Programming

Introduction to programming

What is programming?

Programming is the process of designing and writing a set of instructions (a program) for a computer in a language it can understand.

This can be really simple, such as the program to make a robot toy trace out a square; or it can be incredibly sophisticated, such as the software used to forecast the weather or to generate a set of ranked search results.

Programming is a two-step process.

- First, you need to analyse the problem or system and design a solution. This process will use logical reasoning, decomposition, abstraction and **generalisation** (see pages 6–17) to design algorithms to solve the problem or model the system.
- Secondly, you need to express these ideas in a particular programming language on a computer. This is called coding, and we can refer to the set of instructions that make up the program as 'code'.

Programming provides the motivation for learning computer science – there's a great sense of achievement when a computer does just what you ask it, because you've written the precise set of instructions necessary to make something happen. Programming also provides the opportunity to test out ideas and get immediate feedback on whether something works or not.

What should programming be like in schools?

It's possible to teach **computational thinking** without coding and vice versa, but the two seem to work best hand-in-hand.

Teaching computational thinking without giving pupils the opportunity to try out their ideas as code on a computer is like teaching science without doing any experiments. Similarly, teaching coding without helping pupils to understand the underlying processes of computational thinking is like doing experiments in science without any attempt to teach pupils the principles which underpin them.

This is reflected in the new computing curriculum, which states that pupils should not only know the principles of information and computation, but should also be able to put this knowledge to use through programming. One of the aims of the national curriculum for computing is that pupils can analyse problems in computational terms, and have repeated practical experience of writing computer programs in order to solve problems.

In key stage 1, pupils should be taught *how* simple **algorithms** are implemented as programs on digital devices. The phrase 'digital devices' encompasses tablets, laptop computers, **programmable toys**, and perhaps also distant **web servers**. It can be useful for pupils to be able to see their algorithms, in whatever way they've recorded these, and their code side by side.

Children can use simple arrow cards to record algorithms for programmable toys.

Pupils also should have the opportunity to create and **debug** (see pages 28–29) their own **programs**, as well as predicting what a program will do.

In key stage 2, pupils should be taught to design and write programs that accomplish specific goals, which should include controlling or simulating physical systems, for example making and programming a Lego robot. They should be taught to use **sequence**, **selection** and **repetition** in their programs (see pages 24–27), as well as variables to store **data**. They should also learn to use logical reasoning to detect and fix the errors in their programs.

Classroom activity ideas

- There are simple activities on the Barefoot Computing website; see Further Resources below.
- Here are some ideas for extended programming projects:
  - Year 2: solve a maze using a floor/screen turtle
  - Year 3: create a simple animation
  - Year 4: create a question and answer maths game
  - Year 5: create more complex computer games
  - Year 6: develop a simple app for a tablet or smartphone.

Further resources

- Barefoot on 'Programming', available at: http://barefootcas.org.uk/barefoot-primary-computing-resources/concepts/programming/ (free, but registration required).
- BBC Bitesize: Controlling physical systems, available at: www.bbc.co.uk/guides/zxjsfg8.
- BBC Cracking the Code, for examples of source code for complex software systems such as robot footballers and a racing car simulator, available at: www.bbc.co.uk/programmes/p01661pj.
- CAS Chair, Prof Simon Peyton Jones' explanation of some of the computer science that forms the basis for the computing curriculum: http://community.computingatschool.org.uk/resources/2936.
- Code.org for activities and resources, available at: http://code.org/educate.
- Rushkoff, D., *Program or be Programmed: Ten Commands for a Digital Age* (OR Books, 2010).

How do you program a computer?

Programming a computer involves writing code. The code is the set of instructions for the computer written in a programming language that the computer understands. In fact, the programming languages we use are a halfway house – they're written in a language we can understand which then gets translated by the computer into the 'machine code' of instructions that can be run directly on the silicon chips which control it.

Programs comprise precise, unambiguous instructions – there's no room for interpretation or debate about the meaning of a particular line of computer code. We can only write code using the clearly defined vocabulary and grammar of the programming language, but typically these are words taken from English, so code is something that people can write and understand, but the computer can also follow.

What programming languages should you use?

There are many languages to choose from. The majority are more complex than necessary for those just getting to grips with the ideas of programming, but there are plenty of simple, well supported languages that can be used very effectively in the primary classroom. Try to pick a language that you'll find easy to learn, or better still, know already.

Consider these points when choosing a programming language:

- Not all languages run on all computer systems.
- Choose a language that is suitable for your pupils. There are computer languages that are readily accessible to primary pupils – in most cases this will mean one that has been written with pupils in mind, or at least adapted to make it easier to learn.
- Choose a language supported by a good range of learning resources. It's better still if it has online support communities available, both for those who are teaching the language and those who are learning it.
- It is beneficial to the pupils if they can continue working in the language on their home computer, or, even better, if they can easily continue work on the same project via the internet.

There's a view that some languages are better at developing good programming 'habits' than others. Good teaching, in which computational thinking is emphasised alongside coding, should help to prevent pupils developing bad coding habits at this stage.

Which language is right for which key stage?

The table below illustrates a progressive approach to programming languages in a primary setting.

Key stage	Language type	Language	See
Early Years/KS1	Device-specific	Bee-Bot	Page 20
		Roamer-Too	Page 20
KS1	Limited instruction	ScratchJr	Page 22
		Lightbot™	Pages 20–21
KS2	Game programming	Kodu	Pages 21, 22, 25 and 28
	Block-based	Scratch	Pages 21–22, 24–28
	Text-based	Logo	Pages 22, 26–27, 29
		TouchDevelop	Page 23

Whilst there's much to be said for letting pupils explore several programming languages, it's important that they develop a degree of fluency in one, fairly general-purpose language, so that this becomes a medium in which they can solve problems, get useful things done and work creatively.

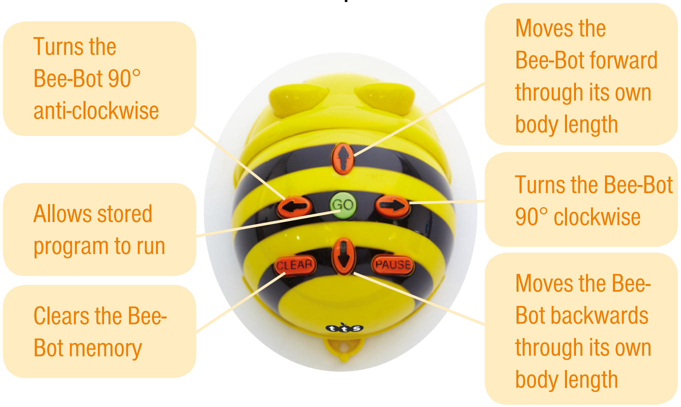
Further resources

- Iry's 'brief, incomplete and mostly wrong history of programming languages': http://james-iry.blogspot.mx/2009/05/brief-incomplete-and-mostly-wrong.html.
- Utting, I., Cooper, S., Kolling, M., Maloney, J. and Resnick, M., (2010) 'Alice, Greenfoot, and Scratch – A Discussion', available at: http://kar.kent.ac.uk/30617/2/2010-11-TOCE-discussion.pdf.
- Wikipedia list of 'Hello, world!' in many programming languages, available at: http://en.wikipedia.org/wiki/List_of_Hello_world_program_examples.

How do you program a floor turtle?

Programming in Early Years and key stage 1 is much more likely to involve working with simple programmable toys than using computers. It's much easier for pupils to learn the idea of programming when working with a really simple language and **interface**, and for them to plan and check their programs when they can, quite literally, put themselves in the place of the device they're programming.

A programmable floor turtle, such as the Bee-Bot or Roamer-Too, is ideal for this. The Bee-Bot programming language consists of five commands: forward, back, turn left, turn right and pause. Programming a Bee-Bot is simply a process of pressing buttons in the desired order to build a sequence of commands, with new commands being added to the end of the sequence.


Turns the Bee-Bot 90° anti-clockwise
Allows stored program to run
Clears the Bee-Bot memory
Moves the Bee-Bot forward through its own body length
Turns the Bee-Bot 90° clockwise
Moves the Bee-Bot backwards through its own body length
GO
CLEAR
PAUSE

This simple device can be used as a basis for many engaging activities, both for early programming and across the curriculum. Younger pupils will often work with the Bee-Bot one instruction at a time, whilst older children will become adept at creating longer sequences of instructions.

A number of tablet or smartphone apps and web-based tools are based on the idea of device-specific languages like these. These are often in the form of a game with a sequence of progressively harder levels in which players create ever more complex sequences of instructions to solve challenges. For example: Bee-Bot, Lightbot™, A.L.E.X and Cargo-Bot.

One approach for scaffolding the transition from floor turtle programming to programming on screen is to use an on-screen **simulation** of a Bee-Bot: it's relatively easy to make (or adapt) one yourself in Scratch 2.0.

Classroom activity ideas

- Allow very young pupils to play with a floor turtle, tinkering with it so they can develop their own sense of the relationship between pressing buttons and running their program.
- Encourage pupils to plan a sequence of instructions for a particular objective, such as getting the floor turtle from one ‘flower’ to another. Ask pupils to predict what will happen when they run their program, and to explain their thinking (logical reasoning).
- For more complex challenges, provide pupils with the code for a floor turtle’s route from one place to another, including an error in the code. Ask the pupils to work out where the bug is in the code and then fix this, before testing out their code on the floor turtle.

Further resources

- BBC on how to program a robot, available at: www.bbc.co.uk/learningzone/clips/programming-robots/4391.html.
- Bee-Bots are available from TTS. Other programmable toys include Roamer-Too (by Valiant) and Pro-Bot (by TTS).
- Barefoot on ‘KS1 Bee-Bots, 1, 2, 3 Programming Activity’, available at: http://barefootcas.org.uk/barefoot-primary-computing-resources/concepts/programming/ks1-bee-bots-12-3-programming-activity/ (free, but registration required).
- Bee-Bot and Roamer-Too simulator activities, available at: http://scratch.mit.edu/projects/19799927/.
- Lightbot™, available at: http://lightbot.com/.

How do you program things to move around the screen?

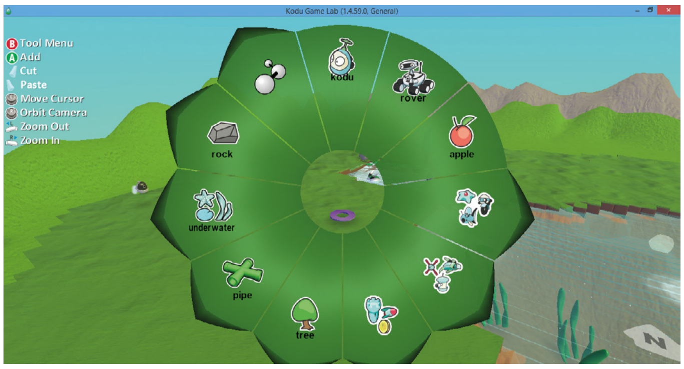
There are a number of graphical programming toolkits available: these make learning to code easier than ever. In most of these, programs are developed by dragging or selecting blocks or icons which represent particular instructions in the programming language. These can normally only fit together in ways that make sense, and the amount of typing, and thus the potential for spelling or punctuation errors, is kept to an absolute minimum.

With toolkits like these it’s easy to experiment with creating code. By letting the programmer focus on the ideas of their algorithm rather than the particular vocabulary and grammar of the programming language, learning to program becomes easier and often needs less teacher input.

Kodu

Microsoft’s Kodu is a rich, graphical toolkit for developing simple, interactive 3D games.

Each object in the Kodu game world can have its own program. These programs are ‘event driven’: they are made up of sets of ‘when [this happens], do [that]’ conditions, so that particular actions are triggered when certain things happen, such as a key being pressed, one object hitting another, or the score reaching a certain level.

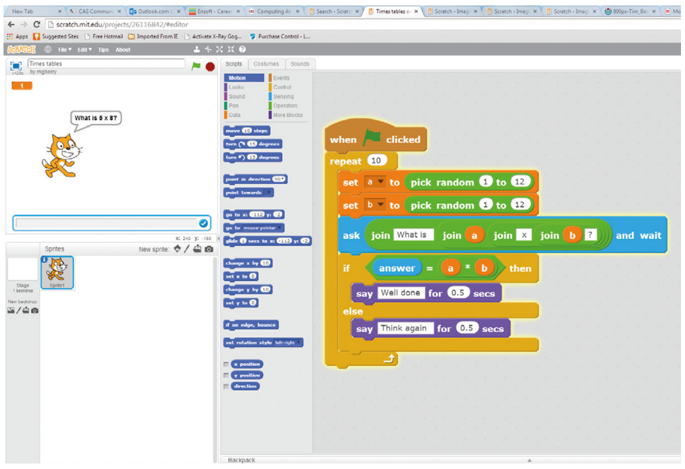


Kodu interface.

Programmers can share their games with others in the Kodu community, which facilitates informal and independent learning. There’s also plenty of scope for pupils to download and modify games developed by others, which many find quite an effective way to learn the craft of programming. This can also offer pupils a sense of creating games with an audience and purpose in mind.

Scratch

In MIT’s Scratch, the programmer can create their own graphical objects, including the stage background on which the action of a Scratch program happens, and a number of moving objects, or **sprites**, such as the characters in an animation or game.



```
when [flag] clicked
repeat 10
    set a to pick random 1 to 12
    set b to pick random 1 to 12
    ask join What is join a join x join b ? and wait
    if answer = a * b then
        say Well done for 0.5 secs
    else
        say Think again for 0.5 secs
```

Screenshot of a Scratch program.

Each object can have one or more scripts, built up using the building blocks of the Scratch language. To program an object in Scratch, you drag the colour-coded block you want from the different palettes of blocks and snap this into place with other blocks to form a script. Scripts can run in parallel with one another or be triggered by particular events, as in Kodu.

A number of other projects use Scratch as a starting point for their own platforms, for example ScratchJr is an iPad app designed for young programmers (key stage 1) and Berkeley's Snap! allows even more complex programming ideas (such as functions) to be explored through the same sort of building block interface.

There's a great online community for Scratch developers to download and share projects globally, making it easier for pupils to pursue programming in Scratch far beyond what's needed for the national curriculum. There's also a supportive educator community, which has developed and shared high quality curriculum materials.

Scratch is available as a free web-based editor or as a standalone desktop application. Files can be moved between online and offline versions.

Classroom activity ideas

- Pupils could develop a game in Kodu, taking inspiration from some of the games on the Kodu community site. As a starting point, tell them to create a game in which Kodu (the player's avatar in the game) is guided around the landscape bumping into (or shooting) enemies.
- Ask your pupils to create a simple scripted animation in Scratch, perhaps with a couple of programmed characters who take turns to act out a story. Designing the algorithm for a program like this is very similar to storyboarding in video work.

Further resources

- Armoni, M. and Ben-Ari, M., 'Computer science concepts in Scratch', available at: http://stwww.weizmann.ac.il/g-cs/scratch/scratch_en.html.
- Berry, M., 'Scratch across the curriculum', available at: http://milesberry.net/2012/06/scratch-across-the-curriculum/.
- Creative Computing, 'An Introductory Computing Curriculum Using Scratch', available at: http://scratched.gse.harvard.edu/guide/.
- Kelly, J., *Kodu for Kids* (Que Publishing, 2013).
- Kinect2Scratch, to program Microsoft Kinect with Scratch, available at: http://scratch.saorog.com/.
- Kodu Game Lab Community, available at: www.kodugamelab.com/.
- Other graphical programming environments for education include Espresso Coding, 2Code from 2Simple and J2Code.
- Scratch, available at http://scratch.mit.edu/.
- ScratchEd online community for educators, available at: http://scratch.mit.edu/educators/.
- ScratchJr: www.scratchjr.org/.
- Snap!, available at: http://snap.berkeley.edu/.

What is *real* programming?

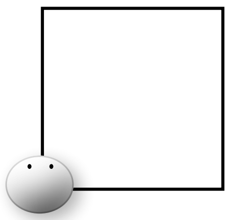
Most software development in academia and industry takes place using text-based languages, where programs are constructed by typing the commands from the programming language at a keyboard.

Historically, text-based programming has been a real barrier for children when learning to code, and there's no need to rush into text-based programming as part of the primary curriculum. It is, however, worth considering text-based programming for an extra-curricular programming club or even in class, if you or your colleagues feel confident with this. Possible text-based programming languages for primary schools could include Logo and TouchDevelop.

Logo

Logo was developed by Seymour Papert and others at MIT as an introductory programming language for children. It's probably best known for its use of 'turtle graphics' – an approach to creating images in which a 'turtle' (either a robot or a representation on screen) is given instructions for drawing a shape, such as:

```
REPEAT 4 [
      FORWARD 100
      RIGHT 90 ]
```



Papert saw Logo as a tool for children to *think* with, just as programming is both the means to and motivation for computational thinking.

In Logo programming, more complex programs are built up by 'teaching' the computer new words. These are called procedures. For example: defining a procedure to draw a square of a certain size using the key words of the language. Once you have defined the procedure 'square', typing it in will then result in the turtle drawing a square. For example:

```
TO SQUARE :SIDE
      REPEAT 4 [
            FORWARD :SIDE
            RIGHT 90 ]
      END
SQUARE 50
```

TouchDevelop

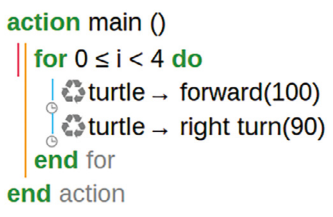
Typing code on a tablet computer or a smartphone is not easy, and this can be problematic for schools that use these devices extensively.

Developed by Microsoft Research, TouchDevelop is a programming language and environment, which takes into account both the challenges posed and the opportunities offered by touch-based interfaces such as those on tablets and smartphones.

TouchDevelop makes it quite easy to develop an app for a smartphone or tablet on the smartphone or tablet itself.

Although TouchDevelop is a text-based language, programmes aren't *typed* but are created by choosing commands from the options displayed in a menu system. In this way, TouchDevelop is a halfway house between graphical and text-based programming.

As with Logo, turtle graphics commands are available as standard. On many platforms TouchDevelop can also access some of the additional **hardware** built into the device, such as the accelerometer or GPS location, allowing more complex apps to be developed: these can be hosted online as web-based apps or installed directly on the device if it's a Windows phone.



```
action main ()
  for 0 ≤ i < 4 do
    turtle → forward(100)
    turtle → right turn(90)
  end for
end action
```

Program to draw a square using a turtle.

A particularly nice feature of TouchDevelop is the use of interactive tutorials to scaffold pupils' learning of the language.

Classroom activity ideas

- Revisit the turtle graphics activities you might have been using for programming in the past.
- Explore how different programming languages can be used to simulate dice being rolled. First, ask pupils to think about how they would do that in Scratch. Then, challenge your pupils to create an app in TouchDevelop which simulates rolling a dice when the phone or tablet is shaken, or when the screen is tapped. Ask pupils to think about how deterministic computers can simulate random events such as these.

Further resources

- Archived lesson plan from DfES for creating crystal flowers: http://webarchive.nationalarchives.gov.uk/20090608182316/http://standards.dfes.gov.uk/pdf/primaryschemes/itx4e.pdf.
- Horspool, N. and Ball, T., *TouchDevelop: Programming on the Go* (APress, 2013), available at: www.touchdevelop.com/docs/book.
- Logo, available at: www.calormen.com/jslogo/ and elsewhere.
- Papert, S., *Mindstorms: Children, Computers, and Powerful Ideas* (Basic Books Inc., 1980), available at: http://dl.acm.org/citation.cfm?id=1095592.
- TouchDevelop interactive tutorials for Hour of Code™: www.touchdevelop.com/hourofcode2.
- TouchDevelop from Microsoft Research: www.touchdevelop.com/.

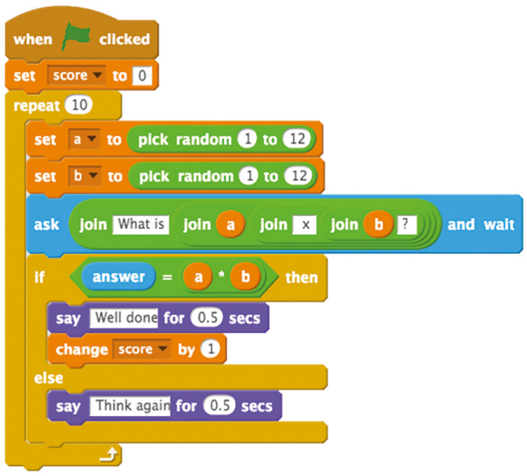
What's inside a program?

Whilst the detail will vary from one language to another, there are some common structures and ideas which programmers use over and over again from one language to another and from one problem to another:

- Sequence: running instructions in order (see below and page 25)
- Selection: running one set of instructions or another, depending on what happens (see pages 25–26)
- Repetition: running some instructions several times (see pages 26–27)
- Variables: a way of storing and retrieving data from the computer's memory (see pages 27–28).

These are so useful that it's important to make sure all pupils learn these.

This Scratch **script** shows sequence, selection, repetition and variables. Can you work out which bit is which before we look at these ideas in detail?



```
when ⚑ clicked
set score to 0
repeat 10
    set a to pick random 1 to 12
    set b to pick random 1 to 12
    ask join What is join a join x join b ? and wait
    if answer = a * b then
        say Well done for 0.5 secs
        change score by 1
    else
        say Think again for 0.5 secs
```

Further resources

- BBC Bitesize programming tutorial 'How do we get computers to do what we want?' (covering sequence, selection and repetition), available at: www.bbc.co.uk/guides/z23q7ty.
- Cracking the Code clip, available at: www.bbc.co.uk/programmes/p016j4g5.
- Scratch multiplication test, available at: http://scratch.mit.edu/projects/26116842/#editor.

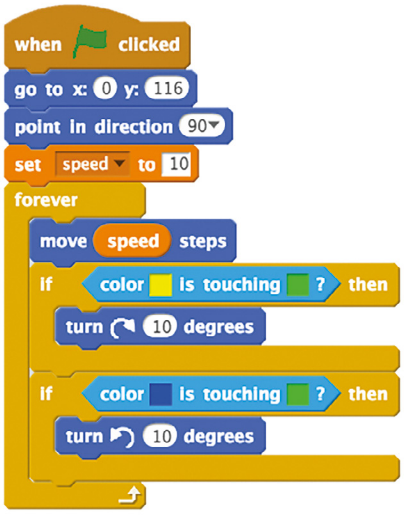
Sequence

Programs are built up of sequences of instructions. When pupils start programming with floor turtles, their programs consist entirely of sequences of instructions, built up as the stored sequence of button presses for what the floor turtle should do. As with any program, these instructions are precise and unambiguous, and the floor turtle will simply take each instruction (the stored button presses) and turn that into signals for the motors driving its wheels.

Initially, pupils might type in just one instruction at a time, clearing the memory after each, but as they become more experienced as programmers, or want to solve a problem more quickly, sequences become more complex.

```
Forward
Forward
Forward
Turn left
Forward
Forward
```

Pupils' first Scratch programs are also likely to be made up of simple sequences of instructions. Again, these need to be precise and unambiguous, and of course the order of the instructions matters. In developing their algorithms, pupils will have to work out exactly what order to put the steps in to complete a task.



```
when ⚑ clicked
go to x: 0 y: 116
point in direction 90
set speed to 10
forever
    move speed steps
    if color 🟨 is touching 🟩 ? then
        turn ↻ 10 degrees
    if color 🟦 is touching 🟩 ? then
        turn ↺ 10 degrees
```

A program that children might create in Scratch.

Classroom activity ideas

- Give pupils progressively more complex problems to solve with a floor turtle, asking them first to plan their algorithm for solving these before creating single programs on the floor turtle.
- Provide pupils with existing projects from Scratch (see Further resources on page 26). Allowing them to remix these projects by changing the code and seeing how this affects the program is a useful learning experience.
- Ask pupils to design, plan and code scripted animations in Scratch, perhaps using a timeline or storyboard to work out their algorithm before converting this into instructions for sprites in Scratch.

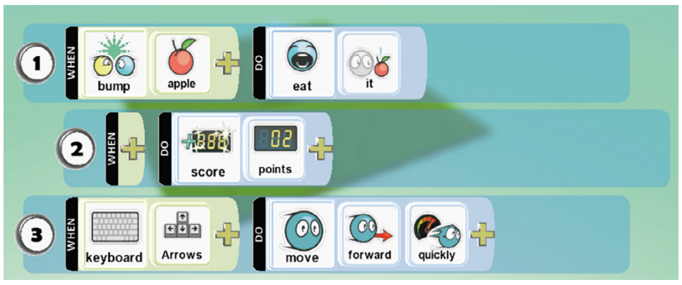
Further resources

- Animation 14: UK Schools Computer Animation Competition (key stage 2), available at: http://animation14.cs.manchester.ac.uk/gallery/winners/KS2/.
- Barefoot on 'Sequence', available at: http://barefootcas.org.uk/barefoot-primary-computing-resources/concepts/programming/sequence/ (free, but registration required).
- Cracking the Code clip on programming a robotic toy car: www.bbc.co.uk/programmes/p01661yg.
- Viking invasion animation in Scratch from Barefoot Computing (for upper KS2), available at: http://barefootcas.org.uk/programme-of-study/use-sequence-in-programs/upper-ks2-viking-raid-animation-activity/ (free, but registration required).

Selection

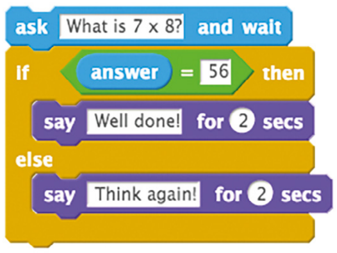
Selection is the programming structure through which a computer executes one or other set of instructions according to whether a particular condition is met or not. This ability to do different things depending on what happens in the computer as the program is run or out in the real world lies at the heart of what makes programming such a powerful tool.

Selection is an important part of creating a game in Kodu. An object's behaviour in a game is determined by a set of conditions, for example: WHEN the left arrow is pressed, the object will move left. Similarly, interaction with other objects, variables and environments in Kodu are programmed as a set of WHEN … DO … conditions. For example, WHEN I bump the apple DO eat it AND add 2 points to score.



```
1  WHEN bump apple +   DO eat it
2    WHEN +   DO score 02 points +
3  WHEN keyboard Arrows +   DO move forward quickly +
```

In Scratch (and other programming languages) you can build selection into a sequence of instructions, allowing the computer to run different instructions depending on whether a condition is met.

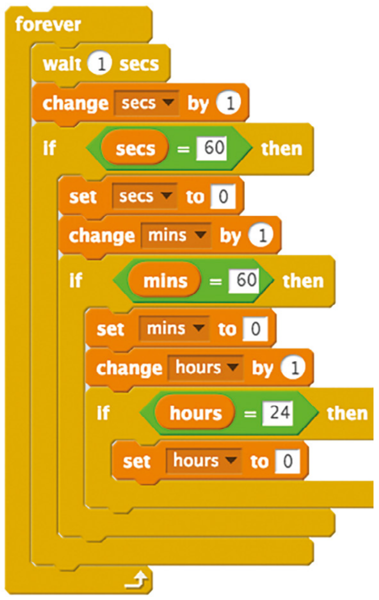


```
ask [What is 7 x 8?] and wait
if <(answer) = [56]> then
    say [Well done!] for (2) secs
else
    say [Think again!] for (2) secs
```

Examples of how selection can be used to start a script in Scratch.

At the core of many educational games is a simple selection command: if the answer is right then give a reward, else say the answer is wrong. See the Scratch script for the times tables game on page 24.

It's also worth noting that selection statements can be nested inside one another. This allows more complex sets of conditions to be used to determine what happens in a program. Look at the way some *if* blocks are inside others in the following script to model a clock in Scratch. The script also uses repetition and three variables for the seconds, minutes and hours of the time:



```
forever
    wait (1) secs
    change [secs] by (1)
    if <(secs) = [60]> then
        set [secs] to [0]
        change [mins] by (1)
        if <(mins) = [60]> then
            set [mins] to [0]
            change [hours] by (1)
            if <(hours) = [24]> then
                set [hours] to [0]
```

Classroom activity ideas

- Encourage pupils to explore the different conditions which the character in Kodu can respond to in its event-driven programming. Get pupils to think creatively about how they might use these when developing a game of their own. Give them time to design their game, thinking carefully about the algorithm, i.e. the rules, they're using.

- Ask pupils to design simple question and answer games in Scratch. Encourage them to first think about the overall algorithm for their game before coding this and then working to develop the user interface, making this more engaging than just a cat asking lots of questions. It's helpful if pupils have a target audience in mind for software like this.

Further resources

- Barefoot Computing on 'Selection', available at: http://barefootcas.org.uk/programme-of-study/use-selection-programs/selection/ (free, but registration required).
- Papert, S., 'Does Easy Do It? Children, Games, and Learning', available at: www.papert.org/articles/Doeseasydoit.html.

Scratch projects to remix

- Analogue clock by mgberry on Scratch, available at: http://scratch.mit.edu/projects/28742256/#editor.
- Addition race by mgberry on Scratch, available at: http://scratch.mit.edu/projects/15905989/#editor.

Repetition

Repetition in programming means to repeat the execution of certain instructions. This can make a long sequence of instructions much shorter, and typically easier to understand.

Using repetition in programming usually involves spotting that some of the instructions you want the computer to follow are the same, or very similar, and therefore draws on the computational thinking process of pattern recognition/generalisation (see pages 15–16). You'll sometimes hear the repeating block of code referred to as a loop, i.e. the computer keeps looping through the commands one at a time as they're executed (carried out).

Think about the Bee-Bot program for a square `(forward, left, forward, left, forward, left, forward, left)`. Notice how for each side we move forward and then turn left. On a Roamer-Too or a Pro-Bot, you could use the repeat command to simplify the coding for this by using the built in repeat command, replacing this code with, for example, `repeat 4 [forward, left]`.

The same would apply in Logo, from which the Roamer-Too and Pro-Bot programming device-specific languages are derived.

Compare:

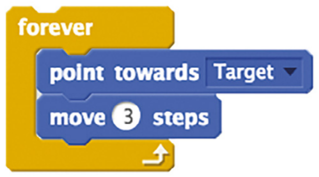
```
FORWARD 100
LEFT 120
FORWARD 100
LEFT 120
FORWARD 100
LEFT 120
```

with:

```
REPEAT 3 [
  FORWARD 100
  LEFT 120 ]
```

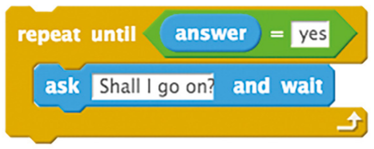
Both programs draw equilateral triangles. Using repetition reduces the amount of typing and makes the program reflect the underlying algorithm more clearly.

In the examples above, the repeated code is run a fixed number of times, which is the best way to introduce the idea. You can also repeat code forever. This can be useful in real world systems, such as a control program for a digital thermostat, which would continually check the temperature of a room, sending a signal to turn the heating on when this dropped below a certain value. This is a common technique in game programming. For example, the following Scratch code would make a sprite continually chase another around the screen:



```
forever
  point towards Target
  move 3 steps
```

Repetition can be combined with selection, so that a repeating block of code is run as many times as necessary until a certain condition is met, as in this fragment in Scratch:

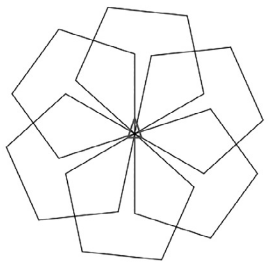


```
repeat until answer = yes
  ask Shall I go on? and wait
```

You can nest one repeating block inside another. The 'crystal flower' programs in Logo use this idea. For example:

```
REPEAT 6 [
        REPEAT 5 [
                FORWARD 100
                LEFT 72 ]
        LEFT 60 ]
```

draws:



Classroom activity ideas

- Ask pupils to use simple repetition commands to produce a ‘fish tank’ animation in Scratch, with a number of different sprites each running their own set of repeating motion instructions. This can be made more complex by including some selection commands to change the behaviour of sprites as they touch one another.
- Encourage pupils to experiment with ‘crystal flower’ programs in Scratch, Logo or other languages that support turtle graphics, and investigate the effect of changing the number of times a loop repeats as well as the parameters for the commands inside the loop. There are some great opportunities to link computing with spiritual, social and cultural education.

Further resources

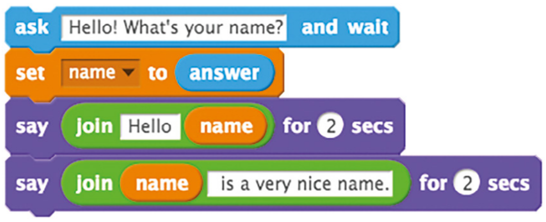
- Barefoot Computing on ‘Repetition’, available at: http://barefootcas.org.uk/programme-of-study/use-repetition-programs/repetition (free, but registration required).
- Digital Schoolhouse dance scripts, available at: www.resources.digitalschoolhouse.org.uk/key-stage-2-ages-7-10/218-scratch-teaching-dance.
- Scratch 2.0 Fishtank Game tutorial, available at: www.youtube.com/watch?v=-qTZ5bFEdC8.

Variables

Unlike the programming structures of sequence, selection and repetition, a variable is an example of a data structure. It is a simple way of storing one piece of information somewhere in the computer’s memory whilst the program is running, and getting that information back later. There’s a degree of *abstraction* involved – the actual detail of how the programming language, operating system and hardware manages storing and retrieving data from the memory chips inside the computer isn’t important to us as programmers, just as these details aren’t important when we’re using the clipboard for copying and pasting text. One way of thinking of variables is as labelled shoeboxes, with the difference that the contents don’t get removed when they’re used.

The concept of a variable is one that many pupils struggle with and it’s worth showing them lots of examples to ensure they grasp this. A classic example which pupils are likely to be familiar with, particularly from computer games, is that of score.

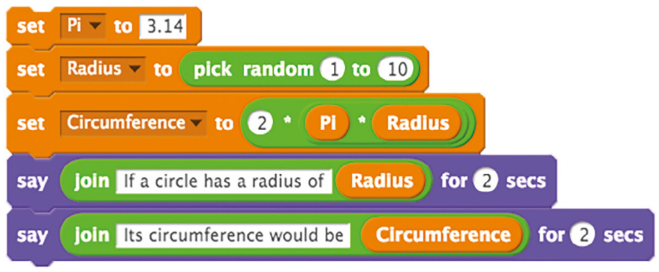
You can use variables to store data input by the person using your program and then refer to this data later on.



```
ask [Hello! What's your name?] and wait
set [name] to (answer)
say (join [Hello] (name)) for (2) secs
say (join (name) [is a very nice name.]) for (2) secs
```

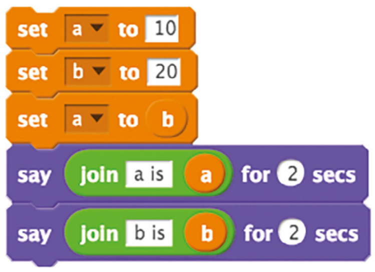
Here, name is a variable, in which we store whatever the user types in, and then use it a couple of times in Scratch’s response; `answer` is a special temporary variable used by Scratch to store for the time being whatever the user types in. Notice that variables can store text as well as numbers. Other types of data can be stored in variables too, depending on the particular programming language you’re working in.

Variables can also be created by the program, perhaps to store a constant value so that we can refer to it by name (`Pi` below), or the result of a computation (`Circumference` in the code below), or random numbers generated by the computer (for example `Radius` below):



```
set [Pi] to [3.14]
set [Radius] to (pick random (1) to (10))
set [Circumference] to ((2) * ((PI) * (Radius)))
say (join [If a circle has a radius of] (Radius)) for (2) secs
say (join [Its circumference would be] (Circumference)) for (2) secs
```

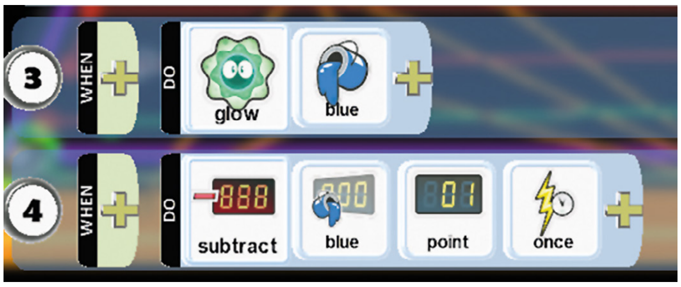
The idea that the contents of the ‘box’ are still there after the variable is used is sometimes a confusing one for those learning to program. Have a look at the following code and decide what will be displayed on the screen:



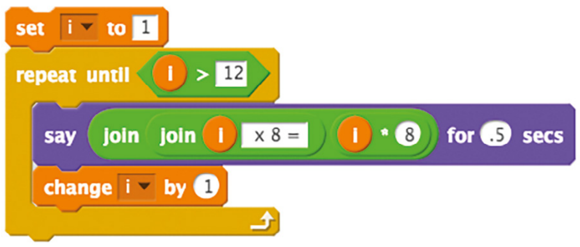
```
set [a] to [10]
set [b] to [20]
set [a] to (b)
say (join [a is] (a)) for (2) secs
say (join [b is] (b)) for (2) secs
```

You should see 'a is 20' followed by 'b is 20'. Try it!

In Kodu and other game programming, variables are useful for keeping track of rewards, such as a score, and for introducing some sort of limit, such as a time limit or health points that reduce each time you're hit. Kodu's event-driven approach allows particular actions to be done when variables reach a predetermined level.



One particularly useful example of variables in programming is as an iterator – this is a way of keeping track of how many times you've been round a repeating loop and of doing something different each time you do. To do this, we initialise a counter to zero or one at the beginning of the loop and then add one to it each time we go round the loop. For example, the following script would get Scratch to say its eight times table:



```
set i to 1
repeat until i > 12
    say join join i x 8 = i * 8 for .5 secs
    change i by 1
```

You can also use an iterator like this to work with strings (words and sentences) one letter at a time, or through lists of data one item at a time. Take care with the beginning and end, as it's all too easy to start or end too soon or too late with iterators.

Classroom activity ideas

- Get pupils to create a mystery function machine in Scratch, which accepts an **input**, stores this in a variable and then uses mathematical operators to produce an **output** shown on screen. Setting the display to full screen in Scratch, pupils can challenge one another (and you) to work out what the program does by trying different inputs.
- Pupils can use variables in their games programs, in say Scratch or Kodu, using a score to reward the player for achieving particular objectives (such as collecting apples), and imposing a time limit.

Further resources

- Bagge, P., 'Text Adventure Game' for Scratch, available at: http://code-it.co.uk/year4/text_adventure_game.pdf.
- Barefoot Computing on 'Variables', available at: http://barefootcas.org.uk/programme-of-study/work-variables/variables/ (free, but registration required).
- BBC Bitesize article 'How do computer programs use variables?', available at: www.bbc.co.uk/guides/zw3dwmn.
- Binary search jigsaw and solution by mgberry, available at: http://scratch.mit.edu/projects/20255402/ and http://scratch.mit.edu/projects/28907496/.
- How to program a Scratch 2.0 times table test, available at: www.youtube.com/watch?v=YHGyPfGg1x8.
- Notes and tutorial on variables in Scratch, available at: http://wiki.scratch.mit.edu/wiki/Variable and http://wiki.scratch.mit.edu/wiki/Variables_Tutorial.

Can we fix the code?

Errors in algorithms and code are called 'bugs', and the process of finding and fixing these is called 'debugging'. Debugging can often take much longer than writing the code in the first place. Whilst fixing a program so that it does work can bring a great buzz, staring at code that still won't work can be the cause of great frustration too: this can be tricky to manage in class.

The national curriculum for key stage 2 expects that pupils will be taught to **use logical reasoning to detect and correct errors in algorithms and programs**, so it's not really enough for pupils to fix their code without being able to give an explanation for what went wrong and how they fixed this.

In programming classes, pupils focused on the task of writing a program for a particular goal might want help from you or others to fix their programs: tempting as this may be, it's worth you and they remembering that the objective in class is not to get a working program, but to learn how to program – their ability to debug their own code is a big part of that.

What strategies can you use to support debugging?

One way that you can help is to provide a reasonably robust, general set of debugging strategies which pupils can use for any programming, or indeed more general strategies which they can use when they encounter problems elsewhere.

Debugging should be underpinned by logical reasoning. The Barefoot Computing team suggest a simple sequence of four steps, emphasising the importance of logical reasoning:

1. Predict what should happen.
2. Find out exactly what happens.
3. Work out where something has gone wrong.
4. Fix it.

One way to help predict what should happen is to get pupils to explain their algorithm and code to someone else. In doing so, it's quite likely that they'll spot where there's a problem in the way they're thinking about the problem or in the way they've coded the solution.

In finding out exactly what happens, it can be useful to work through the code, line by line. Seymour Papert described this as 'playing turtle'. So, in a turtle graphics program in Logo (or similar) pupils could act out the role of the turtle, walking and turning as they follow the commands in the language.

In working out where something has gone wrong, encourage pupils to look back at their algorithms before they look at their code. Before they can get started with fixing bugs, they'll need to establish whether it was an issue with their *thinking* or with the way they've implemented that as code.

Some programming environments allow you to step through code one line at a time – you can do this in Scratch by adding `(wait until [space] pressed)` blocks in liberally. Scratch will default to showing where sprites are and the contents of any variables as it runs through code, which can also be useful in helping to work out exactly what caused the problem.

Debugging is a great opportunity for pupils to learn from their mistakes and to get better at programming.

Classroom activity ideas

- Pupils are likely to make many authentic errors in their own code, which they'll want to fix. You might find that it's worth spending some time giving pupils some bugs to find and fix in other programs, both as a way to help develop strategies for debugging and to help with assessment of logical reasoning and programming knowledge. Create some programs with deliberate mistakes in, perhaps using a range of logical or semantic errors, and set pupils the challenge of finding and fixing these.
- Encourage pupils to debug one another's code. One approach is for pupils to work on their own program for the first part of the lesson and then to take over their partner's project, completing this and then debugging this for their friend.
- A similar paired activity is for pupils to write code with deliberate mistakes, setting a challenge to their partner to find and then fix the errors in the code.

Further resources

- Barefoot Computing on 'Debugging', available at: http://barefootcas.org.uk/barefoot-primary-computing-resources/computational-thinking-approaches/debugging/ (free, but registration required).
- BBC Bitesize 'What is debugging?', available at: www.bbc.co.uk/guides/ztkx6sg.
- Debugging challenges from Switched on Computing, available via: http://scratch.mit.edu/studios/306100/.
- Rubber duck debugging, available at: http://en.wikipedia.org/wiki/Rubber_duck_debugging.