things like PSET, SCREEN, PRINT and other commands that you already know about. They also store **data**—facts like game scores, answers to problems, or scientific formulas.

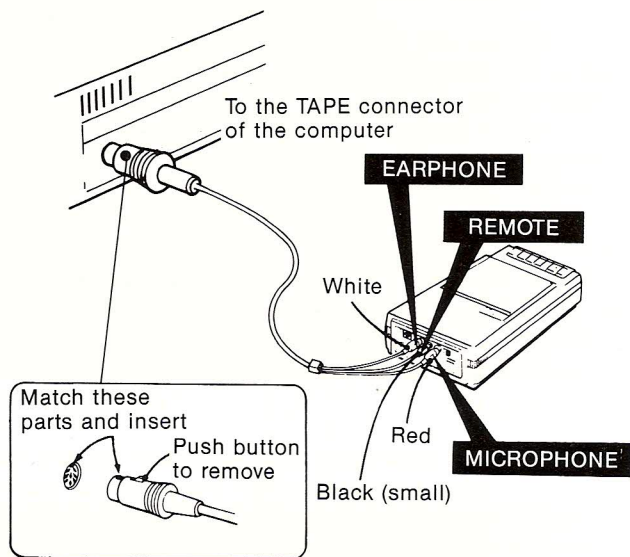
Computers have developed very rapidly, and things which used to require a big, “mainframe” computer can now be done on small machines, like your Sony Computer. Your Sony Computer is so small that a child can lift it, but it is faster and smarter than the old computers that cost many thousands of dollars and filled up entire rooms.

So it's no surprise that your Sony Computer can also store programs and information on tape. As it is smaller than a mainframe computer, it uses a smaller tape drive. Your Sony Computer can use a plain home cassette recorder that is used for music or dictation.

## CONNECTING THE TAPE RECORDER

It's simple to connect a tape recorder to the computer, using the special cable that came with your Sony Computer. One end is a round metal fitting with pins in the middle. This end fits into the plug marked TAPE on the back of the keyboard.

The other end of the cable has three parts, colored red, white and black. The red one connects to the MICROPHONE jack on your tape recorder and the white one goes in the jack marked EARPHONE. If your tape recorder has a REMOTE jack, plug in the black end of the cable. (If your tape recorder doesn't have a REMOTE connection, just leave the black end unconnected.)



Now the computer and the recorder are ready to talk to each other.

## MAKING THE COMPUTER TALK \_\_\_\_\_

The computer will tell the recorder what is in its memory when we give it this command:

CSAVE "file name"

The C of CSAVE stands for cassette, and SAVE means just that: save the program that is in the computer's memory, by recording it on the cassette. The "file name" (don't forget the double quotation marks) can be whatever name you want to give to your program, so that you—and the recorder, and the computer—can tell it apart from others.

A **file name** can have up to six characters. They can be letters, numbers or graphic signs, but the **first** character must be a **letter** of the alphabet.



STAR is a good file name for the program we made in the last chapter. It has four characters, starts with a letter, and it's convenient: it reminds us of the content of the program.

Let's save it:

```
CSAVE "STAR"
```

But don't press **RETURN** yet. When we press **RETURN**, the computer will start sending the information to be saved. First we have to make sure the recorder is ready for it.

If your recorder has a REMOTE connection and the cable is plugged in, then set the recorder in the record mode.

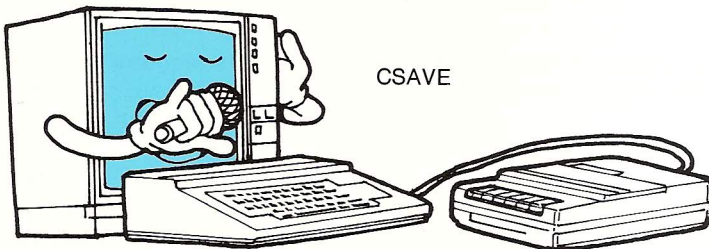
However, you will notice the tape does not start turning. That is because now, your computer is in "control" of your tape recorder. Press **RETURN**, and the tape will turn itself on, record the program, and turn itself off. When it's finished recording the screen will show

```
CSAVE "STAR"
```

```
Ok
```



If your tape recorder doesn't have a REMOTE connection, you **press the RECORD button** and the tape will start turning at once. **Wait** a few seconds for the tape speed to become stable, and then press **RETURN**. When the Ok comes on the screen, **press the STOP button** on the recorder, and you're almost finished.



Now you have the program in two places. It is still in the computer's memory, and if things went well, it is also saved on the tape.

## DID THE TAPE REALLY SAVE? \_\_\_\_\_

You should **always check** to make sure your **tape machine** really has saved your program—before you do anything else. Your Sony Computer will do the checking for you, by comparing its memory with the recording. First you must **disconnect** the cable from the “REMOTE” jack. Then **rewind** the tape to the place just before the program was **SAVED**. **Set the volume** of the tape recorder at about 1/2 of full. Now **reconnect** the REMOTE plug, and type this command:

```
CLOAD? "STAR"
```

If there is no “REMOTE,” hit **RETURN** **before** setting the recorder in the playback mode. If there is a “REMOTE” plug, set the recorder in the playback mode and the tape will start automatically when you press **RETURN**. When the computer hears the beginning of the program, the screen will show:

```
Found:STAR
```

As the computer plays back to itself it will compare the recorded program point by point with its own memory, and then display Ok if everything is correct. (Don't forget to stop the tape by hand afterward, if there is no remote control.)

This is an important process—you don't want to lose all the work you've done writing your program—and sometimes there may be a problem. If the “Found” message does not appear on the screen, perhaps you haven't rewound the tape far enough.

If the Ok sign does **not** appear, increase the volume on the tape recorder and try it again. If this fails, then there is electronic interference, or the program was not properly recorded. Check your connections, and then start over from CSAVE “STAR”.

If the check goes well and you see the Ok, you know the program has been saved. You can label that tape, put it away and use it when you need the program in the future.

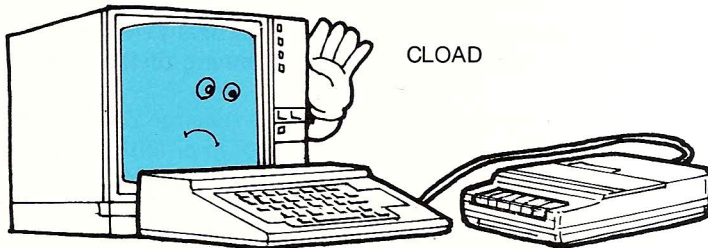
When you are sure the program is safely recorded on tape, you can type in NEW, or push the **RESET** button. This erases the **computer's memory**—but not the **external memory** on the tape.

## LOADING A PROGRAM

Using that tape in the future is called "loadig the program" into the computer, and this is a good time to practice. The procedure is almost the same as the checking you just did. Be sure the recorder is set to **PLAY** into the computer, and the tape is at the right place—just before the start of the recorded program. Then give this command:

```
CLOAD "STAR"
```

Notice that there is no question mark (?) this time. Of course, if you're using a different file name, type that inside the quotation marks instead of STAR.



When the computer has finished listening to the tape, the screen will show

```
CLOAD "STAR"  
Found:STAR  
OK  
■
```

Be sure the tape is stopped—and you're ready to **RUN** your program on the computer.

Occasionally, the computer will display

```
Device I/O error
```

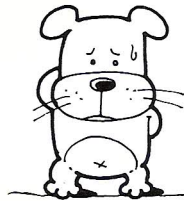
I/O means **input/output**, and in this case refers to the connection with the tape recorder. Change the volume just a little and try the loading again.



# DECISIONS AND GAMES

How to...

- Use Your Computer to Make Decisions
- Program a Conditional Formula
- Play a Game on Your Computer
- Make Your Program Simpler
- Improve Your Program



Remember Fido, our friendly family dog? He is very good at simple tricks, like "Sit," "Bring" and "Shake." Until now, your Sony Computer has been doing the same kind of simple tricks as Fido—following our commands to "PRINT," "GOTO," "PSET," etc.

Then we met Sueprdog, and he did some rather difficult tricks, based on "if the ball is balck..." or "if the ball is white..." Superdog could **make decisions**. You already know that your Sony Computer is smart enough to do these things, and now we're going to start writing "super-programs" that make decisions.

Start by clearing off your last program with the NEW command, and then enter:

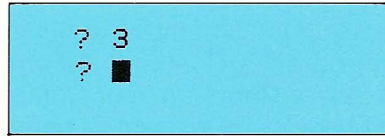
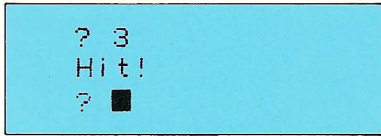
```
10 A=INT(RND(1)*5)+1
20 INPUT B
30 IF A=B THEN GOTO 50
40 GOTO 10
50 PRINT "Hit!"
60 GOTO 10
```

After typing it correctly, give the RUN and then **RETURN**.

```
RUN
? ■
```

The computer has given us a question mark before, but this time the cursor is on the same line, not the next one. That means it is waiting for you to **put something in**—because of the INPUT command in line 20. You input something—in this program, any number from 1 to 5—by simply pressing a key, as you know. So let's give the computer the number 3. (What happens if we input a letter or a graphic?)

Press `3` and `RETURN`. Which screen display do you see now?



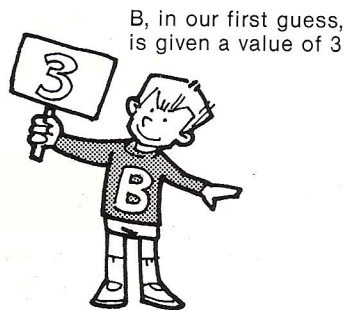
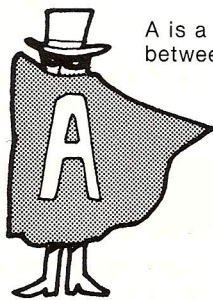
Both screens have question marks at the end, so you know the computer wants you to input another number. And another. And another. And another. That's the game: is your number a correct guess—a "Hit"—or a miss? How often do you get a "Hit!"? (The law of chance says you will get about one hit with five tries, or about two hits with ten tries, or about 20 hits out of 100.) You will become tired before the computer does (it **never** gets tired), so press `CTRL` and `STOP`.

What's happening here? Let's look at the program. The first line is our old friend, the **random number command**. Each time the program runs back through the loop (line 60), it chooses a number between 1 and 5, and makes it the value of variable A.

Now we see a new friend in line 20:

```
20 INPUT B
```

B is a second variable, and it is waiting for you to give it a value from the keyboard. After you type in a number, both variables have values, and the computer goes to the next line.



# HIT OR MISS—THE COMPUTER DECIDES

Line 30 reminds us of Superdog, because the first word is IF.

```
30 IF A=B THEN GOTO 50
```

IF **always** goes with THEN. For example, “IF you are hungry THEN you should eat” or “IF you have a Sony Computer THEN you can have fun.”

IF [condition] THEN [something]

What can be the **condition** of variables A and B? They can be equal ( $A=B$ ), or A can be larger ( $A>B$ ), or B can be larger ( $A<B$ ).

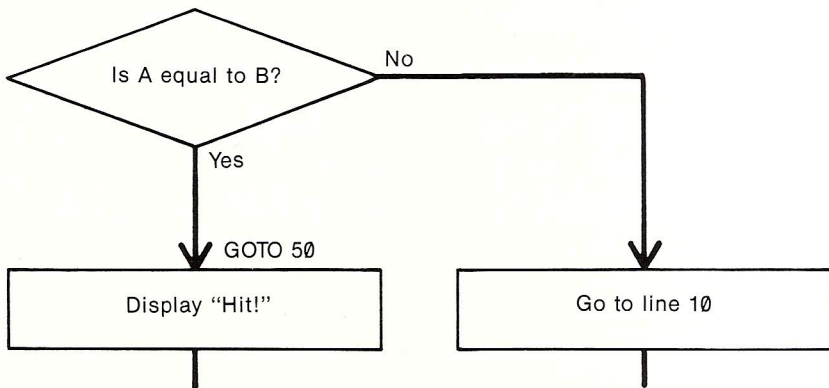
“Something” can be any command. In this case, it’s GOTO 50.

This is called a **conditional formula**. There are six of them in BASIC, and some of them can be written two different ways.

Symbol	Meaning	Example
=	Equal to	IF A=B    If A is equal to B
>	Larger than	IF A>B    If A is larger than B
<	Smaller than	IF A<B    If A is smaller than B
>=	} Equal to or larger than	IF A>=B } If A is equal to or larger than B
=>		IF A=>B }
<=	} Equal to or smaller than	IF A<=B } If A is equal to or smaller than B
=<		IF A=<B }
<>	} Not equal to	IF A<>B } If A is not equal to B
><		IF A><B }

Line 30, then, is a conditional formula. When the IF part is true ( $A=B$ ), the computer goes on to the command (GOTO 50). But when A and B are not equal, it goes on to the next line of the program, line 40, without reading the THEN command. Line 40, of course, sends the computer back to the beginning, to generate a new random number—which you try to guess.





Now you understand that “Hit!” can only be displayed when A and B—the number you supply, by guessing—are equal. “Hit!” means that the number you entered for variable B is the same as the value that the computer gave to variable A.

After displaying “Hit!” the computer moves to line 60, and the game starts again. The game continues in either case, if A and B are equal or if they are not equal.

## MAKING THE PROGRAM SIMPLER

Computer programs should always be written as simple as possible, so they are easier to understand and there is less chance of making a mistake in typing. Let’s take another look at our guessing program.

```

10 A=INT(RND(1)*5)+1
20 INPUT B
30 IF A=B THEN GOTO 50
40 GOTO 10
50 PRINT "Hit!"
60 GOTO 10
  
```

Now we know that if  $A=B$ , the computer will display “Hit!” It sounds like a single action, but in the program, there are actually two commands. The first, which comes after THEN in line 30, is GOTO 50. The second, in line 50, is PRINT “Hit!” A single action should be a single command—so we can change line 30 to

```
IF A=B THEN PRINT "Hit!"
```

And now we no longer need the commands in lines 50 and 60, do we?

```
10 A=INT(RND(1)*5)+1
20 INPUT B
30 IF A=B THEN PRINT "Hit!"
40 GOTO 10
```

Much simpler! If A and B are equal, after printing "Hit!" the computer will go down to line 40 and then GOTO 10. If A and B are not equal, it will do the same thing, but without printing "Hit!" Our game now works with only four commands.

Now make the changes on your screen. First change the command in line 30 from GOTO 50 to PRINT "Hit!". Then erase lines 50 and 60, this way: move the cursor to the position below line 60, and enter 50. Press **RETURN**, and line 50 is erased from the computer's memory. Do the same with 60 to erase line 60.

Did lines 50 and 60 disappear from the screen? No, they didn't! But they have been erased from the memory. If you give the LIST command, the computer will display what's now in the memory:

```
LIST
10 A=INT(RND(1)*5)+1
20 INPUT B
30 IF A=B THEN PRINT "Hit!"
40 GOTO 10
```

That's better, isn't it?

### Some refinements

We've made the guessing program as simple as possible, and now we can think of ways to make it better, by adding things that are convenient and fun. Of course, as we add them we will try to keep each improvement as simple as we can.

When A is equal to B, the computer tells you by displaying "Hit!" But when they are not equal, it doesn't give you a message before it starts again. The program will be easier for your friends to understand if we change line 30 to look like this:

```
30 IF A=B THEN PRINT "Hit!" ELSE PRINT "Miss"
```

ELSE means "if not." It's a useful word to put after a conditional formula, because it makes that step of the program clearer.

```
IF conditional formula THEN command 1 ELSE command 2
```

Since this new command is longer than 37 characters, it will "run over" onto the next line—making line 40 temporarily disappear. But don't worry—it's still in the computer's memory. Just use the LIST command to bring it back again! There's one more improvement that will make the program easier for your friends to understand when they see it. We can change line 20 to:

```
20 INPUT "Make a guess(1-5) ";B
```

You will understand the meaning of this new command if you enter it on your screen and then give the RUN command.

```
RUN  
Make a guess(1-5) ? █
```

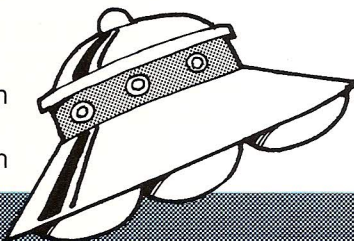
Very easy to understand now, isn't it? That's much better than the "?" message that we were seeing a few minutes ago. Now you can share this guessing game with your friends, and everybody will understand your program quickly.



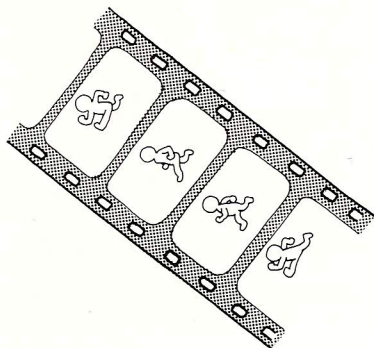
# GRAPHICS THAT MOVE

How to...

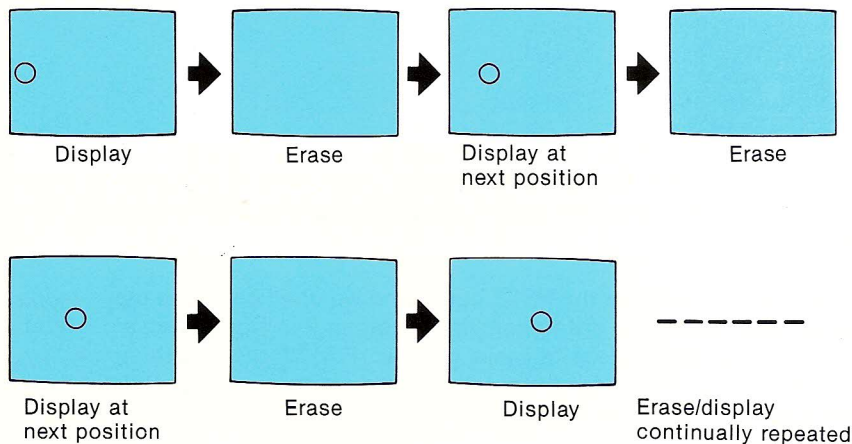
- Move characters around the screen
- Make a UFO movie
- Use variables to shorten a program



Everyone knows that movies don't really move. A movie is just a fast series of still pictures one after another. Because the pictures are each a little different, we see the illusion of movement.



Television is the same. The picture on our TV screen changes many times within a second, and as a result we seem to see motion. Your Sony Computer screen is almost the same as a TV screen, so naturally we can use it to make moving pictures. Here's how it's done:



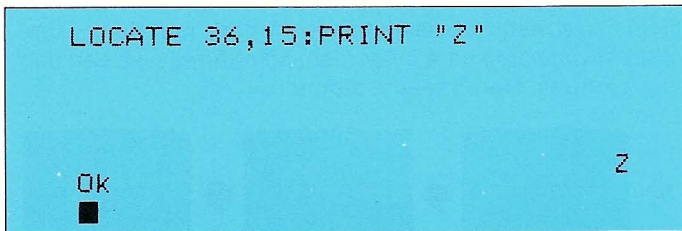
To move the  $\circ$  mark across the screen, we show it close to the left, then erase it, then show it a little bit further over, then erase it... and so on. To do one step in this series, we must tell the computer to erase, then to locate the correct position, then to print something.

```
CLS  
LOCATE 36,15:PRINT "Z"
```

There are two new commands, and then PRINT. CLS means "Clear Screen." If it is entered by itself, everything on the screen disappears.

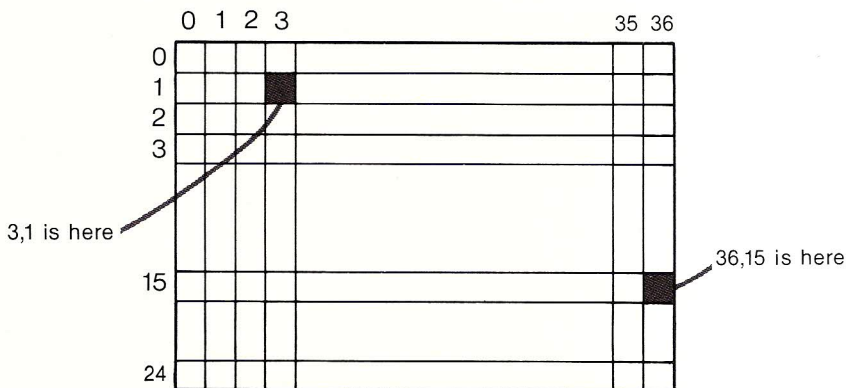


To see what the LOCATE command means, enter the above three commands together.



LOCATE 36, 15 tells cursor to go across 36 spaces, then down to line 15. The **colon** (:) tells the computer another command is on the same line. Then the PRINT command tells it to display Z at the place where the cursor is.

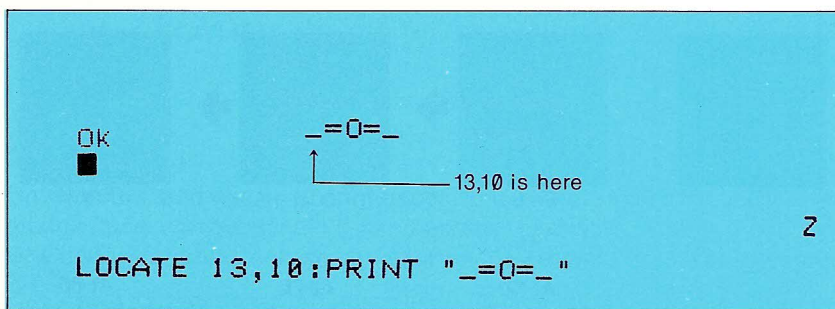
This is very much like the PSET command we used before to make twinkling stars, but with a difference: the character Z is bigger than a dot, and Z is a character while a dot is a graphic. For characters, then, we say LOCATE 36,15: PRINT "something"; and for graphics we say PSET (36,15) for example: the result is the same. In both cases, you can think of 36,15 as an **address**.



The PRINT command which follows the LOCATE command and the colon (: ) is not limited to a single character like "Z." **Any amount** of letters or words can be positioned with the LOCATE command. LOCATE just tells the cursor where to go before starting to PRINT. You can use LOCATE to start a paragraph in the middle of a page, to start a game at the bottom of the screen, or to position anything just where you want it.

Let's use a little imagination. Are you ready to see UFO? (UFOs, of course, are "unidentified flying objects"—spaceships from another planet—and they probably use computers.) We can make a UFO out of just five characters:

```
LOCATE 13,10:PRINT "_=O=_"
```



Does it look like a UFO to you? You can try drawing other shapes too, using the many different characters and signs on the keyboard. This UFO is made with the underline ( \_ ), the equal sign ( = ), and the letter O.



Now let's do something else:

```
LOCATE 13,10:PRINT "      "
```

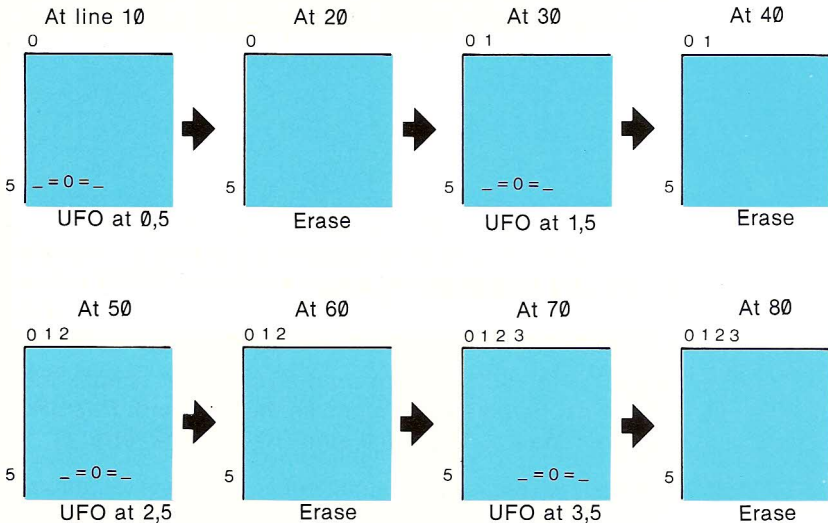
Press **RETURN**, and the UFO disappears! The five spaces replaced its five characters. The "space" is a useful way to erase letters or words when you don't want to use CLS to clear the whole screen.

## DISPLAY, ERASE, DISPLAY, ERASE..... \_\_\_\_\_

The LOCATE and CLS commands, then, are the way to make a moving picture. Here is a program to make our UFO move. (Don't type yet. Just think about how it would work.)

```
5 CLS
10 LOCATE 0,5:PRINT "__=0=_"
20 LOCATE 0,5:PRINT "      "
30 LOCATE 1,5:PRINT "__=0=_"
40 LOCATE 1,5:PRINT "      "
50 LOCATE 2,5:PRINT "__=0=_"
60 LOCATE 2,5:PRINT "      "
70 LOCATE 3,5:PRINT "__=0=_"
80 LOCATE 3,5:PRINT "      "
```

Here's what those commands would look like on the screen:



The first line, CLS, clears the screen, and after that each pair of lines—10 and 20, 30 and 40, 50 and 60, 70 and 80—puts the UFO at a different address. If we continue all the way to the right edge, like this:

	31	32	33	34	35	36	
							0
							1
							2
							3
							4
		_	=	0	=	_	5

the last commands will be

```

650 LOCATE 32,5:PRINT "_=0=_ "
660 LOCATE 32,5:PRINT "      "

```

The whole program is 67 lines long. Whew! It might be more fun just to watch television!

But don't leave. There is a much easier way. You have already noticed that the commands after line 5 all look very much alike, and that is the key. Actually, they are in nearly identical pairs.

```

10 LOCATE 0,5:PRINT "_=0=_ "
20 LOCATE 0,5:PRINT "      "

```

Only the first address number changes, and it varies systematically. That means (have you guessed?) it can become a **variable**. Now you can see how useful variables are: we can shorten the program to just seven lines.

```

100 CLS
110 I=0
120 LOCATE I,5:PRINT "_=0=_ "
130 LOCATE I,5:PRINT "      "
140 I=I+1
150 IF I<33 THEN GOTO 120
160 END

```

Clear the computer's memory with the NEW command, and then enter this program. (We've started the line numbers at 100 this time, so that we can add some more lines later.) Watch the UFO this time: it flies!

Let's go through the program line by line and see **why** it flies. Line 100 clears the screen for us. (We don't want our UFO to hit anything.) Line 110 names the variable and gives it a value. Since "I" means left-right position and we want to start at the left side, "I" is given the **initial value** of zero. Lines 120 and 130 are the display-erase pair that we saw before, but now they use the variable "I".

Line 140 changes the variable: it says  $I = I + 1$ . Remember this is not arithmetic, it's BASIC. Therefore "=" doesn't mean "equals", but means "takes the value of". "I" takes the value of "I+1" which makes "I" one number larger before we go to line 150.

Line 150 means if "I" is smaller than ( $<$ ) 33, then the computer will go back to line 120. Now "I" is one bigger, and the UFO will be displayed one space further to the right. When "I" is no longer smaller than 33, the computer will not "GOTO 120", but will move on to line 160. Actually, even without END the program would stop, because there would be no command after line 150.

But this program contains a **loop** where the computer goes around and around from 120 to 150 to 120 many times. It is controlled by the condition that "I" is less than 33, so it is called a **conditional loop**.

### A still better way to do it

Let's look at that program again.

```
100 CLS
110 I=0 ←————— The initial value is zero
120 LOCATE I,5:PRINT "_=0=_ "
130 LOCATE I,5:PRINT " "
140 I=I+1
150 IF I<33 THEN GOTO 120 ←————— The value goes up one
160 END                    with each loop;
                           the loop stops at 33
```

The most important parts of this program are lines 110, 140, and 150. They tell the computer where to begin (at 0), and to repeat the loop 33 times. This kind of repetition is used often in many programs, and BASIC has a special way to do it, with two commands called **FOR** and **NEXT**.



```

100 CLS
110 FOR I=0 TO 32 ← Repeat from 0 to 32
120 LOCATE I,5:PRINT "_=0=_ "
130 LOCATE I,5:PRINT "      "
140 NEXT I ← Increase the value of I by 1;
           return to line 120
150 END

```

These two commands, FOR in line 110 and NEXT in line 140, are the Loop Specialists. They use only two lines where we needed three before.

The important thing to remember is this form:

```

FOR variable = initial value TO final value
and
NEXT variable

```

For our program, it's

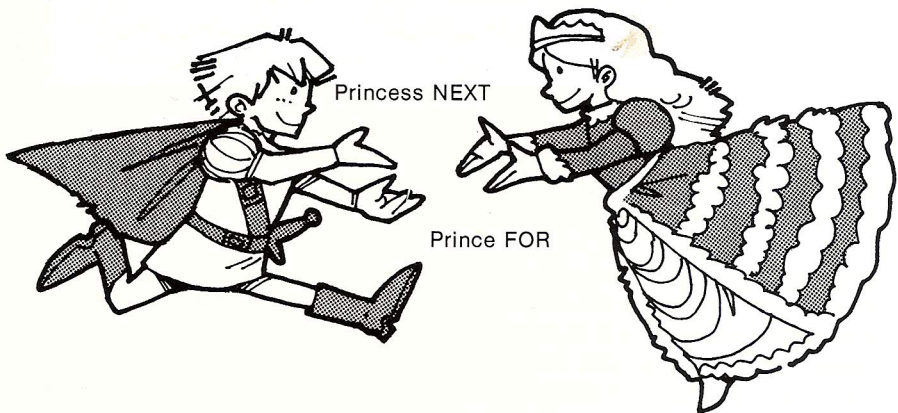
```

110 FOR I=0 TO 32
and
NEXT I

```

FOR tells the computer how many loops to make. NEXT is a very clever command, which counts the steps and sends the computer to the next line after the variable reaches the final value.

FOR and NEXT **always** go together as a pair.



If Prince FOR can't find Princess NEXT, the computer will be so unhappy that it will display an error message and stop until you correct it.

# ■ PUTTING IT ALL TOGETHER ■

How to...

- Add Color and Sound to Your Programs
- Put Two Programs Together
- Make a Flow Chart
- Improve Your Program



## ADDING COLOR AND SOUND \_\_\_\_\_

Now we're ready to add some functions we learned before—to get a colorful, noisy, moving UFO. Enter this program in your computer:

```
100 COLOR 15,1:CLS
103 Y=INT(RND(1)*22)
106 C=INT(RND(1)*14)+2:COLOR C
110 FOR I=0 TO 32
120 LOCATE I,Y:PRINT "_=0=_ "
130 LOCATE I,Y:PRINT " "
133 A=I MOD 12
136 IF A=0 THEN BEEP
140 NEXT I
150 GOTO 100
```

Repeated by FOR—NEXT

You can watch it as long as you like, because the whole program is a repeating loop, but let's push **CTRL** and **STOP** so we can see how the commands work.

Line 100 does two things: it sets the color (black background, white characters), and clears the screen. 103 makes a new variable, Y, and gives it a **random value** from 0 to 21. 106 is similar: it gives a random value, between 2 and 15, to a new variable C, and then makes that variable the color number. (We don't want color 1, because black characters would be invisible!)

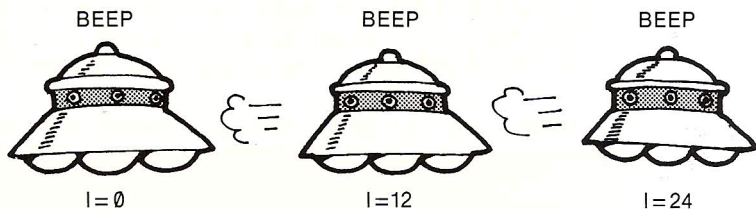
Starting at line 110 we have the FOR-NEXT sequence, and this time the variable Y is part of the "address" for each LOCATE command. The horizontal (left-right) location is I, which changes **regularly** from 0 to 32, but the vertical (up-down) location was made **random** (RND) in line 103: Y can be any value from 0 to 21. That's why those different-colored UFOs come in at different altitudes.

Now let's look at lines 133 and 136:

```
A=I MOD 12
IF A=0 THEN BEEP
```

There are two new words. BEEP, of course, means beep—the noise the UFO makes. But it beeps only if A is zero, and A has the value of I MOD 12, so what does MOD mean? MOD is short for **modulus**, and it means the number left over when one number is divided by another. I MOD 12 is the number left over when I is divided by 12. If I is 15 then I MOD 12 is 3. But if I is 24, I MOD 12 is 0.

As I increases from 0 to 32, I MOD 12 becomes 0 three times for each UFO: at 0, 12 and 24. That means each UFO gives three beeps on its way across the screen.





## GUESSING GAME+ SOUND+VIDEO

One of the most interesting things in computer programming is putting two programs together into one. Combining two or more program can be very useful and entertaining, as you will see here.

Here they are again:

Guessing program

```
10 A=INT(RND(1)*5)+1
20 INPUT "Make a guess";B
30 IF A=B THEN PRINT "Hit!" ELSE PRIN
T "Miss"
40 GOTO 10
```



UFO program

```
100 COLOR 15,1:CLS
103 Y=INT(RND(1)*22)
106 C=INT(RND(1)*14)+2:COLOR C
110 FOR I=0 TO 32
120 LOCATE I,Y:PRINT "_=0=_ "
130 LOCATE I,Y:PRINT " "
133 A=I MOD 12
136 IF A=0 THEN BEEP
140 NEXT I
150 GOTO 100
```

If we simply put the two programs into the computer the way they are, one after another, you can see that they won't fit together. The program will run as far as line 40, and then go to line 10 again, and the UFOs will never arrive. The first change we need, then, is to connect the UFOs to the first program, and the best place to connect it is to a correct guess. That's line 30 (A=B). Let's write a new line 30:

```
30 IF A=B THEN GOTO 100 ELSE PRINT "Miss"
```

Now there's another problem: how to stop the UFOs after somebody guesses the number. We don't want to watch the UFOs endlessly, and we don't want to stop the computer—we want to go back to the guessing game. So we change line 150 to:

```
150 GOTO 10
```

Now the program will look like this:

```
10 A=INT(RND(1)*5)+1
20 INPUT "Make a guess";B
30 IF A=B THEN GOTO 100 ELSE PRINT "M
iss"
40 GOTO 10
100 COLOR 15,1:CLS
103 Y=INT(RND(1)*22)
106 C=INT(RND(1)*14)+2:COLOR C
110 FOR I=0 TO 32
120 LOCATE I,Y:PRINT "_=0=_ "
130 LOCATE I,Y:PRINT "      "
133 A=I MOD 12
136 IF A=0 THEN BEEP
140 NEXT I
150 GOTO 10
```

↑ If Hit! go to UFO

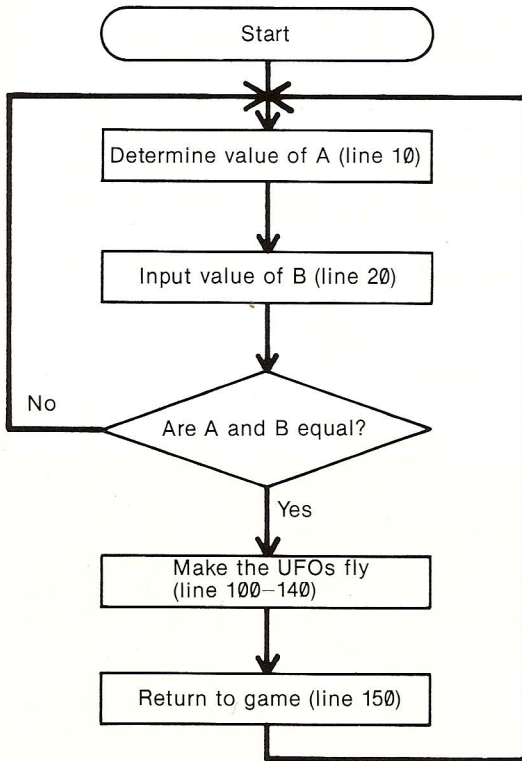
← Return to guessing

### The flow chart: a way to think about programs

Our programs are becoming longer and more complicated, with loops and multiple functions. When computer professionals make complex programs, they plan carefully before they start writing commands, and the best way to do that is with a flow chart.

This book doesn't use flow charts, and you don't need to understand them to use your Sony Computer. You can go on to the next section if you like.

But just to show you what a flow chart is, here's one for our guessing game/UFO program.



The top box is where the program starts, and each of the other boxes shows one operation which needs a command (or commands). The line shows the flow of action, or the logic of the program. Notice that the diamond-shaped box has two different lines running out of it; that shows that a condition (IF) is checked by the computer, to see which way to go. Notice also that the "No" line and the bottom line both make loops, to repeat the game.

### Improving the guessing game

The difference between an acceptable program (one that runs) and an **excellent** program is the improvements that are made after the program is written. Here are three things that we could make better in the guessing program.



- After each UFO, the next number is the same color as the UFO. If the numbers were all white, wouldn't they be easier to read?
- The guessing game moves up and down on the screen—always starting on the line below the last UFO. Can't we make it stay still?
- When the first UFO flies, the background color changes from dark blue to black, and stays black. wouldn't it be better to have one background color all the time?

With some simple additions and changes, we can solve these problems.

```

5 COLOR 15,1
7 CLS:LOCATE 0,0 ← Add
10 A=INT(RND(1)*5)+1
20 INPUT "Make a guess";B
30 IF A=B THEN GOTO 100 ELSE PRINT "M
iss"
40 GOTO 10
100 CLS ← Change
103 Y=INT(RND(1)*22)
106 C=INT(RND(1)*14)+2:COLOR C
110 FOR I=0 TO 32
120 LOCATE I,Y:PRINT "_=0=_ "
130 LOCATE I,Y:PRINT "      "
133 A=I MOD 12
136 IF A=0 THEN BEEP
140 NEXT I
150 GOTO 5 ← Change

```

Before you read further, see if you can understand these changes by yourself...

Did you get it right? The new lines 5 and 7 solve all of the problems quite nicely. Line 5 commands a black background and white letters at the start. Line 7 keeps the game at the top of the screen.

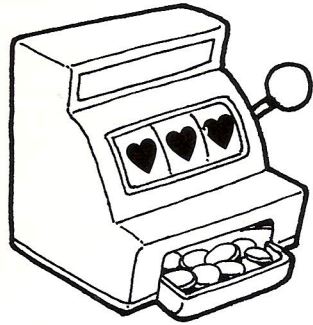
But the most important thing to remember about changing a program is this: **a change in one place often means changes in other places.** In this case, the COLOR command in line 5 came from line 100, so line 100 must be changed, to just CLS. And since the program now starts at line 5 instead of line 10, line 150 must be changed, to make the two new lines run.

Now we have a good, smooth program. Are you completely satisfied with it? Or would you like to make it more interesting? .... Why do the UFOs always fly in straight lines? .... Your Sony Computer can do many, many tricks—whatever you tell it to do.

# THE SLOT MACHINE GAME

How to...

- Program a Slot Machine Game that...
  - Changes the Display until You Stop It
  - Compares the Slot Displays
  - Keeps the Score According to Your Bets
- Use an Array Variable
- Use the ON-GOTO Command
- Use a String Variable
- Stop a Program by Pressing a Key
- Set a Format for Printing
- Use More than One Condition in an IF—THEN Command
- LIST a Long Program in Sections



When you correctly guessed the number in our last program, the reward was a flying UFO. Now your Sony Computer will become a Las Vegas-style "slot machine." Each time you play, you will win or lose! The computer cannot handle money, like the real Las Vegas slot machines, but it is very good at keeping score with points.



Here are the rules: first, the computer will ask you how many points you want to bet. Next it will rapidly change the symbols displayed in its three "slots," until you stop it. Then it will tell you your new score: if all three symbols are the same, you win three times the number you bet. If only two are the same, you win the same number you bet. But if all three are different, you lose two times your bet. You start with a **stake** of 100 points, and if you reach 300, you win the game. But if your score goes down to zero, you lose and the game is over.

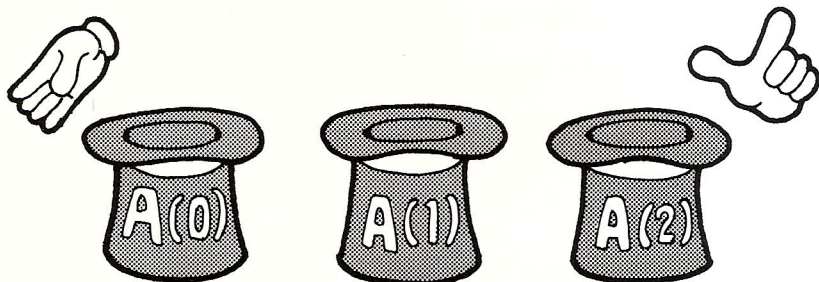
Does that sound OK? Let's program it!

The main part of this game is the three slots. In each one, four symbols (♥, ♠, ♦, ♣) are continually appearing and changing—so quickly that you can't see which is there, you can only guess. Naturally, our program will use variables for the changing symbols in each slot.

## WHAT IS AN ARRAY VARIABLE? \_\_\_\_\_

We have programmed variables several times, so we understand how a letter or name can have a changing value. We used names like A, B and Q, and we gave them values which varied with the number of times a loop was repeated, or the position of a UFO, or at random.

This time, we need a variable that can become any of four symbols. The three slots each show the same set of symbols, which means we can use the same variable for all of them. But each slot functions alone: sometimes they will match, often they won't. For this situation, we will use our variable in three different places—three of the same variable. Each is called A, but also has a number (in brackets), so that we know which variable is for each slot.



A (0), A (1) and A(2) are **array variables**. Does it seem confusing? Do you think it would be easier to use three different names? Soon you will see that array variables make programming much simpler, because they are very flexible.

Our array of variables is three variables **wide**, and is as **long** as the number of different values we give it. You can think of them as **multi-dimensional** variables, and this will help you remember the BASIC command for making an array variable: **DIM**. In this program we will use the command line.

```
10 DIM A(2)
```



to make three array variables A (0), A (1) and A(2). (Other examples of array variable commands are: **DIM A (10)**, which makes 11 variables from A (0) to A (10); and **DIM A (2), B (2)**, which makes six variables—A (0) to A(2) and B (0) to B(2).)

There is one more thing to understand about array variables: the number in brackets can also become a variable, such as A (I), which makes them even easier to program.

## MAKING A SLOT MACHINE

We want our array variable to represent different symbols—hearts (♥), spades (♠), diamonds (♦) and clubs (♣). The first thing we need is number codes for the symbols. You remember that we use number codes to represent colors, and your Sony Computer already knows the codes for the different colors. This time, we must decide our own code for the symbols.

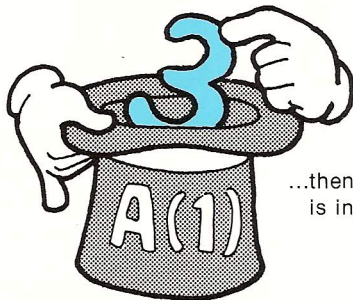
♥.....1      ♠.....2  
♦.....3      ♣.....4

Now we know that when variable A (1) takes the value of 3, for example, then a diamond is in that slot.

First decide the code

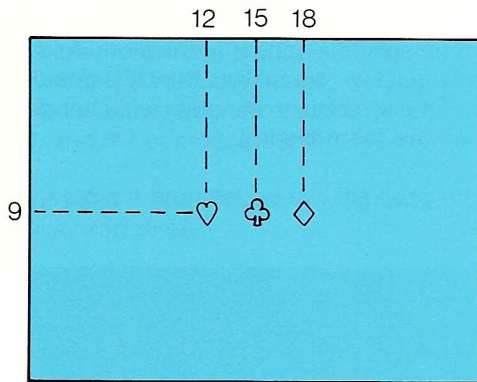


If this happens...



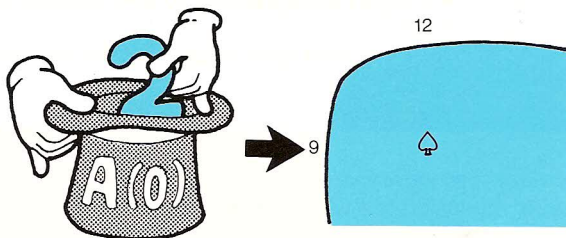
...then a diamond (♦)  
is in slot A (1)

Next we must decide where the slots will be on the screen. Let's put them in the middle.

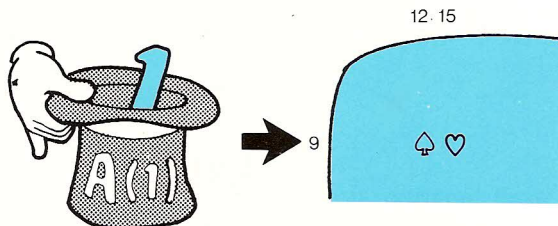


We have placed the A (0) slot at 12,9. The A (1) slot is at 15,9. And the A (2) slot is at 18,9.

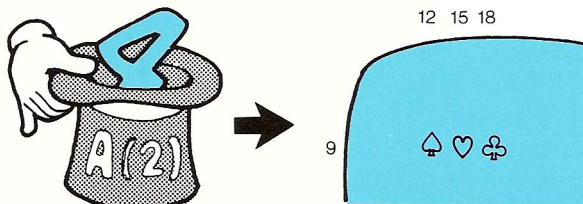
Our planning for the program to make a “slot machine” is now finished. Let’s review the system that we have designed.



If A (0) takes the value 2, 2 is the code for ♠, So ♠ appears at 12,9.



If A (1) takes the value 1, 1 is the code for ♥, So ♥ appears at 15,9.



If A (2) takes the value 4, 4 is the code for ♣, So ♣ appears at 18,9.

This system makes the symbols appear in the slots. And since each variable will change very quickly—as fast as the UFO positions changed in our earlier program—the symbols in the slots will change so quickly that you won't be able to see them clearly.

We have planned our program, we understand the plan, and we're ready for the commands.

```
10 DIM A(2)
20 CLS
30 FOR I=0 TO 2
40 A(I)=INT(RND(1)*4)+1
50 LOCATE I*3+12,9
60 ON A(I) GOTO 70,80,90,100
70 PRINT "♥":GOTO 110
80 PRINT "♠":GOTO 110
90 PRINT "♦":GOTO 110
100 PRINT "♣"
110 NEXT I
120 GOTO 30
```

Line 10 creates our three array variables, as we explained in the last section. Whenever array variables will be used, we must tell the computer at the **beginning** of the program.

```
10 DIM A(2)
```





Line 20, of course, clears the screen. Line 30 begins with FOR, and we know that when we see a FOR command, we must also have a NEXT command somewhere in the program. The FOR-NEXT section goes from line 30 to line 110, and the commands in this section will be repeated in a loop.

After FOR, line 30 makes a new variable I, which we will use between the brackets in our array variables.

Does line 40 look familiar to you? It is the same kind of **random number** command that we used in the guessing game. When I takes the **initial value** of 0, line 40 will become

```
A (0)=INT (RND (1) * 4)+1
```

This means that A (0) can be any number between 1 and 4, right? A (0) is the left slot, and the numbers 1-2-3-4 are codes for the symbols ♥, ♠, ♦, ♣. In other words, when I takes the value of 0, line 40 tells the computer to choose one of the symbols for the left slot.

Next we must tell the computer where to display the three slots on the screen.

```
50 LOCATE I * 3+12,9
```

When I has the value 0, for the left slot, the cursor is at 12,9. When I has the value of 1, for the center slot, the location will become 15,9. For the right slot, I will be 2 and the location will be  $2 \times 3 + 12,9$ —or 18,9.

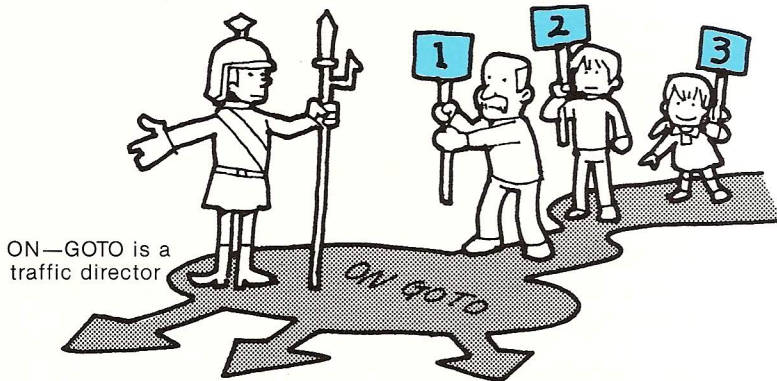
Now the computer has chosen a symbol, and the cursor has moved to the slot. But the computer will not display the symbol, until we tell it to. For this, line 60 is very important.

```
60 ON A(I) GOTO 70,80,90,100
```

Here we see the GOTO command for four different lines at the same time—70, 80, 90, 100. How will the computer know which line to go to? the ON A (I) part of this command makes the decision. This is the form that is always used for an ON-GOTO command:

```
ON something GOTO line 1 , line 2 , line 3 ...
```

This means, for example: if something is 1, go to line 1; if something is 2 go to line 2; if something is 3 go to line 3 ... In other words, the value of "something" decides automatically which line the computer will go to.



ON—GOTO is a traffic director

As an example, let's say that A (0) has the value 3 (the code for ♦). Line 60 tells the computer to go to line 90 when the variable has this value.

```
90 PRINT "♦":GOTO 110
```

What happens if 2 is the value for variable A (0)? Then line 60 sends the computer to

```
80 PRINT "♠":GOTO 110
```

Now you understand how the computer chooses one of the four symbols and displays it in a slot.

After each of the PRINT commands, the computer goes on to line 110. (There is no GOTO 110 command in line 100, because the computer automatically goes to the next line.)

Line 110 tells the computer to change the value of I from 0 to 1. Next it goes back to line 40. Now, do you know what will happen next? Think about the program, and try to understand it by yourself before we explain it.

OK. When I is 1, line 50 becomes

```
LOCATE 1 * 3 + 12, 9
```

which means LOCATE 15,9. In other words, the position where the symbols will be displayed has moved from 12,9 to 15,9—from the left slot to the center slot. One of the symbols will appear on the screen at 15,9. Then the computer will change I to 2, and come back to line 40. And this time the position becomes 18,9—the right slot. The computer has gone through the program three times (in much less than one second!).



That is what you might see on the screen, but we can never know which symbols the computer will choose at random.

Lines 30 to 120 are a loop. Your computer will go around and around, using the array variables to choose and display symbols in the left slot, center slot, right slot, left, center, right... Since the program is an endless loop, the symbols will appear to be constantly “revolving” on the screen, just as they would in a real slot machine.

If you haven't tried the program yet, enter it now on your computer. Check to be sure that there are no mistakes. When you give the RUN command, you can see that we have succeeded in creating a slot machine by using array variables. (In the SCREEN 0 condition, however, the right edge of each symbol is not displayed.)

But how can we make this endless loop stop, to tell us the score? And how will the computer know the score? To turn our slot machine into a game, we have to put some more commands in the program. When we do that, we will use some variables that take letters instead of numerical values—called **String Variables**. And that requires a special explanation.

## STRING VARIABLES

---

We have learned that a variable can take the value of any **number**—such as 1, 2, 3, 191 or 252. Variables can **also** be given the “value” of any **letter**—or of a set of letters that makes up a **string**.

Let's do some simple exercises to see how a variable is used to mean a letter or string. First, type **PRINT A\$** and push **RETURN**. (\$ is on the 4 key, and requires the **SHIFT** .)

```
PRINT A$
```

```
OK
```



Nothing happened. There was no beep or error message to indicate a mistake, and the computer didn't print anything. Now enter this command:



```
A$="ABC"  
PRINT A$
```

Press **RETURN**, and your screen should look like this:

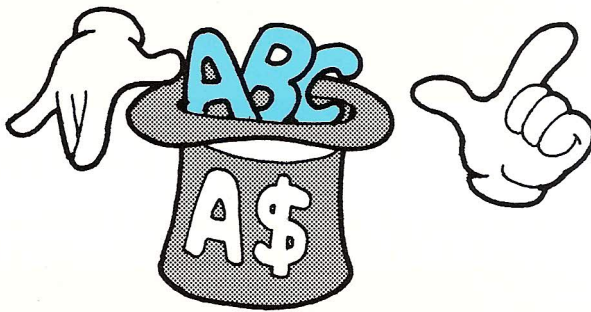
```
A$="ABC"  
OK  
PRINT A$  
ABC  
OK  
■
```

You can see that A\$ is being used as a variable, and its value is "ABC." Compare this with another command:

```
A=345  
OK  
PRINT A  
345  
OK  
■
```

This looks familiar, doesn't it? Variable A takes the value of the number 345, and there is no \$ sign (dollar sign) at the end of the variable name A.

Have you guessed the rule? **When the variable name ends with a \$ sign, the variable has the value of a letter or a set of letters (called a "string.") Without the \$ sign at the end, the variable takes the value of a number.**



There is one more thing to remember: the value must be in **quotation marks**, like this:

```
A$="ABC"
```

Try these examples, and you will clearly understand the difference between numerical and string variables.

A=123	A takes the value of 123
OK	
B=234	B takes the value of 234
OK	
PRINT A+B	Display A+B
357	The sum is 357
OK	
A\$="123"	A\$ takes the value of "123"
OK	
B\$="234"	B\$ takes the value of "234"
OK	
PRINT A\$+B\$	Display A\$+B\$
123234	The sum is "123234"
OK	
A\$=987	A\$ takes the value of 987
Type mismatch	Beep! Not possible
OK	
■	

When we enter `A$="123"`, the value of variable `A$` becomes 123—not the number one-hundred-twenty-three, but the **characters** one-two-three. And in the next line, `B$` takes the value of 234 (characters two-three-four). Why? Because we used quotation marks and the \$ sign. That told the computer that we are using a string variable, so it reads 123 not as a number, but as characters, as if they were letters.

When the computer adds `A$+B$`, then, it connects them together into a single set of characters. A similar example would be `"ABC"+"BCD"="ABCBCD"`.

Finally we entered `A$=987`. The computer responded with an error message:

```
Type mismatch (you typed two things that do not match)
```

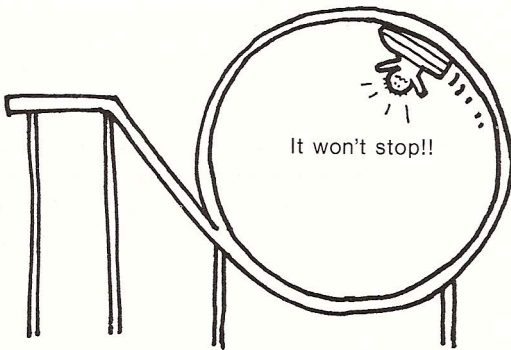
Without quotation marks, 987 is a number (nine-hundred-eighty-seven) which **does not match** the string variable with the \$ sign.



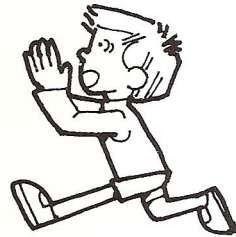
Now we know that when the variable name ends in a \$ sign, it is a string variable, that should be assigned a letter or a set of letters, and the value must be in "quotation marks". Now, let's go back to our program for the slot machine game.

## HOW TO STOP A LOOP WITH INKEY \_\_\_\_\_

The FOR-NEXT loop that we used in the guessing game stopped itself when its variable reached the last value. But command line 120—the GOTO 30 command—means that our loop in this slot machine game will never stop.



Don't worry, I'll help you...



Of course, you can always stop your computer by pushing **CTRL** and **STOP**. But that stops the whole program, and if you give the RUN command to start again, we still have an endless loop. We want to stop the slot machine and look at the symbols, to see if they match and learn how many points we win or lose. And then we want the program to start again, so that the game continues.

A good way to stop the slot machine is by adding these two lines to the program:



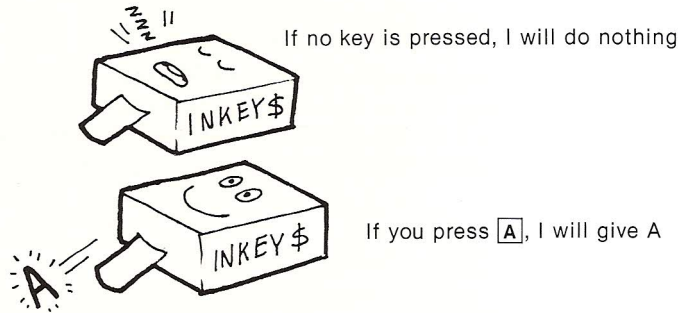
```

103 K$=INKEY$
106 IF K$="A" THEN END

```

K\$ is a string variable, of course. And we can see in line 106 that when the value of K\$ is A, the slot machine will END its revolving display.

What does **INKEY\$** mean? It is a function in the BASIC language. INKEY means **input from the keyboard**—any key that you press. Line 103, then, means that **K\$ takes the value of any letter key that you press**. If you don't press a key, the computer will advance without giving a value to K\$. If you press **B** or **C** or **X**, the computer will continue to revolve the slot machine. But if you press **A** (in capital letter), line 106 will command the computer to END.



Now our FOR-NEXT and GOTO 30 loop, which goes from line 30 to line 120, has an END command in the middle. But the computer will not hear the command (will not stop the slot machine) unless we tell it to, by pressing the **A** key. In other words, the slot machine keeps changing the symbols until we stop it.

```

30 FOR I=0 TO 2

```

```

103 K$=INKEY$
106 IF K$="A" THEN END
110 NEXT I
120 GOTO 30

```

FOR—NEXT loop that displays 3 symbols in the slots

Program will END if **A** key is pressed

## WHAT'S THE SCORE? \_\_\_\_\_

When the slot machine stops changing the symbols, we want to know how many points we won (or lost!). This means we must put the rules in the program. Do you remember the rules? We begin with a **stake** of 100 points. Then we **bet** some of our points—any number between 1 and 100. Let's add this information to the program.

```
23 P=100;LOCATE 2,18:PRINT USING "STAKE:####";P
26 LOCATE 4,20:INPUT "BET";B
```

Let's look first at line 23. Our stake will change each time we bet, so we use a variable, P. The stake P has an initial value of 100. To display the score, we give a LOCATE command, and then a PRINT command.

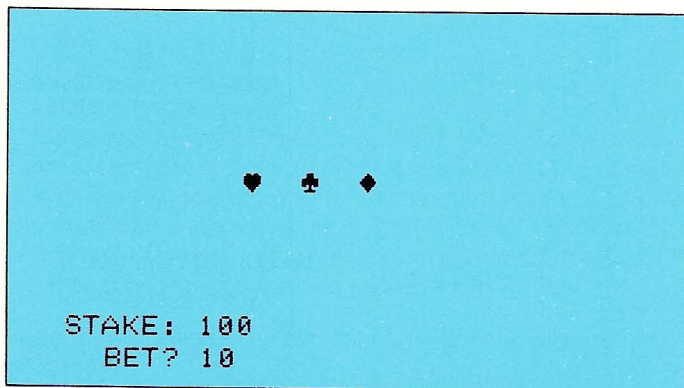
```
PRINT USING "STAKE:####";P
```

This new PRINT command tells the computer how to **format** the information it is displaying. The # sign means "digits". This command tells the computer: display P by using the four empty spaces on the screen next to "STAKE:". The game starts with a stake of 100, so the first display at 2,18 will be:

```
STAKE: 100
```

4 digits

Line 26 tells the computer to input your bet on the screen, just below the stake. Your bet will change every time, so this line uses variable B for your bet. The INPUT command tells the computer to wait for the value of B—the bet—to be entered from the keyboard. If you bet 10 points, it will be displayed like this:



After lines 23 and 26, the computer advances to the FOR-NEXT section. Now let's enter the new command lines 23, 26, 103 and 106 in our program.

```
10 DIM A(2)
20 CLS
23 P=100:LOCATE 2,18:PRINT USING "STA
KE:####";P
26 LOCATE 4,20:INPUT "BET";B
30 FOR I=0 TO 2
40 A(I)=INT(RND(1)*4)+1
50 LOCATE I*3+12,9
60 ON A(I) GOTO 70,80,90,100
70 PRINT "♥":GOTO 103
80 PRINT "♠":GOTO 103 ← Change
90 PRINT "♦":GOTO 103
100 PRINT "♣"
103 K$=INKEY$
106 IF K$="A" THEN END ← Add
110 NEXT I
120 GOTO 30
```

Notice that we must change the GOTO 110 command in lines 70, 80 and 90 to GOTO 103. Do you know what would happen if we forgot that?

Now the slot machine is ready to take your bet, but we have not yet told the computer how to change your stake, that is, how to keep the score. We will add some new commands to the end of the program beginning with line 130. But first, go back to line 106.

```
106 IF K$="A" THEN END
```

When we make the slot machine stop changing the symbols, by pressing the **A** key, we don't actually want the program to END. Instead, we want the computer to tell us the score. So change line 106 to this command:

```
106 IF K$="A" THEN GOTO 130
```

We will know whether we have won or lost when we see the symbols on the screen. If all three are the same, then we will win three times our bet. (If our stake is 100, and we bet 10, and we see three of the same symbol, then our new score is 130.) If two of the symbols are the same, we win the same amount that we bet (the new score is 110). If we see three different symbols, we lose two times our bet (the new score is 80). For example, when we see

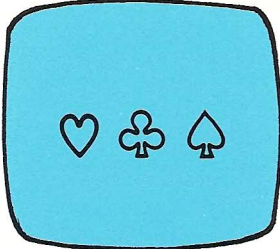




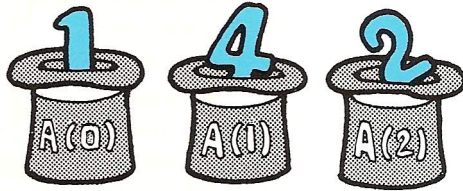
we know that we have lost some points.

But your computer is not looking at the symbols on the screen. It uses the variables that are in its memory. The computer will know very quickly if  $A(0)$ ,  $A(1)$  and  $A(2)$  are equal to each other, by checking the values of those array variables.

We see this



The computer's memory has this



If the screen shows:



do you know what will be in the computer's memory?

As you can see, it is not very difficult for the computer to decide whether two, three or none of the symbols are the same. And that means that our program commands for scoring are also not very difficult.

```
130 IF A(0)=A(1) AND A(0)=A(2) THEN P
=P+B*3:GOTO 150
140 IF A(0)=A(1) OR A(1)=A(2) OR A(0)
=A(2) THEN P=P+B ELSE P=P-B*2
150 LOCATE 8,18:PRINT USING "####";P
160 GOTO 26
```

The first line of these new scoring commands is 130. This means that the scoring part of the program will be used only after the **A** key is pressed and the slot machine stops.

```
130 IF A(0)=A(1) AND A(0)=A(2) THEN P
=P+B*3:GOTO 150
```

Notice that the IF—THEN command has **two** conditions, connected with **AND**. If they are **both** true— $A(0)=A(1)$  and  $A(0)=A(2)$ —then all three array variables are equal, and the three symbols on the screen are the same. If this is true

```
THEN P=P+B * 3:GOTO 150
```

That means your bet will be multiplied by 3, and added to the stake, P; and the computer will go to line 150. If you bet 10 with a stake of 100, the new score is 130.

But if the three array variables are not the same, the computer will not follow the THEN command. Instead it will go to the next line in the program.

```
140 IF A(0)=A(1) OR A(1)=A(2) OR A(0)
    =A(2) THEN P=P+B ELSE P=P-B*2
```

Here we have an IF command with three conditions, and they are connected by **OR**. This means that if any **one** pair of variables are equal, (or, two symbols are the same),

```
THEN P=P+B
```

adds your bet to your stake, for a new score of 110. But if none of the three conditions is true (all the symbols are different), the computer will go on to

```
ELSE P=P-B*2
```

You lose! Your bet is multiplied by two and subtracted from your stake—the new score is only 80.

Now the computer knows the score, and the next step is to display it.

```
150 LOCATE 8,18:PRINT USING "####";P
```

This is easy to understand, because it is almost the same as the PRINT-format command in line 23: first LOCATE, then PRINT USING the format of four digits, then the name of the variable that is taking a new value. Each time you stop the slot machine, the computer will calculate the new score and display it at position 8,18.

What happens next? The game continues, which means you must make another bet. So line 160 sends the computer back to line 26, where it waits for you to enter your bet on the keyboard.

Now enter the new scoring commands in the program. It should look like this:

```

10 DIM A(2)
20 CLS
23 P=100:LOCATE 2,18:PRINT USING "STA
KE:####";P
26 LOCATE 4,20:INPUT "BET";B
30 FOR I=0 TO 2
40 A(I)=INT(RND(1)*4)+1
50 LOCATE I*3+12,9
60 ON A(I) GOTO 70,80,90,100
70 PRINT "♥":GOTO 103
80 PRINT "♠":GOTO 103
90 PRINT "♦":GOTO 103
100 PRINT "♣"
103 K$=INKEY$
106 IF K$="A" THEN GOTO 130 ← Change
110 NEXT I
120 GOTO 30 ← Add
130 IF A(0)=A(1) AND A(0)=A(2) THEN P
=P+B*3:GOTO 150
140 IF A(0)=A(1) OR A(1)=A(2) OR A(0)
=A(2) THEN P=P+B ELSE P=P-B*2
150 LOCATE 8,18:PRINT USING "####";P
160 GOTO 26

```

Wait a minute! This program is too long to fit on the screen. The first part disappears as you enter the LIST command. To solve this simple problem, we must tell the computer **what** to list, like this:

```
LIST 10-100
```

With this command, lines 10 to 100 are displayed. To see the last part, enter

```
LIST 110-
```

Now use the LIST commands to check carefully for mistakes. When you are sure the program is correct, give the RUN command—and enjoy your new slot machine game!



## THE FINISHING TOUCHES

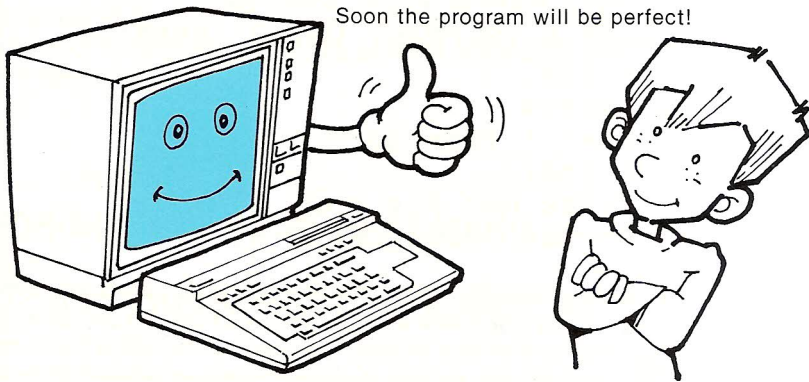
---

Your Sony Computer has become a slot machine that works very smoothly. It is doing some very complex tricks: asking for a bet, changing the symbols in three slots, and calculating the score when you stop the symbols. Our program for all these things has only 20 command lines—not very many when you remember that a computer must be told **each** time to display, calculate, stop, wait for a bet, etc.

But one thing is missing. In the beginning, we planned that you would win the game when your score reaches 300, or lose the game when your score becomes 0. By now, you have probably run your slot machine score over 300, or below 0, or perhaps both.

Let's make our good program into a better program, by adding commands for winning and losing. At the same time, there are some other improvements we can make, for a **perfect** program.

- We can expand our game a bit, by making it possible to enter another, different bet—only after pressing the space bar on the keyboard.
- The space bar would also be more convenient than the **A** key when we stop the game, because the space bar is easier to press.
- When we make a new bet, the previous bet is still on the screen. This is a little confusing.



Here are the changes that we should enter in our program.

```

10 DIM A(2)
20 CLS
23 P=100:LOCATE 2,18:PRINT USING "STA
KE:####";P
24 LOCATE 7,20:PRINT "          " ←—— Add
26 LOCATE 4,20:INPUT "BET";B
30 FOR I=0 TO 2
40 A(I)=INT(RND(1)*4)+1
50 LOCATE I*3+12,9
60 ON A(I) GOTO 70,80,90,100
70 PRINT "♥":GOTO 103
80 PRINT "♠":GOTO 103
90 PRINT "♦":GOTO 103
100 PRINT "♣"
103 K#=INKEY#
106 IF K#=" " THEN GOTO 130
110 NEXT I ↑—— Change
120 GOTO 30
130 IF A(0)=A(1) AND A(0)=A(2) THEN P
=P+B*3:GOTO 150
140 IF A(0)=A(1) OR A(1)=A(2) OR A(0)
=A(2) THEN P=P+B ELSE P=P-B*2
150 LOCATE 8,18:PRINT USING "####";P
152 GOTO 170
154 K#=INKEY# ←—— Add
156 IF K#=" " THEN GOTO 24
160 GOTO 154 ←—— Change
170 IF P<300 AND P>0 THEN GOTO 154
180 IF P>=300 THEN LOCATE 3,5:PRINT "
YOU WIN!"
190 IF P<=0 THEN LOCATE 3,5:PRINT "YO
U LOSE!"
200 END

```

↑—— Add

Do you understand? Before you read the explanation, take a few minutes to see if you understand it by yourself.

Line 170 begins a new set of winning and losing commands. If the score is less than 300 AND more than 0, the game continues, and the computer goes back to the program. If the score is 300 or more, the game is over, and YOU WIN! is displayed near the top of the screen. If the score is less than 0—YOU LOSE! And if you win or lose, the computer goes to line 200 and the game reaches the END.

Line 24 erases the previous bet at the beginning of each round of the game. It moves the cursor to the bet display, and types six empty spaces there instead.

Line 106 has a very simple change. Now we stop the slot machine with the space bar, not with the **A** key. Remember that your computer reads a **space** (" ") as a **character**.

Lines 160, 154 and 156 mean that the game will start again when we press the space bar.

Those are the four changes we wanted to make: winning/losing, and the three small improvements.

Line 152 does not actually change the game, but it is required because of the other changes in the program. This line comes just after the new score is displayed, and it sends the computer ahead to the new winning/losing section in line 170. If you have not won or lost, line 170 returns the program to line 154, and the game continues when you press the space bar.

## FOR EASIER PROGRAM READING

---

When we began writing the Slot Machine program, we used line numbers in tens: 10, 20, 30, 40... Then, as we made the program more complete, we used some of the "empty" lines to add new commands. As a result, our line numbers came in unusual groups, like 90, 100, 103, 106, 110, 120. To make the program easier to read, we can **renumber** the lines.

Enter this command:

```
RENUM
```

The lines stay in the same order, but the line numbers change to tens, as we can see when we give the LIST command.



Now the line numbers are much easier to read. But what has happened to our GOTO commands, which contained line numbers? No problem! For example, look at line 100 in the new program:

```
100 PRINT "♥":GOTO 140
```

This GOTO command—GOTO 140—was GOTO 103 before you gave the RENUM command. Now line 103 has become line 140, and GOTO 103 is GOTO 140. That means your Sony Computer is quite smart, and when you run the Slot Machine program again, the game will not be changed at all.

## PUTTING TITLES IN THE PROGRAM

---

Suppose you want to show your program to a friend. Or suppose you want to read a program again several months after you wrote it. When 20 or 30 lines of BASIC come on the screen, it may be difficult to understand their functions, because the reader is not familiar with the program. To solve this problem, BASIC lets you put "remarks" in the middle of your program, like this:

```
5 REM *** SLOT MACHINE GAME ***
7 REM
10 DIM A(2)
20 CLS
30 P=100:LOCATE 2,18:PRINT USING "STA
KE:####";P
40 LOCATE 7,20:PRINT "      "
50 LOCATE 4,20:INPUT "BET";B
55 /
60 / *** SLOT MACHINE START ***
62 /
65 FOR I=0 TO 2          — Previous line 60
70 A(I)=INT(RND(1)*4)+1
80 LOCATE I*3+12,9
90 ON A(I) GOTO 100,110,120,130
100 PRINT "♥":GOTO 140
110 PRINT "♠":GOTO 140
120 PRINT "♦":GOTO 140
130 PRINT "♣"
140 K#=INKEY#
150 IF K#=" " THEN GOTO 180
```

```

160 NEXT I
170 GOTO 60
175 /
180 / *** CALCULATE STAKE ***
182 /
185 IF A(0)=A(1) AND A(0)=A(2) THEN P
=P+B*3:GOTO 200 — Previous line 180
190 IF A(0)=A(1) OR A(1)=A(2) OR A(0)
=A(2) THEN P=P+B ELSE P=P-B*2
200 LOCATE 8,18:PRINT USING "####";P
210 GOTO 250
220 K#=INKEY#
230 IF K#=" " THEN GOTO 40
240 GOTO 220
245 /
250 / *** WIN OR LOSE *** Previous line 250
252 /
255 IF P<300 AND P>0 THEN GOT O 220 ←
260 IF P>=300 THEN LOCATE 3,5:PRINT "
YOU WIN!"
270 IF P<=0 THEN LOCATE 3,5:PRINT "YO
U LOSE!"
280 END

```

Our "remarks" here are the **title** of the program and **subtitles** for its different parts. They make the program longer, but much easier to read and think about. And they don't change the game at all, because your computer doesn't read the new lines—they are only for the convenience of people who use the program.

Let's look at some of the new remark lines:

```

5 REM *** SLOT MACHINE GAME ***
7 REM

```

The REM command tells the computer to **ignore** the characters after the command itself. You can type in whatever you want, and it will not change the program. On line 5, we see the title of this program. Line 7, though, is empty after the REM command. This way there is empty space

around the title—just like on the title page of a book. The empty space, and the stars before and after the title, make it easy to find the title among all those lines of BASIC commands.

Now look at lines 55, 60 and 62. You can see that there is a shorter way to give the REM command: with the ' (single quotation mark). Of course, these lines are not actually part of the program.

The computer will not read them as commands, because they begin with '. The subtitle, the stars and the empty space are for people, not the computer.

The computer **does** see the line numbers, because they come before the ' (or REM). For example, in line 150, we see:

```
150 IF K#=" " THEN GOTO 180
```

But 180 is a remark line:

```
180 ' *** CALCULATE STAKE ***
```

The computer finds no command in line 180, so it goes on to the next one. Line 182 has no command, either, so the computer goes on to 185. But the **reader**—you or your friend—goes from line 150 to line 180, and knows immediately that this is the section where the stake is calculated.



## YOU'VE COME A LONG WAY! \_\_\_\_\_

Congratulations! After doing the exercises and games in this book, you have a working knowledge of a new language—BASIC. Perhaps it was just a few hours, or one or two days ago that you began with very simple commands, like PRINT 3+5. And now you can understand and **use** complex lines like `A(I)=INT(RND (1) * 4)+1`, and `PRINT USING "# # # #";P`. Your Sony Computer will give you years of service, entertainment and education, now that you know how to "speak its language."

As you practice writing and using programs, you will become more and more skillful with numbers, graphics, commands, functions and games. It's just a matter of time until you are an **expert**, and the time will go quickly as you share the fun of your Sony Computer with your family and friends.

You can continue practicing commands, and learn some new ones, in the next section of this book. After that, you can use the **MSX-BASIC Programming Reference Manual** to try out many new BASIC commands, and discover what they do. And then you can experiment with your computer, to make new colors, graphics, sounds and games.

When you want to create **your own program**, first spend a few minutes planning. Perhaps you will want to make a flow chart. When you know what **functions** and what types of **commands** you will need, use the Index at the back of the book to find them and review the explanations. Plan carefully, and then write some commands and run them. Your Sony Computer will help you in two ways: it always does **exactly** what you tell it to do; and it always forgets your mistakes when you enter the NEW command.

Talk about your programs with your friends. Show them what you can do on your computer. They will probably have some different ideas, and together you can have more fun. If your friends have their own computers, you can trade programs with them, by using a tape recorder or by writing your program on paper.

Remember, the best way to have fun with your Sony Computer is to try new things, and see what happens. There is no limit to what you can tell the computer with the BASIC language. Are you ready to begin the wonderful voyage into your imagination? Good luck!


# BASIC COMMAND PRACTICE

## PRINT \_\_\_\_\_

```
10 PRINT "JOHN"  
20 PRINT "AND"  
30 PRINT "MARY"  
RUN  
JOHN  
AND  
MARY
```

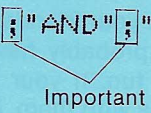
Do you remember this simple program? Let's change it a little.

```
10 PRINT "JOHN";  
20 PRINT "AND";  
30 PRINT "MARY"  
RUN  
JOHNANDMARY
```



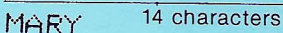
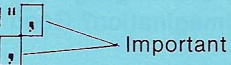
With the ; (semi-colon) after "JOHN" and after "AND", the computer displays the three words together, on one line. Actually, the commands can all be put in just one program line:

```
10 PRINT "JOHN";"AND";"MARY"  
RUN  
JOHNANDMARY
```



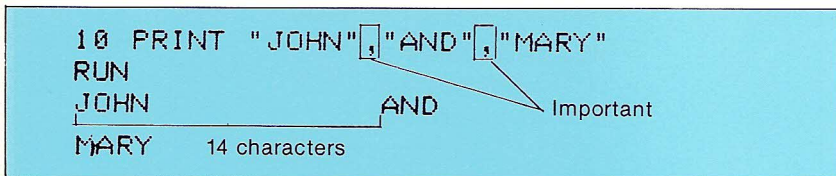
But it's difficult to read three words without spaces between them, so let's try something else.

```
10 PRINT "JOHN";  
20 PRINT "AND";  
30 PRINT "MARY"  
RUN  
JOHN          AND  
MARY          14 characters
```



When we use a , (comma) instead of a ; the first letters of each word are printed 14 spaces apart. The same thing will happen if you enter the command on one line.

```
10 PRINT "JOHN","AND","MARY"
RUN
JOHN           AND           MARY
14 characters
```



Now try this command, with a single space between each word and the second quotation mark.

```
10 PRINT "JOHN "; "AND "; "MARY"
RUN
JOHN AND MARY
```

Remember: the **space** is an important character.

Now let's review the different ways that number characters can be printed.

```
10 PRINT "3+5="; ← Display characters
20 PRINT 3+5     ← Make calculation, display answer
RUN
3+5= 8
```

Line 10 is a command to display characters, and line 20 is a command to calculate. But these two different functions are **connected** by the ; at the end of line 10. That told the computer to print them together on one line, just as we would normally write the problem on paper. (In this answer there is one empty space before 8. That space is used to display the minus sign (-) if the answer is a **negative number** below zero—for example,  $3 - 5 = -2$ .)

Let's use the INPUT command and the PRINT command in the same simple program.





Sometimes it is easier to read displays when there are **empty lines** between words, or between sets of words.

```
10 PRINT "JOHN"  
20 PRINT  
30 PRINT "AND"  
40 PRINT  
50 PRINT "MARY"  
RUN  
JOHN  
AND  
MARY
```

← One empty line between  
← One empty line between

When only the PRINT command is entered on the line (as in command lines 20 and 40), an empty line is displayed.

## INPUT

```
10 INPUT "A is ";A  
20 INPUT "B is ";B  
30 PRINT "A+B=" ;A+B  
40 PRINT "A-B=" ;A-B  
RUN  
A is ? 15  
B is ? 3  
A+B= 18  
A-B= 12
```

Here is another program that combines the INPUT and PRINT commands. It's easy to see that it tells the computer to **ask** for values for A and B and then to **display** two calculations. Now let's put the two INPUT commands together, and add some more calculations.

```

10 INPUT "A and B are ";A,B
20 PRINT "A=";A,"B=";B
30 PRINT
40 PRINT "A*B=";A*B
50 PRINT "A/B=";A/B
RUN
A and B are ? 15,3
A= 15          B= 3

A*B= 45
A/B= 5

```

In line 10, you can enter two values for two different variables,—but the , (comma) between the two values (in this case, 15 and 3) is **very** important. If the computer displayed 153 instead of 15,3 it would be difficult to understand. (The PRINT command in line 20 also has important punctuation. Do you know what the , and ; mean?)

Now we will add a **string variable** to our INPUT command.

```

10 INPUT "Name";N$
20 PRINT N$;" is great!"
RUN
Name? John
John is great!

```

When the variable name ends with the \$ sign, the variable "takes the value of" a letter or a word. Here is a program that uses both a letter variable and a number variable.

```

10 INPUT "Name";N$
20 INPUT "Age";Y
30 PRINT N$;" is ";Y;" years old."
RUN
Name? John
Age? 10
John is 10 years old.

```



## FOR—NEXT

```
10 CLS
20 FOR I=0 TO 36
30 LOCATE I,10
40 PRINT "$"
50 NEXT I
```



```
#####
```

In this program the repeat function (**loop**) of the FOR-NEXT command makes the computer display \$ signs on the screen from position 0,10 to position 28,10.

Here is the same program with a new direction added.

```
10 CLS
20 FOR I=0 TO 36 STEP 3
30 LOCATE I,10
40 PRINT "$"
50 NEXT I
```



```
$ $ $ $ $ $ $ $ $ $ $ $ $
```

What is the meaning of **STEP 3** in line 20? As you can see, it has moved the cursor over **three steps** after each \$ sign. In other words, the value of I now changes in steps of 3, so the \$ sign appears at positions 0,10 and 3,10 and 6,10...—not at position 0,10 and 1,10 and 2,10... The \$ signs cover almost the same area of the screen as in the above program, from 0,10 to 36,10.

By adding **STEP and a number** to the FOR command, then, you can **make the variable change in steps**. In the last example, the variable was the position, so the position changed in steps. The next program uses the STEP direction to change the way the computer counts.

```

10 FOR I=50 TO 0 STEP -5
20 PRINT I,
30 NEXT I
RUN
  50      45
  40      35
  30      25
  20      15
  10       5
  0

```

**I=50 TO 0** makes the computer count as it does each FOR—NEXT loop. There are 51 values for I between 50 and 0. But the **STEP -5** command makes the values decrease in steps of -5, so there are only 11 values for I.

#### FOR—FOR—NEXT—NEXT: A Loop Inside a Loop

```

10 FOR I=0 TO 2
20 PRINT "I=";I
30 FOR J=0 TO 4
40 PRINT "J=";J;
50 NEXT J
60 PRINT
70 NEXT I

```

FOR  
{  
NEXT

FOR  
{  
NEXT

```

RUN
I= 0
J= 0 J= 1 J= 2 J= 3 J= 4
I= 1
J= 0 J= 1 J= 2 J= 3 J= 4
I= 2
J= 0 J= 1 J= 2 J= 3 J= 4

```

This program shows how we can put one FOR-NEXT loop in the middle of another, wider FOR-NEXT loop. Variable I is controlled by the first and last lines of the program, and variable J is controlled by lines 30 and 50. The result is that for each I value, the computer makes four small loops to find all the J values. I changes from 0 to 2, and with each I, J changes from 0 to 4.

Here is another way to use a loop.

```
10 INPUT N,S
20 CLS
30 FOR I=0 TO N STEP S
40 LOCATE 5,10:PRINT "I=";
50 PRINT USING "####";I
60 FOR J=0 TO 500
70 NEXT J
80 NEXT I
```

This program uses the INPUT, FOR-NEXT and STEP commands to make variable I change in loops from 0 up to N, in steps of S. Of course, we will give values to N and S from the keyboard.

But what is the function of variable J and its short FOR-NEXT loop in lines 60 and 70? J is never printed. This variable is never used at all, but the computer will find 501 values for J each time, before it continues calculating and printing I. This is a **dummy** loop. It takes up some time—long enough for the computer to do 501 FOR-NEXT loops—but it does nothing else.

The function of the J loop, then, is to put a **time interval** into the bigger I loop. When you run this program, you will see that the computer pauses after each display.

Now you know how quickly your Sony Computer can do 501 simple things (rather fast!), and how to use a dummy loop to put some extra time in your program. Can you write a dummy loop that keeps the computer busy for a longer time?

## IF—THEN and IF—THEN—ELSE

---

Here is another guessing game, where you will try to match the number that the computer chooses. But this game is a little different from the guessing game and slot machine game that we programmed earlier in this book. Those were games of pure luck, and this time you can use some **skill** to guess the correct number more quickly. If you guess cleverly, you should be able to find the number in six or seven tries. But if you guess at random, it might take you many more guesses than that.



```

10 X=INT(RND(1)*100)+1
20 INPUT "Make a guess";A
30 IF A=X THEN GOTO 70
40 IF A>X THEN PRINT "SMALLER"
50 IF A<X THEN PRINT "LARGER"
60 GOTO 20
70 PRINT "Hit!"

```

Lines 30, 40 and 50 tell the computer to compare your guess to the value of X, and give you a hint to help you make your next guess.

Can you find the change in this next program?

```

10 X=INT(RND(1)*100)+1
20 INPUT "Make a guess";A
30 IF A=X THEN 70
40 IF A>X THEN PRINT "SMALLER"
50 IF A<X THEN PRINT "LARGER"
60 GOTO 20
70 PRINT "Hit!"

```

This program is actually the same as the previous one, even though line 30 has been changed. IF—THEN has the same meaning as IF—THEN—GOTO. Also, the program would still be the same with: 30 IF A=X GOTO 70. In other words, IF—GOTO has the same meaning as IF—THEN—GOTO. Either THEN or GOTO can be eliminated (but not both!)

Now we will combine two lines of this program into a single line:

```

10 X=INT(RND(1)*100)+1
20 INPUT "Make a guess";A
30 IF A=X THEN 70 ELSE
    IF A>X THEN PRINT "SMALLER"
50 IF A<X THEN PRINT "LARGER"
60 GOTO 20
70 PRINT "Hit!"

```

The two IF—THEN commands in lines 30 and 40 have become one IF—THEN—ELSE command. There is no line 40 now, but the meaning is the same. If that is possible, then it should also be possible to make lines 30 and 50 in this program into just one line.

```

10 X=INT(RND(1)*100)+1
20 INPUT "Make a guess";A
30 IF A=X THEN 70 ELSE
    IF A>X THEN PRINT "SMALLER"
    ELSE PRINT "LARGER"
60 GOTO 20
70 PRINT "Hit!"

```

You can check to see if each of these four programs is actually the same, by running each one on your computer.

## DIM (Array Variables)

When we used array variables in the Slot Machine Game program, we used the command

```
DIM A (2)
```

to make three variables:

```
A (0)    A (1)    A (2)
```

Then we changed the numbers in the brackets to a variable, (I), and told the computer that I=0 TO 2. We still had the same three array variables, and the computer changed them in a FOR-NEXT loop.

Array variables can have more than one character in the brackets. For example, if we enter

```
DIM A (3,4)
```

then we will have 20 variables:

A (0, 0)	A (1, 0)	A (2, 0)	A (3, 0)
A (0, 1)	A (1, 1)	A (2, 1)	A (3, 1)
A (0, 2)	A (1, 2)	A (2, 2)	A (3, 2)
A (0, 3)	A (1, 3)	A (2, 3)	A (3, 3)
A (0, 4)	A (1, 4)	A (2, 4)	A (3, 4)

Do you understand the 20 different variables? The first number in brackets can have four values, and the second number can have five values—4 times 5 is 20. (How many variables would we have if we write DIM A (9,9)?)

We showed you the 20 variables not in a simple list, but in the form of a **chart**. (You might think of it as a two-dimensional array, with the first value changing from left to right, and the second from top to bottom). As you can see, array variables are very useful for programs that make charts and tables. To use these two-dimensional array variables in a program, we would replace the numbers in brackets with variables, such as A (I,J). Here is an example of a real chart:

	Reading	Writing	Math	Total
1st term	68	88	70	226
2nd term	73	53	91	217
3rd term	92	98	82	272
Total	233	239	243	715
Average	77	79	81	238

This chart shows somebody's grades in three classes for three school terms. With the totals and averages, there are exactly 20 different numbers in the chart. That means we can use the same DIM (3,4) command to make enough variables for this chart. Now, let's think of ways that we can use programs to assign actual values to our array variables, A (I,J). Here's one:

```
100 FOR J=0 TO 2
110 INPUT A(0,J)
120 NEXT J
```

These three lines are for the Reading grades, which become variables A (0,0), A (0, 1) and A (0, 2). Each time the computer comes to line 110 in the FOR—NEXT loop, it will ask us to input the values (68, 73, 92).

Next, to calculate the total of the three Reading grades, we enter:

```
200 T=0 ..... T is the variable for Reading Total
210 FOR J=0 TO 2
220 T=T+A(0,J)
230 NEXT J
240 A(0,3)=T
```

In the same way, we can enter the 1st Term Writing grade into variable A (1, 0), and the 1st Term Math grade into A (2, 0). To enter the 1st Term total for the three classes into A (3,0), the following program will do.



```

300 T1=0          .....T1 is the variable for 1st Term Total
310 FOR I=0 TO 2
320 T1=T1+A(I,0)
330 NEXT I
340 A(3,0)=T1

```

To program the other grades and totals, and the averages, we can use similar FOR—NEXT loops with array variables in the form of A (I,J). This is how the entire program will look:

```

10 / *** GRADES CHART PROGRAM ***
20 /
30 DIM A(3,4)
40 CLS
50 /
60 / *** ENTER GRADES ***
70 FOR I=0 TO 2
80 FOR J=0 TO 2
90 ON I+1 GOTO 100,120,140
100 LOCATE 0,J:PRINT "Reading, term ";J+1;
110 INPUT A(0,J):GOTO 160
120 LOCATE 0,J+3:PRINT "Writing,term ";J+1;
130 INPUT A(1,J):GOTO 160
140 LOCATE 0,J+6:PRINT "Math, term ";J+1;
150 INPUT A(2,J)
160 NEXT J
170 NEXT I
180 /
190 / **CALCULATE TOTALS AND AVERAGES**
200 FOR J=0 TO 2
210 T=0
220 FOR I=0 TO 2
230 T=T+A(I,J)
240 NEXT I
250 A(3,J)=T
260 NEXT J
270 FOR I=0 TO 3
280 T=0
290 FOR J=0 TO 2
300 T=T+A(I,J)
310 NEXT J

```

```

320 A(I,3)=T
330 A(I,4)=INT(A(I,3)/3)
340 NEXT I
350 /
360 / *** MAKE CHART ***
370 CLS
380 LOCATE 5,0
390 PRINT "READ WRITE MATH  TOTAL"
400 FOR S=1 TO 3
410 LOCATE 2,S+1:PRINT S
420 NEXT S
430 LOCATE 1,5:PRINT "TTL"
440 LOCATE 1,6:PRINT "AVR"
450 FOR I=0 TO 3
460 FOR J=0 TO 4
470 LOCATE I*6+4,J+2
480 PRINT A(I,J)
490 NEXT J
500 NEXT I

```

To make our planning easier, we divide the program into three sections—Enter Grades, Calculate Totals and Averages, and Make Chart. Also, if you write on paper the places where each variable is entered, it will be easier to remember how the program works.

Naturally, array variables can be used for other purposes than “slot machines” and charts. And, as you may have guessed, the number of variables inside the brackets can be more than two: A (P,Q,R) or B (X,Y,Z,XY,XZ,YZ) ... or any other combination, up to 255 different variables! Computer professional often use array variables to design video games, or to calculate the finances of large businesses, or for other complex jobs. Your Sony Computer can use array variables for many kinds of things, as you will learn when you keep practicing programming with your friends, or with other books. You already have a good start, because you have used array variables in two different programs.

So good luck to you, and keep practicing!

# INDEX

## A

Address ..... 64  
Array variables ..... 77-78, 109

## B

Brackets ( ) ..... 42  
Bugs ..... 32

## C

Characters ..... 26, 38  
Charts ..... 110  
CLS ..... 64  
Colon ( : ) ..... 42  
COLOR ..... 12-13, 48  
Color Chart ..... 12  
Commands ..... 5, 8, 45  
comma ( , ) ..... 42  
Conditional formula ..... 59, 62  
Conditional loop ..... 68  
CSAVE ..... 53  
CTRL key ..... 8

## D

Decimal point ..... 43  
DIM ..... 77-78, 109  
Dollar sign (\$) ..... 84  
Dummy loop ..... 107

## E

ELSE (IF-THEN-ELSE) ..... 62, 107  
Error message ..... 10, 14, 34, 42

## F

File name ..... 53  
Format ..... 88  
FOR-NEXT ..... 68-69, 87, 105

## G

GOTO ..... 38  
Graphics ..... 37

## H

Hyphen ( - ) ..... 49

## I

IF-THEN ..... 59, 107  
IF-THEN-GOTO ..... 108  
Initial value ..... 68, 69, 81  
INKEY\$ ..... 87  
Input ..... 8  
INPUT ..... 57, 103  
INT (integer) ..... 43

## K

Keyboard ..... 6

## L

LINE ..... 49  
Line number ..... 35  
LIST ..... 35, 92  
LOCATE ..... 65  
Loop ..... 39, 68, 106

## M

MOD (modulus) ..... 71

## N

Negative number ..... 101  
NEW ..... 35

## O

ON-GOTO ..... 81

## P

Programming ..... 28, 34  
PSET ..... 38  
PRINT ..... 14, 17-18, 100  
PRINT USING ..... 88  
Punctuation ..... 42

## Q

Quotation marks ( " ) ..... 18, 85

## R

Random numbers ..... 43, 81  
REM ..... 97  
Remarks ..... 97  
RENUM ..... 95



Renumber.....	95
<b>RETURN</b> key .....	9
RND (1).....	43-44, 81
RUN .....	34

**S**

SCREEN .....	37
Sequencing .....	28
<b>SHIFT</b> key.....	14
Single quotation mark .....	98
Space .....	10, 66, 101
STEP .....	105
STOP key.....	8
String Variable .....	83, 104
Subtitles.....	97

**T**

Titles .....	96-97
TO.....	69

**U**

USING (PRINT USING).....	88
--------------------------	----

**V**

Variables.....	19, 25, 67
----------------	------------

THIS BOOK WAS  
SCANNED ON WEDNESDAY  
7TH OF NOVEMBER 2018  
FOR THE BENEFIT OF ALL  
BY PAUL KENNEDY OF  
THURSO SCOTLAND.  
HP PSC 1510 USED  
DURATION OF TWO DAYS  
ACTUAL TIME OF 9 HOURS  
APPROXIMATELY.



