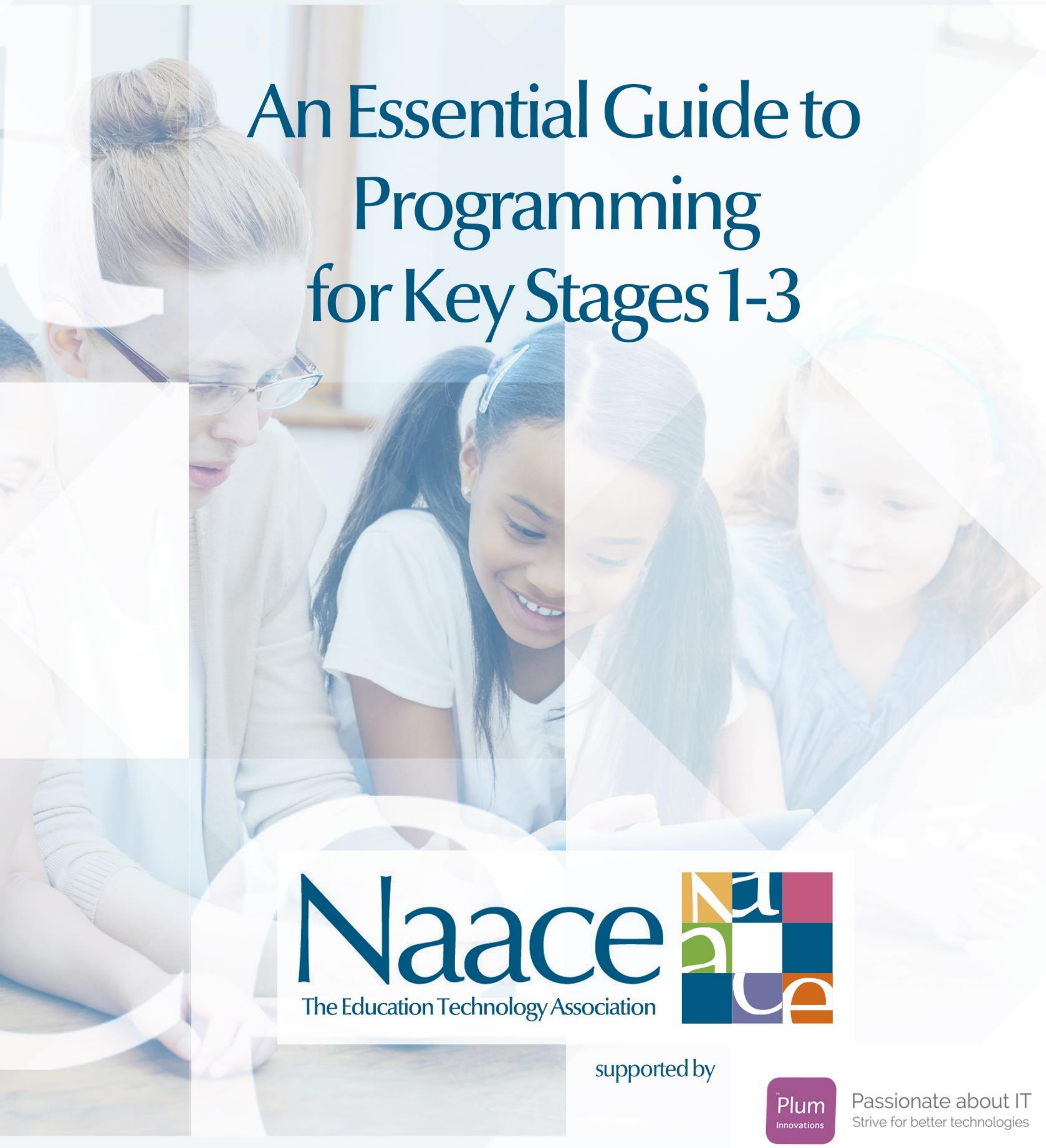


An Essential Guide to Programming for Key Stages 1-3

Naace
The Education Technology Association

supported by

Plum Innovations Passionate about IT
Strive for better technologies

A Naace Publication by Tim Scratcherd & Dr Carol Porter
Series Editors: Mark Chambers, Dr Carol Porter, Tim Scratcherd

Foreword from the Sponsors

Here at Plum Innovations, we have always been proud to be a company that is dedicated to helping teachers succeed in their professional fields with efficient and solid IT platforms by providing a stress-free IT experience. So having an opportunity to sponsor the Naace Essential Guides trilogy: Programming, Digital Literacy and Information Technology was very exciting for us.

As a Naace sponsor partner and British Computer Society organisational member, the chance to really help spread correct understanding of computing felt extremely important to us in regards to helping ensure teachers are able to teach Computing with confidence and ultimately help to boost children's future career prospects. These Essential Guides explain the boundaries and relationships between the three strands of the Computing Programme of Study. The authors of these guides stress the necessity of maintaining a broad, balanced computing curriculum with various technologies available to children, especially with the current emphasis on coding.

We hope after reading these guides, you will find the answers that you are looking for.

Plum Innovations Ltd

Email: info@pluminnovations.co.uk
Twitter: @pluminnovaz Website: www.pluminnovations.co.uk

About the Authors

Tim Scratcherd is director of Learning Linked. Contact him via tim@learninglinked.co.uk

Tim has in the past been Chair of the Naace Executive and Naace Board.

Dr Carol Porter was the Technology Curriculum Support Centre Manager in Bury LA, offering training and consultancy advice on the computing curriculum, schemes of work, progression and assessment to teachers in Bury. She also developed training courses based around the 2014 Computing curriculum and in effective uses of technology across the entire primary curriculum. Carol has in the past been Chair of the Naace Board of Management.

Summary

This eGuide is the first in a series about programming. It covers the big ideas of programming, by unpacking the Computer Science strand of the Computing Programme of Study. It does not go into detail about any one programming language, but a programming language should have within it most if not all of these big ideas. It will be followed by other eGuides, which show how the big ideas in this eGuide are expressed in a particular language, and how that language might be taught in schools.

Introduction

Programming is part of the Computer Science strand of the Computing Programme of Study. It is the **process** of **solving a problem** or **making something** by applying **computational thinking** to write **sequences of instructions.**

Computational thinking is a powerful way of problem solving and making, not limited to programming. Computational thinking includes a set of techniques, which can be expressed in normal language, and is the place where you start to solve problems and make things.

Once you have solved a problem using computational thinking, then there are lots of different **programming languages** with which you can write instructions. The one you choose will be appropriate to what you are making.

When you **make something**, the thing made is (the outcome), generally, either an **application** (such as you might download from an app store), a **device** (such as a drone), or an **online tool** (such as you can access using a web address). Behind all of these are sequences of instructions. The programming language you choose will be appropriate to one of these. For example, if you are making an online tool, you will certainly need to know about and use HTML (HyperText Markup Language). When planning your programming experiences for children, a good quality programming curriculum will include the opportunity to learn and solve problems in at least three languages, one for each of the main outcomes.

The process of actually writing the instructions in the chosen language is called **coding**. As you will see, although there is a lot of fuss made about coding, it is actually the activity which happens towards the end of the programming process!

There is a big difference between making something for **you** to use, and making something for **someone else** to use. When you are making something for someone else to use, you will need to explain how it works, and make it easy and attractive to use. For example, you are building an online guide to a walk. If you want someone else to use your map and photographs, you will need to tell them the web address, and carefully mark on the map where the photographs can be found. You will need to think about the **audience**, and the **human/computer interface.**

Problem solving is usually done in response to a challenge, and it should happen throughout schooling. Some challenges you might set are:

- Can you program a Beebot (a programmable toy) to get through a jungle without falling into any swamps?

- Can you use Logo to draw the plan of a tennis court?
- Can you use Scratch to make a virtual model of fish swimming in a fish tank?
- Can you make your Codebug flash a message?
- Can you link your photographs to a map, to show how a place looks?

Key Concepts

Computational thinking is at the heart of the whole Computing Programme of Study, not just the Computer Science strand. It is a way of thinking about solving problems and making things in general. Here is an everyday example.

To make a jigsaw,

- Turn all the pieces the right way up
- Find the four pieces with two straight sides (the corners)
- Use the picture on the box to put the corners in the right places
- Find the pieces with one straight side (the edges)
- Use the picture on the box to put the edge pieces in the right places
- Roughly position the other pieces to match the colours on the box
- Use detail on the pieces in position to look for pieces which might match
- Test to see if they do
- Do this again and again until all the pieces are in place.

Note that

- This is a good way to make a jigsaw
- It is in plain English
- It is not the only way to make a jigsaw
- It can be made more detailed.

It is in effect a sequence of instructions **which can be used by other people** to make jigsaws. It is an example of computational thinking. Any example where a sequence of instructions is given, to help others solve a problem, is an example of computational thinking. Other examples abound:

- Recipes
- Dance routines
- Directions to a destination
- Musical scores
- Knitting patterns.

For each of these, you might like to think about what a typical set of instructions might be. Note that the order of the instructions is important. This is less obvious in the jigsaw example, but imagine what would happen if the order were changed in directions to a destination.

Computational thinking applies to the other strands of the Programme of Study i.e. Information Technology and Digital Literacy. Here are examples from the Digital Literacy strand. The slides which go to make up a presentation can be thought of as a sequence, and when you put the slides together you are in a way encoding the presentation. If you make links between different slides of the presentation, you are coding different routes through the presentation, for a user to follow. When making a video, your storyboard can also be thought of as a sequence of instructions. When using a spreadsheet, the cells containing formulas can be thought of as instructions so that every time the spreadsheet is recalculated, the cells with formulas in them display the results of following an instruction to calculate. This idea can give powerful ways of using ideas from programming to solve problems in a spreadsheet instead. An example of this can be found in the Essential Guide to Models and Modelling. In there, a model of tossing coins is built in a spreadsheet. Each cell is turned into a coin, and then the concept of fairness of tossing a coin is investigated by recalculating the spreadsheet with ever growing numbers of coins.

A quick comparison of the examples shows two very important common structures, which control the flow through a sequence of instructions. **Repetitions** simply repeat a set of instructions. In the context of dance routines, say the words to the Hokey Cokey. In the context of the coin example above, the coin tossing is repeated many times. There are different sorts of repetition, often combined with **Decisions,** and the general word for all sorts of repetition is **iteration**. These two structures are first referred to in the Key Stage 2 Programme of Study:

> *use sequence, **selection**, and **repetition** in programs; work with variables and various forms of input and output,*

where selection is used to mean decision making.

These three ideas are generally enough to solve problems when programming. The first step is to give a solution in ordinary language. Then take each part of that solution, and rewrite it using only sequence, selection and repetition. Looking back at the jigsaw example, some of the ordinary language could be rephrased, for example:

> Find the four pieces with two straight sides

Becomes:

```
REPEAT
    Select a piece
    IF it has two straight sides, THEN place it to the left
    ELSE place it to the right
UNTIL all the pieces have been sorted.
```

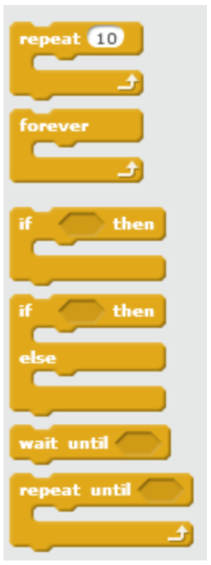
Note the REPEAT…UNTIL for iteration and the IF...THEN...ELSE for decision. If this was a real programming situation, you would then be able to write this in a computer language – the **coding** step. This transitional use of programming structures is formally described as **pseudocode**.

The programming examples which follow are simply given to show examples of iteration and decision as they appear in different languages. They will be put in context in the eGuides for particular languages.

Logo is a language specifically designed for education. It has been around since 1980, and still remains a great way of introducing students to programming. Its big idea is that you write a sequence of instructions to move an object, which may leave a trail as it goes. This is an example of repetition in Logo, to draw a square side 50 units. The plain language solution to the problem is to move forward 50 units and turn right 90 degrees, and do this four times. The logo instructions drive a pointer, which leaves a trail. Logo is a great language for solving simple problems which require graphical solutions:

Forward 50 units Right 90 degrees

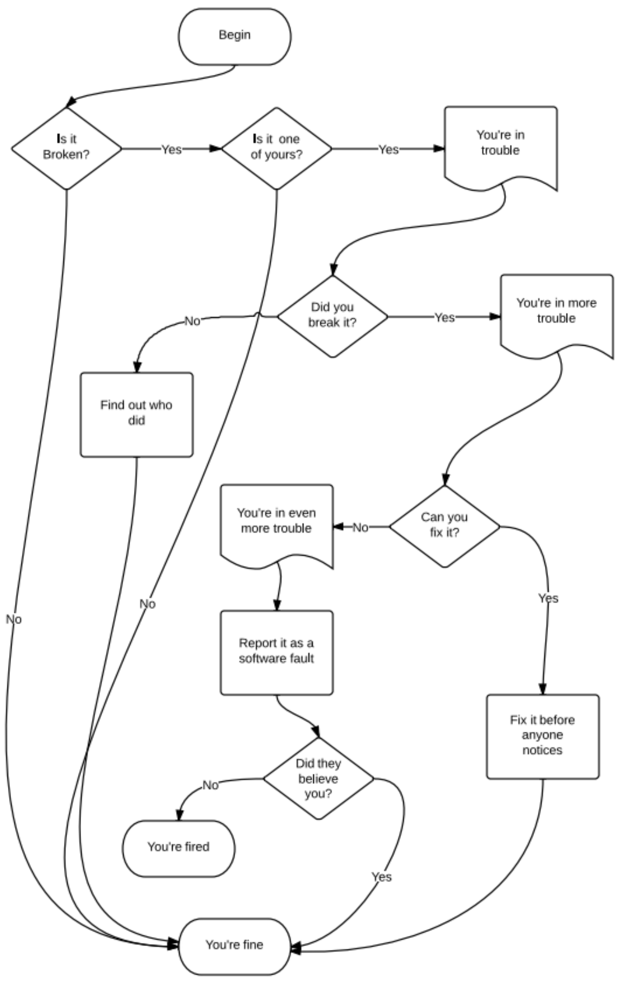
REPEAT 4 [FD 50 RT 90]



You can copy and paste this line into www.j2e.com/j2code and then watch what happens when the run button is pressed.

The language **Scratch** contains several iteration tools. Here is a sample from the Scratch control box. Which of these are iteration tools? Which are decision tools?

Next is a classic example of repetition in a simple **Basic**, which is being used to make a table of square numbers. The repetition is done by using a FOR…NEXT loop. Note the use of a variable, called N, which as the loop works starts off with a value of 1 and finishes when N is 10.



```
1000 FOR N=1 TO 10
1010 PRINT N; " ";N*N
1020 NEXT N
```

You can actually see this working if you copy and paste the code into the BASIC program window in www.quitebasic.com and then click on the run button. Remember to delete the current program first.

Decisions change the route through the instructions, depending upon a condition. A good way of representing these is with a **flowchart**. The shapes in flowcharts have defined meanings, and they are a great tool for moving from a program in plain English to a sequence of instructions in a particular programming language. They represent a decision by asking a yes/no question, and then depending on the answer, take action by following a particular line of instructions. **Flowol** is a programming environment where the coding is done as a flowchart. Most word processors contain editable versions of flowchart shapes. Here is a fun example about maintenance, which uses the shapes for comments, actions and decisions.

The main way that decisions appear when coding is by using IF…THEN, as you have seen above. You may have picked out the two decision tools from the Scratch control box above.

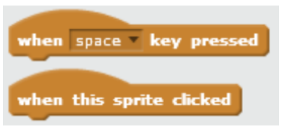
Algorithms are parts of programs. They are sequences of instructions **which perform a single task**. The task of finding the corner pieces in the jigsaw was an algorithm. In the Key Stage 1 Programme of Study, algorithmic thinking is a way of approaching a problem, which needs only a single idea to solve the problem.

There are other key concepts in the Key Stage 2 Programme of Study:

> *use sequence, selection, and repetition in programs; work with variables and **various forms of input and output***

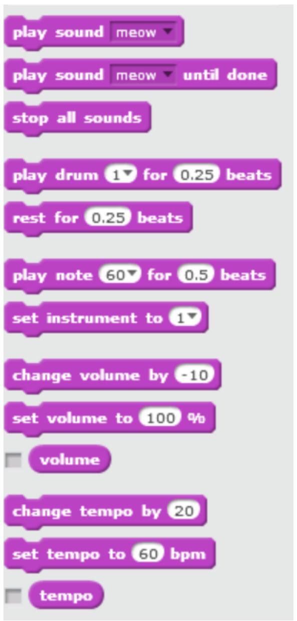
An **input** is something which allows you to interact with technology. You will be very familiar with keyboards, which provide inputs. They are in effect a set of switches, which when pressed, send a signal. Depending upon the key pressed, the device 'knows' how to respond. A mouse is also an input device. A mouse which moves a screen pointer, and has buttons on it, allows a much more flexible approach to input. Another very common form of input is the touch sensitive screen on a tablet, where both the keyboard and the mouse functions can be provided just by using your fingers. It is also becoming more and more common for devices to have both microphones and cameras, both of which are inputs. Note that they do not usually allow you to control your device, and so are less the concern of programmers. Instead, they provide a powerful, accessible way of providing services, and making multimedia.

An **output** is something produced by technology. By far the commonest will be on the screen, which can output all kinds of graphics, but sound can also be output, and printers are output devices.

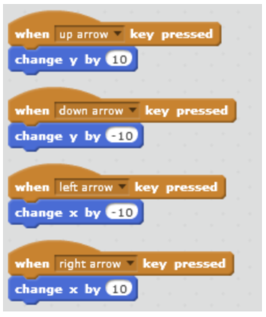


```
when space key pressed
when this sprite clicked
```

Programming environments should provide you with the means to control inputs and outputs from within programs, **so that you can make programs for other people,** who have no knowledge of programming, to use. For example, the programming environment Scratch contains tools under its events tab for input from the key board and by clicking on an object on the screen with the mouse and pointer.



```
play sound meow
play sound meow until done
stop all sounds
play drum 1 for 0.25 beats
rest for 0.25 beats
play note 60 for 0.5 beats
set instrument to 1
change volume by -10
set volume to 100 %
volume
change tempo by 20
set tempo to 60 bpm
tempo
```



```
when up arrow key pressed
change y by 10

when down arrow key pressed
change y by -10

when left arrow key pressed
change x by -10

when right arrow key pressed
change x by 10
```

Here in these four blocks of code is a way of programming the arrow keys to move a sprite round the screen. Just make these in the Scratch scripting area and then try out the arrow keys. One of the most powerful things about Scratch, which this illustrates, and which makes it very different from Logo and Basic, is that you can have **multiple blocks of code which work at the same time**. They are not working in sequence. So with regard to output, the big idea behind Scratch is that you can attach sequences of instructions to sprites, so that things happen on the screen. Scratch can also output sounds. In fact, Scratch has sufficient commands for it to be thought of as a simple music editor.

> *use sequence, selection, and repetition in programs; work with **variables** and various forms of input and output*

A **variable** is used when you want to refer to a number in a program, but you want the value of the number to be able to change. Programming environments use variables in different ways. You have already seen a variable (N) being used in the BASIC example above.

Here is an example from Logo, which can be found at
http://www.j2e.com/code/examples/Logo/Polygons

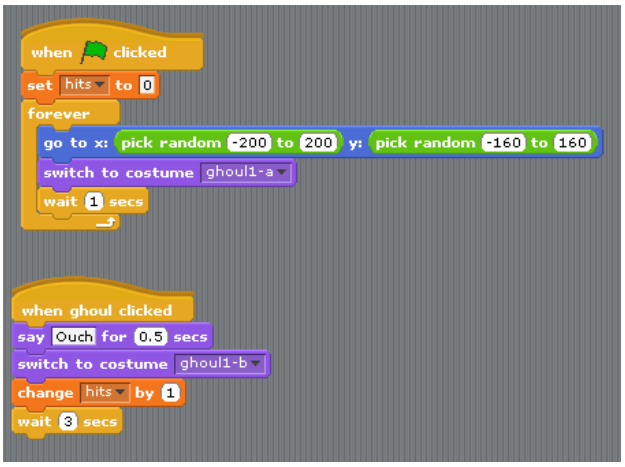
This is an algorithm, because it has one task; to draw a polygon.

```
TO POLY :sides
REPEAT :sides [FD 100 RT 360 / :sides]
END

CS
PD
POLY 8
```

The name of the variable is :sides. The first three lines define the algorithm. After that, CS clears the screen, PD puts the pen down, and then to draw the polygon, you enter POLY followed by the number of sides. This draws a regular octagon. The power of this algorithm comes from the fact that by using the variable, you can draw any regular polygon by simply giving the number of sides as part of the command.

Here is an example of using a variable as a counter in Scratch 1.4.



```
when ⚑ clicked
set hits to 0
forever
    go to x: pick random -200 to 200 y: pick random -160 to 160
    switch to costume ghoul1-a
    wait 1 secs

when ghoul clicked
say Ouch for 0.5 secs
switch to costume ghoul1-b
change hits by 1
wait 3 secs
```

It is a very simple game, where a sprite representing a ghoul jumps randomly round the screen. If you click on it, it says 'Ouch' and turns red. The variable **hits** is the counter and it counts the number of successful times you do this. In the script you can see the way the counter works by looking at the orange block.

The concept of a variable is an abstract one, and needs to be introduced carefully. It is not at all the same concept as that of a variable in mathematics. For example, the following will print the numbers 1 to 10 in Basic. The variables are N and X.

```
1000 LET X=0
1010 FOR N=1 TO 10
1020 LET X=X+1
```

```
1030 PRINT X
1040 NEXT N
```

Copy and paste this into Quitebasic, if you like.

This is a pretty pointless algorithm, but it illustrates a very important point about the difference between mathematical variables and programming variables. The best way to think of a variable is that it is a box, with a name, into which you can put a number. Look at line 1000. This puts the number 0 into a box named X. Then at line 1020, we take whatever number is in box X, add 1 to it, and put it back in the box. This works fine. But as a mathematical equation, X=X+1 has no meaningful solution. There is no finite number which is one greater than itself.

*design, write and debug programs that accomplish specific goals, including controlling or simulating physical **systems**; solve problems by decomposing them into smaller parts*

If you put together the concept of inputs and outputs with the idea that a program **processes** the inputs to achieve the outputs for a given purpose, you have the concept of a **system**. There is more detail in the key concepts in the Essential Guide to Models and Modelling.

Systems thinking is the general way of solving problems. The problem solving process goes like this.

- What do I want my system to do? In other words, what sort of **outputs** do I want, to achieve my purpose?
- What will my system need to know? In other words, what **inputs** are required?
- What do I have to do, to get the outputs from the inputs? What **processing** is required?

Often the process can be a sequence of instructions, a program. Apply this to the game in Scratch above. The purpose of the game is to zap as many ghouls as possible. The outputs are an 'ouch', a temporary change of colour, and the total number of successful zaps. The inputs are the position of the pointer and the mouse clicks, which will be successful if the pointer is on the ghoul when the mouse is clicked. The processes are the scripts above.

*design, write and debug programs that accomplish specific goals, including **controlling or simulating physical systems**; solve problems by decomposing them into smaller parts*

Physical systems are generally those things which are not computers, but which are programmed to achieve their purposes. They use **generalised** inputs and outputs. How does an automatic door 'know' how to open when you approach it? It has a **sensor**, which detects your presence. There are many different sorts of sensors, which can be used by physical systems, and these include light, sound, pressure and position. Physical systems can also make all kinds of things happen, by switching other things on and off, such as lights and motors. This is generalised output. More detail of this is in the Essential Guide to Information Technology.

There are some physical systems especially designed for educational use. The **BBC micro:bit**, being given to Y7 pupils, and the **Codebug**, more suitable for primary pupils, are two very good examples. For both of these, the program is written and tested in simulation on a computer, and then the instructions are downloaded to the device. Once this is done, the device can be disconnected and will work standalone. There are other systems, which offer generalised input and output and can also act as computer systems. The two most common are the **Raspberry Pi**, and the **Arduino**.

Flowol is a good way of **simulating** physical systems. The environment uses the concept of mimics of all sorts of machines, including car parks and fairground rides. Like Scratch, Flowol can also have separate blocks of code which execute at the same time. This is often called **parallel processing.**

> *design, write and debug programs that accomplish specific goals, including controlling or simulating physical systems; solve problems by **decomposing them into smaller parts***

The concept of **decomposition** goes along with the concepts of algorithm, and program. It is a key approach to solving a problem, by breaking up the problem into smaller parts, finding the solutions to those parts, and then putting it all together. You have seen two examples of this, in the separate steps for how to make a jigsaw, and in the Scratch game, where the first block randomly moves the ghoul, and the second block deals with what happens when the ghoul is clicked.

There are further key concepts, needed for programming solutions to real world problems.

> *design, use and evaluate **computational abstractions** that model the state and behaviour of real-world problems and physical systems*

A **computational abstraction** is a representation of something in the real world, in a programming environment. Defining and using computational abstractions is part of the overall modelling process. More detail can be found in the Essential Guide to Models and Modelling. Here is a simple example, appropriate at Key Stage 2. Pupils were challenged to use Logo to build a simple elevation of a house. This was one solution. The code, in Logo, was

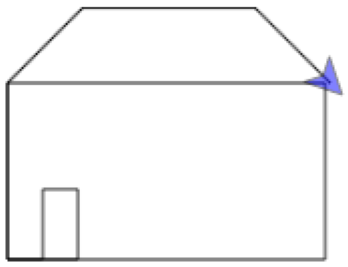
```
TO WALL
REPEAT 2 [FD 100 RT 90 FD 180 RT 90]
END

TO DOOR
REPEAT 2 [FD 40 RT 90 FD 20 RT 90]
END

TO ROOF
RT 45 FD 60 RT 45 FD 98 RT 45 FD 60
END

HOME
CS
WALL
RT 90 FD 20 LT 90
DOOR
HOME
FD 100
ROOF
```

When run, the output was



You can see this for yourself by copying the code into j2code, or if you subscribe to Purple Mash from 2Simple, their coding tool simply called 2Code. Here the pupil built three computational abstractions, and labelled them well as WALL, DOOR, and ROOF. Furthermore the pupil defined them as separate procedures, then called them up by name. This is an example of a **modular program** which uses **procedures**, as in:

> *make appropriate use of data structures [for example, lists, tables or arrays]; design and develop **modular programs** that use **procedures** or functions*

Lists, tables and arrays are all methods for extending the concept of a variable.

An example of using a **list** in Logo can be found in j2code at http://www.j2e.com/code/examples/Logo/lists where the list can be seen in the line make "col ["red "yellow "green "blue]. The list is the four items between the square brackets.

An example of using an **array** in Basic can be found at http://www.quitebasic.com/prj/algorithms/bubble-sort/ . The name of the array is A, but the actual items in the array are variables with an index, for example A(1), A(2) and so on. The usefulness of the array comes from the fact that both the whole item and its index are variables.

<u>Key Outcomes</u>

Plan for progression in terms of programming environments.
In key stage 1

- Programmable toys

In key stage 2

- Logo
- Scratch
- HTML
- Device programming in blocks; Codebug

In key stage 3

- Device programming in blocks; the BBC Micro :bit
- Programming in a simple Basic
- Programming in Python; Raspberry Pi.

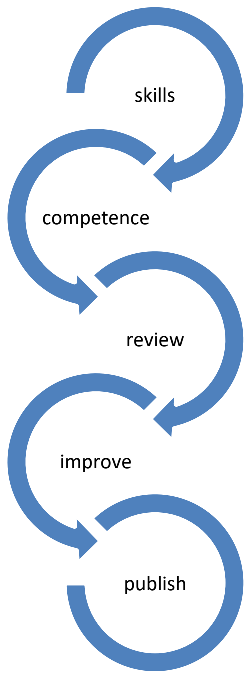
Bury SoW age related expectations

Year	End of Year Expectation
1	• Understand what algorithms are • Create and debug simple programs
2	• Understand how algorithms are implemented as programs on digital devices, and that programs execute by following precise and unambiguous instructions • Use logical reasoning to predict the behaviour of simple programs
3	• Use a variety of programming platforms to create a range of programs. • Design, write and debug programs that accomplish specific goals • Plan, create, test and modify algorithms to solve open ended problems using a variety of programmable devices. • Use more advanced programming, including penup/pendown, and repeat commands to create, test, debug, modify and refine algorithms and programs.
4	• Use a variety of programming platforms to create a range of programs. • Design, write and debug programs that accomplish specific goals • Plan, create, test and modify algorithms to solve open ended problems using a variety of programmable devices. • Use more advanced programming, including penup/pendown, and repeat commands to create, test, debug, modify and refine algorithms and programs.
5	• Create and refine sequences of commands using logo programming, including the use of procedures e.g. to construct, and investigate geometric patterns and problems • Refine sequences of commands to control outputs only e.g. lighting sequences, buzzers and motors (this could include screen simulation or real devices) • Make predictions regarding the consequences of decisions when creating sequences of commands
6	• Plan, create, modify and refine control sequences which use inputs and outputs e.g. using if... then... commands to control events taking account of purpose and needs • Devise, test and refine more effective control sequences incorporating conditional statements, procedures and sub-routines, taking account of purpose and needs

Naace age related expectations by year

7	• Devise, test and refine solutions by using modular approaches • Use programmable devices with general purpose input and outputs to solve problems
8	• Embed scripting in web based solutions • Use simple computational abstractions to build models of real world processes
9	• Use general purpose text based languages in environments capable of supporting stand alone applications

Key Methodologies



Prospective chefs are not expected to learn their craft only by listening to lectures and reading recipe books. Similarly, students need to do more than passively watch demonstrations of how to use programming languages.

So how should they learn?

Clearly, there needs to be some direct skills teaching, and Naace believes this is best done within the context of code which is an existing solution to a problem. Make collections of programs appropriate to the progression guidelines above. The first step is for students to know how to make the programming environment work. Students should run the programs. The next step is to understand which parts of the code produce which outputs. Encourage them to change the program and observe what happens. The next step is to set related problems, which they can solve using solutions they have seen. Finally, set them new problems, where they are free to write their own code, or use code acquired from elsewhere. At all times be open to learning from the students; they often know far more about language functionality that you haven't had time to explore, so embrace their knowledge and expertise.

Self-review should be an ongoing process. Encourage positive and constructive peer-review by projecting onto the interactive whiteboard student work-in-progress for discussion, and sharing of solutions expressed in sequences of instructions. Always compare the instructions to the outputs. Having listened to the feedback, students are able to make informed decisions to improve their work further.

Ensure that the programs produced are written to be used by others, with no programming experience.

When ready, the finished artefact should be published. Confidence and competence are of course different, but the growth of each is usually linked to the other. Confidence often comes with positive praise from a genuine audience. Celebrate your students' growing competence by posting their work on your class blog, or photograph the work and Tweet about it.

The sequence of skills acquisition, develop competence, review, improve, publish can be followed with any strand of the Computing curriculum, not just programming, with any new set of skills, and with any age group.

Resources

2Simple's Purple Mash contains a coding environment, 2Code. www.2simple.com/2Code

Arduino – more information is available at www.arduino.cc

Bury Primary Computing Solution is a primary computing scheme of work. More information is available here http://tcsc.primaryblogger.co.uk/2014/12/12/bury-primary-computing-solution/

Codebug and the BBC micro:bit are programmable physical systems. See www.codebug.org.uk and www.microbit.co.uk

Computing at School (CAS, https://www.computingatschool.org.uk/) is a community devoted to the teaching and learning of Computer Science. It is free to join and is packed with all sorts of good things.

Flowol is a programming environment where the coding is done as a flowchart. www.flowol.com

j2code www.j2e/j2code is a collection of three free programming environments, which comprise a scratch-like blocks programming environment, a logo-like environment, and a version of the BBC Micro:bit environment

Lucidchart is an online tool for drawing flowcharts www.lucidchart.com

Quitebasic is a free online tool for writing simple programs in Basic. See www.quitebasic.com

Raspberry Pi – more information is available at www.raspberrypi.org/education/

Scratch is free and it is really useful. It comes in three versions; Scratch 1.4 https://scratch.mit.edu/scratch_1.4 and Scratch 2 https://scratch.mit.edu/scratch2download to download, and Scratch 2 online https://scratch.mit.edu/projects/editor/?tip_bar=getStarted There is a full implementation of Scratch 1.4 for the iPad. It is called Pyonkee. There is also a Scratch Junior app for iOS.

Naace has produced this series of Essential Guides, or "eGuides" in response to an identified gap in teachers' CPD. That is, how to teach programming in the Computer Science strand of the Computing Programme of Study.

Furthermore, Naace believes that technology has a major role to play in raising standards in learning across the curriculum, provided teachers know how to adapt their pedagogies in order to maximise the potential gains offered by learning technologies.

Sponsored by Plum Innovations

For further information please contact
tim@learninglinked.co.uk

Cover artwork by Gaia Technologies

Having read this "eGuide", you may wish to register with The Naace Open Badge Academy for Open Badge CPD accreditation