Trying Forth Programming

1. Use of the data stack - for passing variables between 'words'

100 .	(puts 100 on stack and prints it - removed from stack)
100 200 . .	(puts 100 and then 200 on stack - printing in reverse order)
100 200 SWAP . .	(swaps both items on stack and prints in reverse order)
500 DUP . .	(500 is duplicated on stack)
1 2 3 ROT . . .	(move 1 to the top, printing '1 3 2' instead of '3 2 1'
40 30 * .	(prints 40x30 and removes both items from stack)
8 4 / .	(prints 8/4)
4 3 / .	(prints integer result of 4/3)
90 50 30 */ .	(prints (90x50)/30)
8 FINT 6 FINT F/ F.	(8/6 in floating point)

2. Defining new words

VLIST	shows all the current words in the dictionary
42 EMIT	displays a star ('*')
: star 42 EMIT ;	new word 'star' compiled on dictionary - use VLIST to see it
star	executes the word and displays '*'

3. Looping (DO-LOOP)

: nstar 0 DO star LOOP ;	new word 'nstar' defined in terms of 'star'
10 nstar	prints 10 stars

4. CONSTANTS and VARIABLES

Relatively few variables are required, as the data stack is used to pass arguments between words.

10 CONSTANT fred	fred is defined as a constant of 10
fred .	prints '10'
50 VARIABLE jim	jim has been initialised to 50
jim @ .	jim puts address of value on stack, @ reads the contents of the address, and . prints it
-10 jim !	! stores the value -10 into jim

5. STRUCTURES - the "Jewel" of Forth

Quite complicated strutures such as lists, patients' records, arrays etc can be defined relatively easily using **<BUILDS ... DOES>** (later Forth versions use **CREATE**....**DOES>**)

1-dimensional array-defining word:

```
: 1-array  <BUILDS  4  *  ALLOT
          (compiles array name in dictionary and  creates space for array elements)
          DOES> SWAP 4 * +   ;     (returns address of required element of array at run-time)
```

then, 10 1-array mydata defines array called 'mydata' of 10 elements

```
100 5 mydata !              (stores 100 as the 5th element of array)
    5 mydata @ .            (reads it back and prints it)
```

An array of paired numbers (eg for complex arithmetic) may be defined as follows:

```
: zarray  <BUILDS  8 * ALLOT
          DOES>   SWAP  8 * + DUP 4 + ;
```

Then,

```
10 zarray myzarray
```

defines an array called 'myzarray'

When typing in, say: 5 myzarray

the addresses of the 5th complex pair will be left on the stack.

To get the actual values requires a little stack manipulation:

```
@ SWAP  @ SWAP
```

Hence we may define a word to read the zarray as follows:

```
: rd_z   myzarray  @  SWAP  @  SWAP  ;
```

which would be invoked by:

5 rd_z to get the values of the 5th element-pair on the stack

Similarly we may define a word to write a data pair to an element as:

```
: wr_z   myzarray  ROT  SWAP   !   !   ;
```

Then,

```
200  550  5  wr_z
```

would store 200,550 as the 5th element-pair.