Teaching guide: Programming concepts (Variables and constants)

This resource will help with understanding the use of variables and constants in programming concepts. It supports Section 3.2.2 of our GCSE Computer Science specification (8525). This guide is designed to address the following learning aims:

- appreciate the need for variables.
- be able to declare and assign variables.
- understand the concept of a constant.
- be able to declare and use constants.

Storing and retrieving data

Computer programs process data and this would be impossible if at least some of that data could not be stored and then retrieved later as the program progresses. Variables are created to hold data so that the values can be retrieved (accessed) and changed as the program is executed by the computer.

For example, there are values that need to be stored and recalled in this simple number guessing game:

1. The setter thinks of a number between 1 and 100
2. The guesser has a guess at this number
3. Is this guess the same as the setter's number?
    a. YES then the game ends
    b. NO then setter tells the guesser if it is too high or too low
4. Go back to step 2

Even though this is a very simple game it would be impossible to play unless the following values were stored:

- the number the setter thinks of in step 1
- the guess (which must be remembered between steps 2 and 3 each time)

The programmer should include these variables (at least) in their program. This section will cover how variables can be created and used.

Variable declaration and assignment

Variables are locations in memory where data is stored.

Each of these locations has a number (its address) that is used by the computer to locate them, a bit like your postal address. Because of the way computers operate, that number will probably be different every time the program is run and will also depend on the computer used to run it and so these memory addresses are not used directly.

Every time you need to use a location in memory to hold data you declare a variable with a name instead of a number – you then let the computer decide which memory location to use. Every time you want to access the value kept in that memory location, or change it, you use the variable's name. This way you can always be sure which item of data you are referring to. The following pseudo-code statement stores the integer `14` in a memory location that can then be referred to as `age`

```
age ← 14
```

which you can read as "the variable `age` is assigned the value `14`".

Some programming languages require that you also say what type the variable must be (in the previous example `14` is an integer).

The most significant difference between pseudo-code and most programming languages is the ← symbol used for assignment. Programming languages often use the = symbol for assignment and == to mean 'is equal to'.

This is how this pseudo-code could be implemented in different programming languages:

Programming language	Implementation
C#	int age = 14;
Python	age = 14
VB.Net	Dim age as Integer = 14

Assigning new values

As the name suggests the value that a variable holds can be changed. Without this facility all but the simplest programs would be impossible to write. Changing the value a variable holds is very simple: just assign an existing variable a new value using the ← symbol and the old value is overwritten with the new one.

In the following example a programmer is defining the width of a paint brush – this is assigned a new value three times as the program executes.

```
# short program to show how variable values can change
brush_width ← 242
draw_square(brush_width)

brush_width ← 187
draw_square(brush_width)

brush_width ← brush_width - 20
draw_square(brush_width)
```

The second line in our program assigns the value `242` to the variable `brush_width`. After the instruction to draw a square the programmer changes the value of `brush_width` to `187`. This is still the same variable but now it has a new value and from that line of code until it is given another new value the value stored in the variable will be `187`.

The final assignment looks odd at first sight – it assigns itself its own value minus 20. There is nothing particularly strange about this line though if you evaluate the right-hand side first. You need to know the value of `brush_width` which is `187` at this point in the program and then subtract `20` from it to get `167`. Now you know the value of the expression and you can assign this to `brush_width`.

It is worth noting two things here about variable names in general:

- The variable's name, called an identifier, should always describe the value it holds. `brush_width` is a much better name than `xyz`, even though both would be allowed in a language.
- The variable name contains no spaces. Different languages have different rules about naming identifiers but good rules of thumb are:
    - always start with an alphabetic character
    - leave spaces out
    - avoid punctuation (except the _ symbol)

Expressions involving variables

It was shown in the previous example that expressions will normally involve variables, possibly even the same variable to which the value of the expression is being assigned, as in

```
brush_width ← brush_width – 20
```

Any expression can be used on the right-hand side of an assignment statement as long as it evaluates to a value that has the correct type for the variable on the left-hand side.

You can use as many variables and operators as you need as long as the purpose of the expression can be reasonably worked out by a human reader. If you find that you are writing an expression which looks (too) complicated you may want to separate it into separate sub-expressions and assign each of these values to separate variables. For example, if we have a quadratic equation of the form

$$ax^2 + bx + c = 0$$

and we know the values of a, b and c we can calculate the roots (solutions) of the equation (these are the values of x for which $ax^2 + bx + c = 0$).

One of the roots for this quadratic equation is given by this formula:

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

If you were to write this out as one assignment statement it might look like this:

```
solution ← (-b + SQRT(b^2 – 4 * a * c)) / (2 * a)
```

The ^ symbol is used to indicate 'to the power of' and `SQRT` is a subroutine that calculates the positive square root.

Writing exactly the same code, but splitting it over two lines and using an intermediate variable called `discriminant` makes it easier to read and understand:

```
discriminant ← b^2 – 4 * a * c
solution ← (-b + SQRT(discriminant)) / (2 * a)
```

The following program uses variables in the expressions and also reassigns values to variables. It calculates the area of three rectangles where one of the sides increases by 1 every time.

```
length ← 10
width ← 5
area ← length * width
OUTPUT area
width ← width + 1
area ← length * width
OUTPUT area
width ← width + 1
area ← length * width
OUTPUT area
```

This program uses three variables to store the data needed as the program executes. If you take a look at the expressions on the right-hand side of every assignment statement in this program, you can work out their value and then assign this to the variable on the left-hand side.

The **`OUTPUT`** statement means just that – output the value of the variable presumably, but not necessarily, to a screen. (The Teaching guide - Programming concepts (Iteration) covers a way to write programs that repeat statements using a loop and this could be used to make the program above shorter.)

Variables vs constants

As we have seen, variables are the names given to locations in memory that let the programmers store and refer to data as their programs execute. Constants do exactly the same thing except, as the name suggests, their values cannot be changed once they have been given a value.

The mathematical value π (pi) is a constant value and if you were to use π in your programs you wouldn’t want to have to type out 3.14159 (or even more decimal places) every time you needed it, nor would you ever want to give it any other value. These Teaching Guides will use the following pseudo-code to declare a constant:

```
CONSTANT PI ← 3.14159
```

This constant can be used in an expression in exactly the same way as a variable. For example, to work out the circumference of a circle with a radius of `10`:

```
r ← 10
circumference ← 2 * PI * r
```

Your programming language may support declaring constants directly. Other languages, such as Python, do not have a keyword such as `constant` or `final` but use the convention that if you declare a variable name using all upper case characters then it is to be used as a constant. These resources will also use upper case letters when defining constants.

Advantages of using constants:

- Programs are clearer and easier to understand (for example, defining a constant to hold the current VAT rate (**CONSTANT** `VAT_RATE ← 20`) gives the reader of your code much more information about what's going on than just using `20`.
- Since the value for the constant is only held in one place, changing that value means only changing one line rather than searching through a program for all occurrences of the value. Suppose you had used `20` throughout your program for a customer discount but then decide to change it to `25`. After a search and replace you find that your invoices are wrong because you also used `20` when calculating VAT, and the VAT `20`s had also been changed to `25` (and even worse your address had changed from 20 Cobden Road to 25 Cobden Road!) If you'd used **CONSTANT** `DISCOUNT ← 20` and **CONSTANT** `VAT_RATE ← 20` then you'd just need to change the first statement to **CONSTANT** `DISCOUNT ← 25` and everything would work correctly, without any extra work for the programmer.

You can refer to other constants when defining a constant, eg

```
CONSTANT PI_BY_4 ← PI / 4
```

If you increase the accuracy of `PI` to 3.14159265 then `PI_BY_4` will also increase in accuracy, again without any extra work for the programmer.