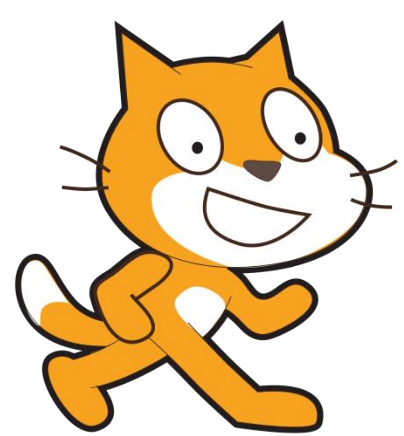
Starting from



An Introduction to Computing Science

by Jeremy Scott

TUTOR NOTES v2

Acknowledgements

This resource was partially funded by a grant from Education Scotland. We are also grateful for the help and support provided by the following contributors:

Cathkin High School
Linlithgow Academy
Perth High School
George Heriot's School
Stromness Academy
CompEdNet, Scottish Forum for Computing Science Teachers
Computing At School
Council of Professors and Heads of Computing (CPHC)
Professor Hal Abelson, MIT
Mitchel Resnick, MIT
Scottish Informatics and Computer Science Alliance (SICSA)
Edinburgh Napier University School of Computing
Glasgow Caledonian University School of Engineering and Built Environment
Heriot-Watt University School of Mathematical and Computer Sciences
Robert Gordon University School of Computing
University of Edinburgh School of Informatics
University of Aberdeen Department of Computing
University of Dundee School of Computing
University of Glasgow School of Computing Science
University of St Andrews School of Computer Science
University of Stirling Department of Computing Science and Mathematics
University of Strathclyde Department of Computer and Information Sciences
University of the West of Scotland School of Computing
International Olympic Committee
ScotlandIS
Turespaña
4J Studios
Brightsolid
Google
JP Morgan
Microsoft Research
Oracle
O2
RunRev
Sword Ciboodle

Special thanks go to Ian King who assisted with updating these materials for Scratch v2.0. The contribution of the following individuals who served on the RSE/BCS Project Advisory Group is also gratefully acknowledged: Professor Sally Brown (chair), Mr David Bethune, Professor Alan Bundy, Professor Quintin Cutts, Ms Kate Farrell, Mr William Hardie, Dr Fiona McNeill, Professor Greg Michaelson, Dr Bill Mitchell and Professor Judy Robertson.

Some of the material within this resource is based on existing work from the ScratchEd site, reproduced and adapted under Creative Commons licence. The author thanks the individuals concerned for permission to use and adapt their materials.

BCS is a registered charity: No 292786
The Royal Society of Edinburgh. Scotland's National Academy. Scottish Charity No. SC000470

Contents

Overview **1**
Introduction 1
Computational Thinking 2
Why Scratch? 3
Using this resource 3
Scratch **5**
Known Issues 6
Installation 6
Useful Resources 6
Lessons and approach **7**
Screencasts 7
Deep Understanding 7
Pair Programming 8
Suggested Activities 8
Inter-Disciplinary Learning 8
Introduction **9**
What is a computer? 9
Types of computer 10
Parts of a computer 11
Hardware 12
Software 13
Programming languages 14
Broader Inter-Disciplinary Learning 15
1: Scratching the Surface **17**
Lazy or smart? 25
2: Story Time **27**
Bugs 30
Event-driven programming 33
3: A Mazing Game **35**
The Importance of Design 36
4: Get the Picture? **45**
Nesting 47
5: Forest Archery Game **55**
Variables 58
Scratch Project **61**
Appendices **63**
Appendix A: Learner Tracking Sheet 64
Appendix B: Sample Code 65

Overview

Introduction

Implementation of Curriculum for Excellence (CfE) and the development of new National Qualifications presented a timely opportunity to revise the way Computing Science is taught in schools and to provide a more interesting, up-to-date and engaging experience for both tutors and learners.

This is version two of the first in a series of three resources developed by the Royal Society of Edinburgh and the BCS Academy of Computing that exemplify a subset of the computing science-related outcomes of CfE at Levels 3 & 4 and beyond.

This resource will seek to introduce learners to Computing Science via the Scratch 2.0 programming environment, developed at the Massachusetts Institute of Technology (MIT).

All three resources build on state-of-the-art understanding of the pedagogy of Computing, drawn from around the world. This should enable learners to develop both programming skills *and* deep understanding of core Computing concepts, including computational thinking (see overleaf).

Whilst this resource is intended to support tutors' thinking about how they might translate the intentions of the curriculum into classroom activity, it should not be seen as prescriptive. Rather, it is intended to stimulate innovation and offer tutors the flexibility and opportunity to deploy their creativity and skills in meeting the needs of learners.

Computational Thinking

Computational thinking is recognised as a key skill set for all 21st century learners – whether they intend to continue with Computing Science or not. It involves viewing the world through thinking practices that software developers use to write programs.

These can be grouped into five main areas:

- seeing a problem and its solution at many levels of detail (**abstraction**[1])
- thinking about tasks as a series of steps (**algorithms**)
- understanding that solving a large problem will involve breaking it down into a set of smaller problems (**decomposition**)
- appreciating that a new problem is likely to be related to other problems the learner has already solved (**pattern recognition**), and
- realising that a solution to a problem may be made to solve a whole range of related problems (**generalisation**).

Furthermore, there are some key understandings about computers:

- Computers are **deterministic**: they do what you tell them to do. This is news to many, who think of them as pure magic.
- Computers are **precise**: they do exactly what you tell them to do.
- Computers can therefore be **understood**; they are just machines with logical working.

Whilst computational thinking can be a component of many subjects, Computing Science is particularly well-placed to deliver it.

[1] Ultimately, if there is a "core truth" that underpins Computational Thinking, it is the idea of abstraction – "zooming in and out of a problem". Arguably, the four other areas of Computational Thinking could be viewed, to varying degrees, as different aspects of abstraction.

Why Scratch?

Since its launch, Scratch has received widespread acclaim as an ideal environment through which to introduce learners to computer programming and computational thinking.

Its building blocks approach all but eliminates a major problem for learners presented by traditional text-based languages i.e. the requirement to recall and type instructions according to a strict syntax.

In addition, its cartoon-style approach with emphasis on rich media types – sound, graphics and animation – have made it popular in the classroom as an engaging and entertaining way to introduce learners to Computing Science.

Using this resource

As well as lessons, exercises and sample answers, this book contains suggested supplementary activities and inter-disciplinary learning opportunities.

It is not the author’s intention that all of these are attempted; rather, they are simply suggestions as to the kind of activities that tutors may find useful in order to enrich learners’ experiences.

Feel free to use this resource as you wish:

- as part of an introduction to Computing Science within Curriculum for Excellence;
- to support aspects of the National 4 Software Design & Development unit.

Above all, these materials should not be seen as prescriptive. They merely contain guidance and suggestions as to the kinds of approach which can make learning more engaging whilst fostering computational thinking and greater understanding of Computing Science concepts in learners.

Curriculum for Excellence outcomes

This resource seeks to address the following outcomes within Curriculum for Excellence:

- Using appropriate software, I can work individually or collaboratively to design and implement a game, animation or other application. *TCH 3-09a*
- I can build a digital solution which includes some aspects of multimedia to communicate information to others. *TCH 3-08b*
- By learning the basic principles of a programming language or control technology, I can design a solution to a scenario, implement it and evaluate its success. *TCH 4-09a*
- I can integrate different media to create a digital solution which allows interaction and collaboration with others. *TCH 4-08c*
- I can create graphics and animations using appropriate software which utilise my skills and knowledge of the application. *TCH 4-09b*

Scratch

Scratch (**http://scratch.mit.edu/**) is a software development environment developed by MIT's Media Lab. It allows users with little or no experience of programming to create rich, multimedia projects which they can share with other users across the world via the Scratch website.

Scratch uses a graphical interface and in creating it, MIT drew upon significant prior research in educational computing which was informed by constructionist learning theories. These theories emphasise programming as a vehicle for engaging powerful ideas through active learning.

As a blocks-based programming language, Scratch all but eliminates a major problem for learners presented by traditional text-based languages i.e. the requirement to recall and type instructions according to a strict syntax.

Scratch takes its name from the practice of "scratching", a term lifted from hip-hop music where DJs reuse other artists' materials to create a new work. Web-based sharing and collaboration is also at the heart of Scratch.

At the time of writing, the latest version of Scratch was v2.0.

http://scratch.mit.edu

Known Issues

- Whilst Scratch 2.0 is available as an offline installation, it works best when used as a web app (http://scratch.mit.edu/). Consequently Scratch 2.0 is available on any platform via a modern browser.
- If using Scratch as an offline installation, Adobe AIR (Adobe Integrated Runtime) is required on the same computer.
- One of the most significant features of Scratch 2.0 is its support for user-defined modules, via the "Make a block" command. However, Scratch 2.0 still supports the "broadcast" command, which could be used in v1.4 to simulate user-defined modules via events.

Installation

Scratch requires a little preparation before using it in the classroom, but is a mature product that should present few problems.

The Scratch URL (http://scratch.mit.edu) should be whitelisted on any school proxy.

Above all, tutors are strongly recommended to test the software on a learner computer and account before trying to use Scratch with a class.

Useful Resources

There are numerous excellent sites for use with Scratch; however, some notable ones include:

http://scratch.mit.edu	the Scratch home page at MIT
http://scratched.gse.harvard.edu/	the Scratch Ed site for educators
http://www.scratch.ie	LERO Scratch site

Lessons and approach

Tutors using this resource may wish to consider the following approaches:

Screencasts

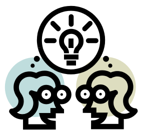
In addition to a traditional booklet, this resource makes use of screencasts to deliver some of the exemplar tutorials. Tutors wishing to use them may wish to use them on a whole-class basis and/or for learners individually, stopping and starting as they need.

The rationale behind using screencasts is that learners can make quicker progress, as it's more visual and immediate. Screencasts are also a medium that learners are familiar with in their own digital lives via services such as YouTube. Feedback from users of the first version of these materials suggests that screencasts have proved popular with the YouTube generation, allowing learners to progress at their own pace. Furthermore, they have been helpful for learners who may have missed classes or who are studying from home.

As the lessons progress, it is assumed that learners have gained an understanding of how to use the environment, so the use of screencasts as “scaffolding” is reduced.

Deep Understanding

To accompany the lessons, there are sample written and discussion tasks to enable learners and tutors to assess “deep understanding” of the Computing Science concepts. This draws upon recent work in the *CS Principles* course at the Universities of San Diego and Glasgow. Aspects of this approach are also seen in UC Berkeley’s *The Beauty and Joy of Computing* course.



Traditionally, tutors have inferred the degree of learners’ understanding of what they have learned from the programs they have produced; however, research has shown that this is not always a strong indicator. Consequently, consolidation via quizzes, group discussion, questioning and class work/homework should be used to enable the tutor to formatively assess the learners’ understanding throughout the course, rather than simply infer this from their completed programs. Discussion is strongly encouraged, as research suggests that both this and peer instruction can aid learners’ understanding.

Pair Programming

Collaborative learning is a cornerstone of Curriculum for Excellence and these materials encourage this, through pair programming. Pair programming can be defined as:

"… an agile software development technique in which two programmers work together at one workstation. One, the driver, types in code while the other, the observer (or navigator), reviews each line of code as it is typed in. The two programmers switch roles frequently.

While reviewing, the observer also considers the strategic direction of the work, coming up with ideas for improvements and likely future problems to address. This frees the driver to focus all of his or her attention on the "tactical" aspects of completing the current task, using the observer as a safety net and guide." **Source: Wikipedia**

Pair programming can encourage collaboration between learners, as well as making good use of available resources within the classroom.

It is recommended that tutors explicitly advise learners to seek help from a neighbour before asking the tutor for help. Tutors will, however, understand the need to ensure that both learners are equally engaged in the work!

Suggested Activities

In addition to the core activities in the leaner notes, further activities are suggested throughout the tutor notes. These should not be seen as prescriptive, but as possible ways to enrich a task or topic. Tutors are free to cherry-pick these as going through all of them is likely to significantly extend the unit.

Suggested Activity These opportunities are indicated by **Suggested Activity** in the left margin.

Inter-Disciplinary Learning

Scratch is a "rich" multimedia environment that offers ample opportunity for inter-disciplinary learning. Within the materials, there are suggestions for possible inter-disciplinary activities, as well as many of the activities being inter-disciplinary in themselves.

IDL Inter-Disciplinary Learning opportunities are indicated by **IDL** in the left margin.

Introduction

This section outlines what a computer is. It is up to tutors to decide whether to use this as an introduction or to embed the tasks within the practical lesson sequence.

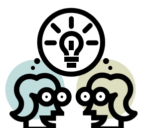
What is a computer?

Stress the ubiquity of computers and how their use is interwoven into numerous aspects of modern life. However, computers are machines that carry out instructions given by a human. Without instructions, computers wouldn't be able to do anything.

Stress the "strengths" and "weaknesses" of computers.

Activity

Write down three everyday tasks that humans perform but computers cannot (or are not very good at).



1. Have a conversation/tell a joke

2. Cook a meal

3. Look after a child

Suggested Activity

Allow learners to use an online chat bot e.g. A. L. I. C. E. (http://alice.pandorabots.com/) to demonstrate limitations of computers' conversational abilities. This could also provide an opportunity to discuss the Turing Test and the wider field of artificial intelligence (which is covered briefly in the second resource in this series "Itching for More").

Suggested Activity

Show video clips of current robots, such as NAO or Pepper from Aldebaran: http://www.aldebaran-robotics.com/en/

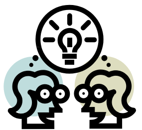
or Honda's Asimo robot trying to perform routine tasks – there are lots of Honda's Robotics YouTube channel: https://www.youtube.com/playlist?list=PLCF004AC7F3E2E811

However, some of the most informative examples show where robots such as Asimo fail to perform! There are many examples of this on YouTube and other video providers.

Types of computer

Learners will likely be familiar with all three categories of modern personal computers.

Activity



The personal computers shown above appear in order of oldest to newest types. What does this tell you about the kind of computers people want?

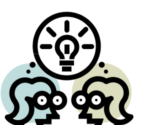
Learners will come up with a range of answers here. Some of the key points are:

- **portable**
- **easy to use**
- **don't take up much space**
- **connected to the Internet**

Learners may be less familiar with mainframes and servers. Many may not appreciate that a games console or smartphone is also a computer.

Many learners will not have heard the term "embedded computer". Spend some time going over this, eliciting from learners where they may find embedded systems.

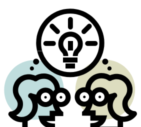
Activity



Write down <u>three</u> devices in your own home that you think might contain an embedded computer (besides those shown above).

Lots of examples e.g. DVD/Blu-Ray player, printer, cooker, microwave oven, dishwasher, alarm clock, radio, satellite/cable receiver

Activity



Write down <u>three</u> technologies that are combined in a modern smartphone.

Hardware technologies include: compass, accelerometer, GPS receiver, touch screen, Bluetooth radio, cellular radio, FM radio, loudspeaker, audio decoder.

There are also many software technologies.

Parts of a computer

This section deals with the input, process, output and storage of information as opposed to actual devices for each of these, which follow on from this.

Suggested activity **Get learners to consider the inputs, processes and output for non-computing devices**

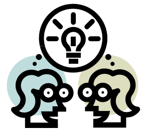
e.g. a washing machine

input = dirty clothes, water, soap powder, electricity

process = heat water, spin drum to wash clothes

output= dirty soapy water, clean clothes

Activity **Write down inputs and outputs for the following activities on different types of computers. When you have finished, create an extra one of your own:**

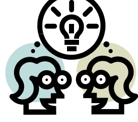


Activity	Input(s)	Output(s)
Playing a video game	Move game controller Click buttons	Character moves Menu selections made
Surfing the WWW	**Mouse clicks**	**Jumps to a new page**
Making a phone call	**Button presses**	**Dials number**
Watching TV	**Press buttons on remote control**	**Changes channel Changes volume**
Learner's own choice		

Hardware

Activity **Decide if the following devices are <u>input</u>, <u>output</u> or <u>storage</u> devices then put each one into the correct column. The first three have been done for you.**

keyboard; hard disc drive; monitor; speaker; scanner; printer; mouse; DVD drive; microphone; flash drive (memory stick)[2]; game controller; smartphone touch screen; memory card



Input Device	Storage Device	Output Device
keyboard	hard disc drive	monitor
scanner	DVD drive	speaker
mouse	flash drive/ memory stick	printer
microphone	memory card	game controller (vibration feedback)
touch screen		touch screen
game controller (buttons/ motion detector)		

Some learners are likely to incorrectly categorise a storage device as an input device because it provides information for the computer (or an output device because information is sent out to it when we save).

Stress that it is neither. A device such as a disc drive is a **storage device**: it stores information when the computer is switched off. Point out that the information it stores would *originally* have been entered into the computer using an input device.

[2] Whilst "flash drive" is the most widely recognised "correct" term, an accommodation has been made here for the commonly-used "memory stick".

Software

Stress that software is the key to making the hardware do something useful e.g.

Question: How many different things can most machines do?
Answer: Only one.

Although it may have different settings, a washing machine washes clothes. How about a kettle? Even a car? In general terms, they all do only one thing: wash clothes, heat water, transport people.

Question: How many different things can a computer do?
Answer: Lots – depending on the software it's running.

This ability to follow different instructions is what makes computers different from other machines.

Activity **Complete the table below of ten different jobs you can do on a computer and the name of a software package that lets you do it.**

Task	Software package
Browse the World Wide Web	Google Chrome
Play games	Angry Birds
Edit a movie	iMovie
Write an essay	Microsoft Word
Keep in touch with friends	Facebook
Record music	Steinberg Cubase
Create a newsletter	Microsoft Publisher
Create a presentation	Microsoft PowerPoint
Do calculations	Microsoft Excel
Play music	Apple iTunes

Programming languages

Stress again that computers are machines which follow instructions and the following ideas from page 2 of these notes:

- Computers are **deterministic**: they do what you tell them to do.
- Computers are **precise**: they do exactly what you tell them to do.

Computers can therefore be **understood**; they are just machines with logical working. If a computer doesn't work properly, it will be because it has been given incorrect instructions (assuming the hardware is functioning).

Suggested Activity

Show learners examples of the same algorithm implemented in two or three different languages (ideally one very high-level, such as Scratch) and one lower-level (such as Basic or C++) – and possibly even an assembly language program. Abstraction is a major focus of Pack 2 in this series (*Itching for More*).

Suggested activity

Discuss with learners the prevalence and importance of computers in society. Due to this importance, the computing industry is one of the biggest employers in the world, with often lucrative and rewarding careers.

Also discuss the fact that, despite this prevalence and importance, many people view a computer as an almost magical box that they understand less than the car they drive in. Why is it important to understand what a computer is?

Broader Inter-Disciplinary Learning

Global Citizenship

How are computers helping people in the developing world?

Responsible Citizens

Where do all the computers go?
Investigate the problem of electronic waste and its disposal.

Healthy Living

Learners could investigate the health risks…

- The risks of a sedentary lifestyle
- Driver and pedestrian distraction due to smartphones
- Disruption to sleep patterns caused by too much screen time

…and benefits

- improved medicines that could only have been created with the use of computers
- safety devices in the home, workplace and transport
- use of computers by police to catch criminals and keep us safe
- improved access to health information on line

associated with computers (of all types).

1: Scratching the Surface

Concepts introduced

- The Scratch environment, including:
  - Sprites & stage
  - Properties:
    - Scripts
    - Costumes/backdrops
    - Sounds
- Creating a program with animation & sound
- Sequencing instructions
- Fixed loops

Scratch commands introduced

- Motion
  - **move <n> steps**
  - **turn <n> degrees**
- Control
  - **when flag clicked**
  - **wait <n> secs**
  - **repeat <n>**
- Sounds
  - **play note <pitch> [for n beats]**
  - **play sound <name> [until done]**
- Looks
  - **change <effect name> effect by <amount>**
  - **next costume**

Computational Thinking themes

- Abstraction
  - what happens e.g. sound plays, sprite moves
  - position represented x & y coordinates
  - pitch represented by sound
- Pattern recognition
  - use of fixed loop to repeat code
- Decomposition
  - use of separate scripts to solve separate sub-problems
- Algorithm
  - sequence
  - event triggers action

Objectives

Learners should be able to:

- identify the major parts of the Scratch environment
- understand how sprites and blocks work and interact
- understand the concept of a computer program as a set of instructions
- work with simple animation and sound
- understand parallel v sequential execution of instructions

Materials

- Scratch projects: **Catwalk, FrereJacques**
- Screencasts: **Catwalk, FrereJacques**

Introduction

- Teachers could display the video introduction to Scratch to learners or let them watch on an individual basis.
- Show a selection of projects from the Scratch website (http://scratch.mit.edu) to show learners what is possible.

Task 1: Up on the Catwalk

- Teachers could then let learners watch the **Catwalk** screencast or demonstrate the Scratch environment themselves, stressing Sprites & Stage and Scripts, Costumes/Backdrops & Sounds.
- Once learners have done this, they should try to create a simple program to make a sprite dance using motion blocks and **next backdrop** blocks to animate

Task 2: Frère Jacques

- Once learners have done this, they should open screencast **FrereJacques**, which will take them through creating a simple music program.
- Stress the difference between **play sound** and **play sound until done**

IDL **Get learners to translate the lyrics of Frère Jacques.**

Task 3: My Tunes

- Learners should then create their own simple tune. Nursery rhymes are ideal for this, as they feature simple tunes and repetition.

Extension 1: Dance away

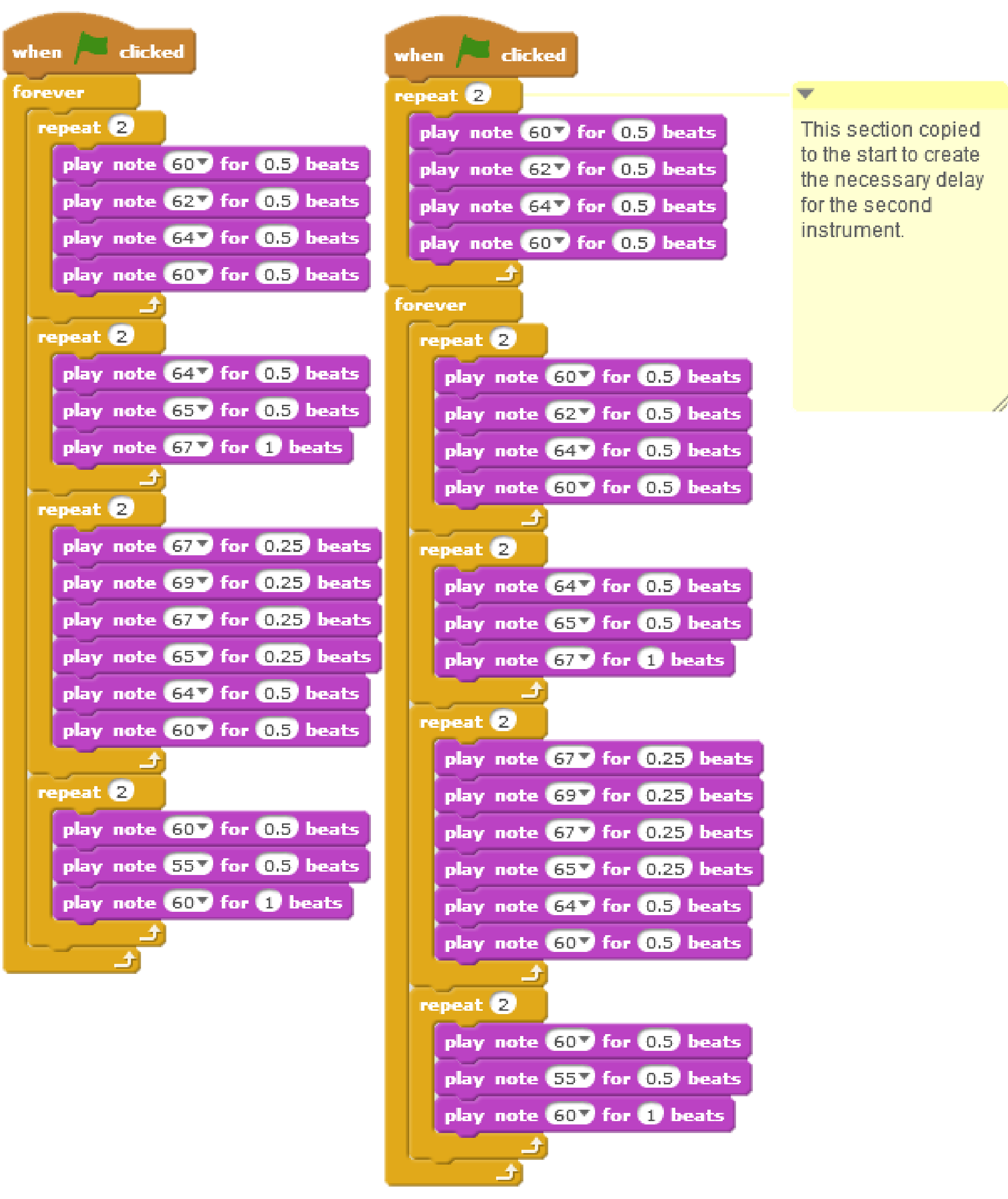
- Create a script which will make a sprite dance to the tune learners created in Task 3 (above).
- Teachers may wish to allow learners to compare making the sprite dance by
    - embedding motion blocks in the script that plays the tune
    - with a separate script that gets executed in parallel.
- A further exploration of parallelism can be explored using multiple scripts to play a chord, rather than separate notes at the green flag even e.g.

Suggested Activity

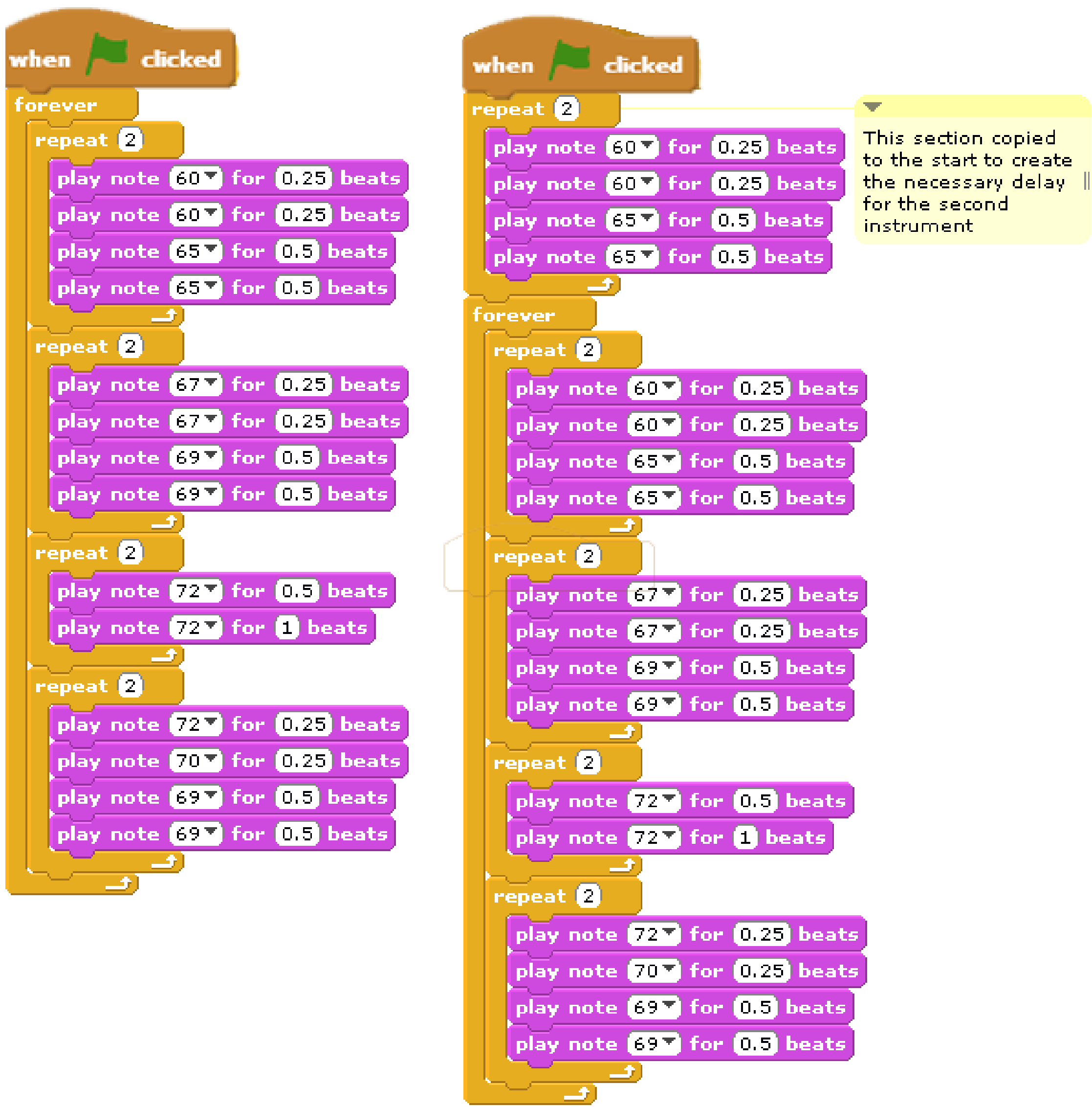
Frère Jacques is often sung as a *canon* or *round*. This is where two or more voices (or instrumental parts) sing or play the same music starting at different times. In a *round* each voice starts at the beginning again, going "round and round".

This could be used not only as an interdisciplinary learning opportunity, but also as an opportunity to explore parallelism further. Note that this can be multi-layered with further levels of paralellism to create a rich musical effect.

An example of parallel scripts for this is shown below. In this example, a **forever** loop is used to create a round.



Another common canon is *London's Burning*. An example of parallel scripts for this is shown below.

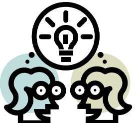


Extension 2

- Learners will enjoy adding colour and other effects to their sprites as they dance using the **change <effect name> effect by <amount>** block (**Looks** category).

Did you understand?

"Did you understand?" activities are one of the principal vehicles for stimulating Computational Thinking in learners. Tutors may wish to consider the following approaches:



- **group discussion – either in pairs or whole-class**
- **as classwork or homework exercises**
- **as vehicles for learners to peer-teach**

1.1 Look at the section of code opposite that controls a sprite. Write down what you think the user will see when the green flag is clicked.

The sprite appears not to move.

Why? **The computer carried out the instructions too quickly for us to see any movement.**

1.2 Now add a **wait 1 secs** block between the two move blocks. Describe what happens now.

We saw the sprite move.
Stress that computers work very quickly. In the first example, the sprite did move back and forth – it just happened so fast that we couldn't see it!

1.3 Look at the section of code below that controls a sprite.

Write down what you think the user will see when the green flag is clicked.

The sprite didn't move.

Why? **Both scripts get executed at the same time – in parallel – thereby "cancelling each other out".**

1.4 In the stack of blocks below, how many times does the sprite move 10 steps?

```
move 10 steps
repeat 3
  move 10 steps
  move 10 steps
move 10 steps
```

Eight times.

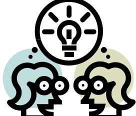
1.5 A programmer wants the cat to dance to some music. However, the cat doesn't start dancing until **after** the music has finished!

```
when [flag] clicked
play sound hip hop until done
repeat 5
  move 20 steps
  wait 0.5 secs
  move -20 steps
  wait 0.5 secs
```

The computer plays sound hip-hop until it is completed before going on to the next instruction. The programmer could have done one of two things:

- **used a play sound hip-hop block**
- **had the play sound until done block as part of another when flag clicked script.**

1.6 In the example below, a programmer has chosen a piece of music (sound "Xylo1") to play during a game. However, when the green flag is clicked, the computer just plays the first note of the music – over and over again!



What mistake has the programmer made?

The program is stuck in the forever loop, which keeps going around, never giving time for the play sound block to complete. The programmer should have chosen the play sound until done block.

1.7 In **Extension 1: Dance Away**, you made a sprite dance to a tune you created. There were **two** ways you could do this:

- create a **single script** that includes the sprite movement blocks amongst the play note blocks
- have **separate scripts** for the same sprite – one script plays the tune whilst the other makes the sprite dance.

Why do you think experienced programmers would use **separate scripts**?

It keeps different activities separate, allowing programmers to focus on one thing at a time.
Stress that this is an important idea in Computing Science (decomposition) – breaking down a big problem into smaller ones and solving each of these separately. We will return to this Computational Thinking theme often.

1.8 Make up a question of your own like those from 1.1–1.5 and pass it to your neighbour.

Encourage learners to anticipate likely mistakes or misunderstandings (possibly ones that they made themselves).

Lazy or smart?

Computer programmers are always looking for shortcuts to make their life easier.

A good example is how we used a **repeat** block in Frère Jacques to repeat the same line of music instead of having two identical sets of blocks. As well as looking neater, it also means that you won't make a mistake when creating a second set of blocks.

Do you think this makes programmers lazy or smart? (**Hint**: the answer is smart!)

You can make your life easier too by spotting shortcuts like this.

Suggested activity Discuss the panel above with learners. Stress the importance of Computational Thinking themes such as pattern recognition – and the satisfaction you get from creating a succinct and elegant solution to a problem.

Conclusion

- Revise the Scratch environment.
- Ask learners to summarise what they saw and learned.
- Emphasise that the **order of execution** is vital when programming.
  Stress that computer will do **only what it is instructed/programmed to do**.
  If a program doesn't work as expected, then we have made a mistake!

Further extension work

- Allow learners to experiment with different sounds/instruments in their tunes.
- Have multiple animates sprites dancing on the stage at once.
- Photograph learners in different dance poses (like the costumes of some of the human characters provided with Scratch). Import these as costumes for a sprite so learners can get themselves dancing.
    - Create a virtual party in Scratch, with entire groups of learners dancing at once.

2: Story Time

Concepts introduced

- Consolidation of the Scratch environment, including
    - Sprites & stage
    - Properties
        - Scripts
        - Costumes/backdrops
        - Sounds
- Sequencing instructions
- Sequential and parallel execution of code
- Bugs and debugging
- Event-driven programming (including programmer-defined events)

Scratch commands introduced

- Control
    - broadcast <message>
    - when I receive <message>
- Sounds
    - play sound <sound name> [until done]
- Looks
    - switch to costume <costume name>
    - say <string> for <n> seconds

Computational Thinking themes

- Abstraction
    - what happens e.g. sound plays, sprite moves
- Decomposition
    - Use of separate scripts to solve separate sub-problems
- Algorithm
    - sequence
    - event triggers action

Objectives

Learners should be able to:

- create stories and plays
- sequence instructions
- create their own events using the broadcast command
- incrementally develop a project

Materials

- Projects: **BadJoke,**
- Screencasts: **BadJoke**

Introduction

Display **BadJoke** screencast to learners, or allow them to watch it on their own.

Task 1: A bad joke

Demonstrate or allow learners to watch screencast **BadJoke**. This shows how to use Scratch to create a joke or play between two characters.

Girl	Boy
Say "Hey, I've got a joke!" for 3 secs	Wait 3 secs
Wait 3 secs	Say "Okay – let's hear it!" for 3 secs
Say "My dog's got no nose" for 3 secs	Wait 3 secs
Wait 3 secs	Switch to costume of boy shrugging Say "How does it smell?" for 3 secs
Say "Terrible" for 2 secs	Wait 2 secs
	Switch to costume of boy laughing Say "<Groan>" for 3 secs

Write down any problems you had and what you did to overcome them.

It is likely that the commonest cause of learners' problems will relate to:

- **ensuring that each sprite is talking at the correct time;**
- **maintaining a synchronised conversation if differing durations are used for each say command;**
- **remembering which script goes with what sprite.**

Task 2: A short play

Stress the following to learners:

- keep it simple with only two or three actors (sprites);
- write a script on lined paper, with each actor's lines side-by-side, as shown in the previous example.

Demonstrate the use of the **broadcast** block. It is left up to teachers' discretion whether to introduce the difference between **broadcast** and **broadcast and wait** blocks at this point. However, the use of **broadcast**, **broadcast and wait** and **procedures** is addressed in activity **4: Get the Picture?**

A screencast (*Haunted Scratch*, created for Scratch version 1.4) might serve to give some learner-friendly inspiration for this task. It can be found at https://www.youtube.com/watch?v=6OV_rJmPn4M

Extension 1: A walk-on part

This introduces the notion of initialising/resetting the program state every time it is run. Stress to learners that every small detail has to be programmed: the starting location, orientation, etc.

Bugs

A **bug** is an error which stops your code working as expected. There are **two** main types of bug which can occur in a program:

- **Syntax error**
  This happens when the rules of the language have been broken e.g. by mis-spelling a command. Syntax errors usually stop the code from running. Languages like Scratch provide code in ready-written blocks, so you won't make many syntax errors.
- **Logic error**
  This means your code runs, but doesn't do what you expect. Unfortunately, it's still possible to make logic errors in Scratch!

A third kind of error is also possible:

- **Execute/run-time error**
  This means your program crashes (stops running) when it is run (executed). This may be the result of performing an operation such as division by zero, for example.

Finding and fixing these errors in a program is known as **debugging**.

Suggested activity

Discuss the panel above with learners.
Introduce idea of programming **errors**. Stress that computers are **deterministic**, so that if code doesn't work as expected, it's because the programmer has made a mistake.

As part of this discussion, pose the question:

Q: How could we find **bugs** in our programs?
A: By **testing**. This is the point of testing – to find errors and correct them.

Now introduce learners to the idea of **design** and **algorithms**:

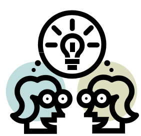
Q: What could we do to reduce the chance of bugs appearing in our code?
A: By planning it out in advance, just as we did for our own play.

Did you understand?

2.1 The program below shows the scripts for two sprites to tell a joke to each other. Why would this program not work?

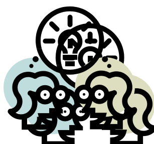
Girl	Boy
when [flag] clicked say Knock Knock for 3 secs say Doris for 3 secs say Doris locked. That's why I'm knocking! for 3 secs	when [flag] clicked say Who's there? for 3 secs say Doris who? for 3 secs say GROAN! for 3 secs

There are no pauses, meaning that the characters are talking at the same time. Appropriate wait commands need to be inserted.



2.2 Look at the example program below to tell a joke. Aside from being a terrible joke, what is wrong with this program?

Both scripts/sides of the joke are stored in the same sprite (Girl). The girl is telling a joke to herself.



2.3 The program below shows the scripts for two sprites to tell a joke to each other. Why would this program not work?

Girl	Boy
when [flag] clicked say Knock, knock! for 2 secs wait 3 secs say Doris for 2 secs wait 3 secs say Doris locked, that's why I'm knocking! for 3 secs	when [flag] clicked wait 3 secs say Who's there? for 3 secs wait 3 secs say Doris who? for 3 secs wait 3 secs say GROAN! for 3 secs

The timings are wrong. The wait and say commands need to have their timings adjusted so that the characters' lines are properly synchronised.

2.4 Now make up a buggy question of your own and pass it to your neighbour.

Learners' own answers here.

Event-driven programming

Some computer programs just run and continue on their own with no input from the user e.g. your program to play a tune.

However, many programs react to **events** (things that happen), such as:

- the click of a mouse or press of a key;
- the tilt of a game controller;
- a swipe of a smartphone screen;
- a body movement detected by a motion-sensing controller such as a Kinect

In Scratch, event blocks have a curved top (sometimes called a "hat"):

Reacts when the green flag is clicked. Often used to start a program.

Reacts when a key is pressed. Click the small black triangle to select the key you want to detect. Useful for controlling a sprite, or triggering an action.

Reacts when a sprite is clicked. Useful for controlling characters in a program

It is also possible to create your own events in Scratch using the **broadcast** command.

2.6 Look at the Scratch environment and write down some other **events** or **conditions** that Scratch programs can react to.
Hint: the **Control** and **Sensing** blocks are a good place to start.

For example:

- **Hitting the edge of the screen (if on edge, bounce)**
- **broadcast a user-definable event**
- **<property> of <object> e.g.**
  - **X position of sprite**
  - **direction of sprite**
  - **backdrop of stage**
- **touching sprite/edge**
- **touching color**

3: A Mazing Game

Concepts introduced

- Game creation
- Collision detection
- Loops
- Conditional statements

Scratch commands introduced

- Motion
  - if on edge, bounce

Only one new command is introduced in this lesson, as the intention is to consolidate those already learned within the new context of creating a game.

Computational Thinking themes

- Abstraction
  - repetition
  - position: x & y coordinates
- Algorithms
- Decomposition
  - breaking down game into main components
- Pattern Recognition
  - use of similar code for movement in different directions

Objectives

Learners should be able to:

- understand the use of an algorithm to develop a solution to a problem;
- become more familiar with translating an algorithm into code;
- use conditional statements.

Materials

- Screencasts: **Maze**

Introduction

Display **Maze** screencast to learners, or allow them to watch it on their own.

Alternatively, tutors could demonstrate the game and try to elicit an algorithm from learners.

Task 1: Setting the scene

Get learners to create new project and import Maze backdrop. Alternatively, they could create a maze of their own for the backdrop using Scratch's own graphics tools.

NB If doing this, ensure that learners' mazes are wide enough to accommodate sprites – and indeed navigable!

The Importance of Design

Before we make anything – a house, a dress or a computer program – we should start with a **design**. Because there are two important parts to most programs – the **interface** and the **code** – we design these separately.

- The easiest way to design the **interface** is by sketching it out on paper.
- The most common way of designing the **code** is to write out in English a list of steps it will have to perform. This is known as an **algorithm**.

Writing an algorithm is the key to successful programming. In fact, this is what programming is *really* about – solving problems – rather than entering commands on the computer.

All good programmers design algorithms before starting to code.

- Go over the panel above and introduce the concept of an **algorithm** – a list of steps to solve a problem (usually written in English).
- Point out to learners that they are learning to "think like a computer".

Task 2: Designing the solution

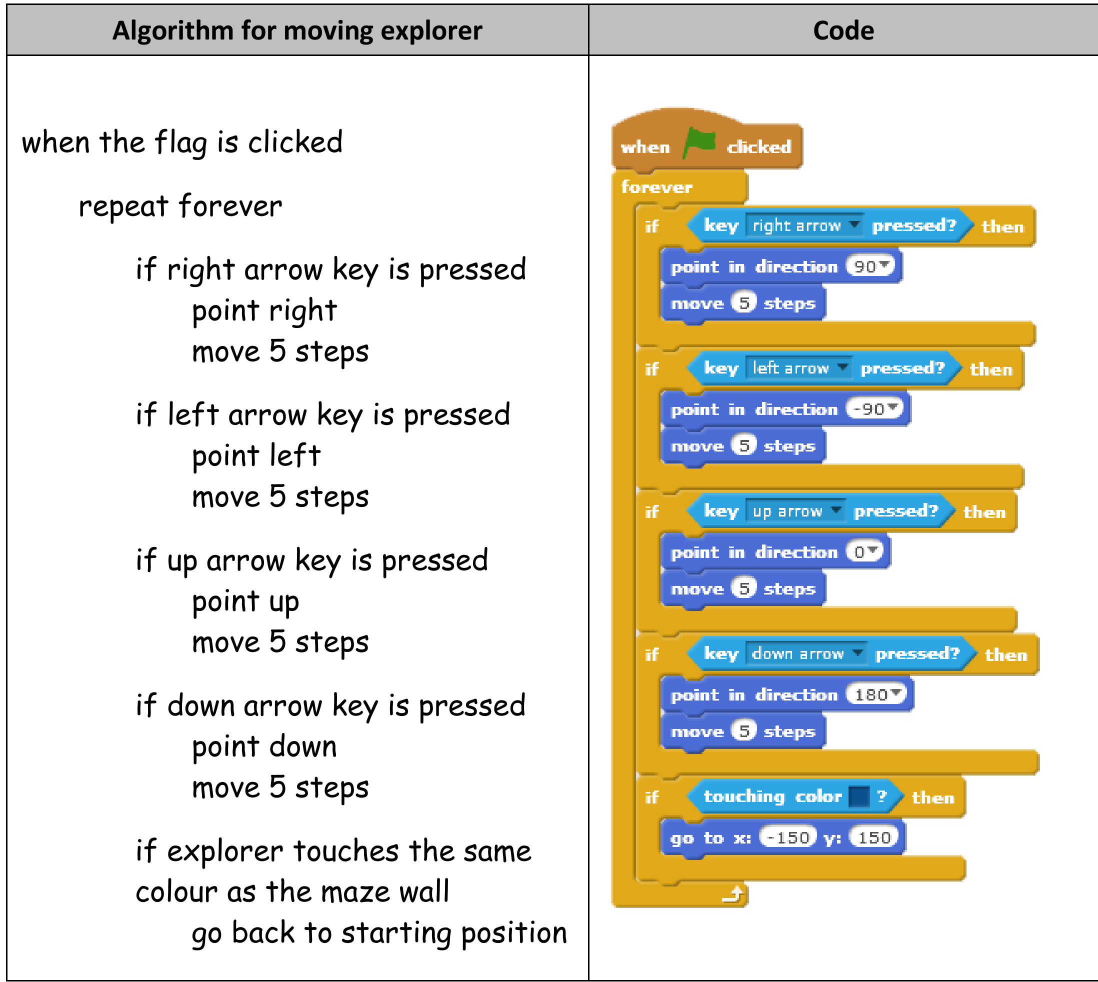
Stress to learners that there are **two** main things we need to code in our game:

1. moving the explorer;
2. reaching centre of the maze (and rescuing the explorer's friend).

By decomposing the problem, each part can be solved separately. This is a good opportunity to stress the importance of problem decomposition and why we do it i.e.

- it makes solving a large problem easier if we can break it down and solve each "sub problem" separately.
- in commercial software development, programs are often written by teams. By breaking down a problem like this, it enables different team members to work on different parts. The parts can them be combined to create the whole program.

Go over the table showing an **algorithm** for moving the explorer and Scratch **code** that does the same thing. Point out that the algorithm and code are two representations of the same thing – just at different levels (**abstraction**).

Algorithm for moving explorer	Code
when the flag is clicked repeat forever if right arrow key is pressed point right move 5 steps if left arrow key is pressed point left move 5 steps if up arrow key is pressed point up move 5 steps if down arrow key is pressed point down move 5 steps if explorer touches the same colour as the maze wall go back to starting position	

Stress how algorithms are indented to show structure.

Extension 1: Getting in tune

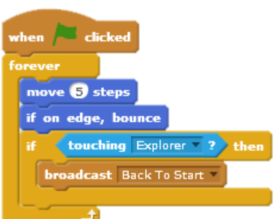
Where would be the best place to store this, since it applies to the whole game? **The stage.**

How will you get the music to keep playing? **Have it in a forever loop**

Should you use a play sound or play sound until done block to play the music? **play sound until done, so that the whole piece of music gets repeated, not just the first note.**

Extension 2: Add an enemy

The code for this is quite simple if the enemy collision appears in the Explorer script.



If learners put it in the enemy script, they would need to broadcast an event to the Explorer sprite to reset its position (with a corresponding when I receive Back To Start event in the Explorer script).

There is, of course, justification for this sort of modularity and this could be discussed with learners.

Extension 3: Two-player game

Whilst not in the Learner notes, tutors may wish to set learners the challenge of creating a two-player version of the game.

In such a game, different sets of keys on opposite ends of the keyboard could be used to control two different sprites racing towards the middle from opposite ends of a maze (ideally created by the learner with this purpose in mind).

Did you understand?

3.1 A programmer creates a maze game like the one you've just created. Unfortunately, her character doesn't move as expected.

```
when [flag] clicked
forever
  if key [right arrow] pressed? then
    point in direction (90)
    move (5) steps
  if key [left arrow] pressed? then
    point in direction (90)
    move (5) steps
  if key [up arrow] pressed? then
    point in direction (0)
    move (5) steps
  if key [down arrow] pressed? then
    point in direction (180)
    move (5) steps
```

What mistake has she made?

Left and right arrow both point right (90). Left arrow should point left (-90).

3.2 Look at the examples of code below.

```
when [flag] clicked
forever
  if <key (right arrow) pressed?> then
    point in direction (90)
    move (5) steps
  if <key (left arrow) pressed?> then
    point in direction (-90)
    move (5) steps
  if <key (up arrow) pressed?> then
    point in direction (0)
    move (5) steps
  if <key (down arrow) pressed?> then
    point in direction (180)
    move (5) steps
  if <touching color [ ] ?> then
    go to x: (-150) y: (150)
```

```
when [flag] clicked
forever
  if <key (right arrow) pressed?> then
    point in direction (90)
    move (5) steps
  if <key (left arrow) pressed?> then
    point in direction (-90)
    move (5) steps
  if <key (up arrow) pressed?> then
    point in direction (0)
    move (5) steps
  if <key (down arrow) pressed?> then
    point in direction (180)
    move (5) steps

when [flag] clicked
forever
  if <touching color [ ] ?> then
    go to x: (-150) y: (150)
```

Do they perform the same task? **Yes.**

Explain your answer

The forever / if touching colour stacks are both executed throughout the program, despite being in separate stacks in the right-hand example.

This could be a good opportunity to discuss the concepts of parallelism and modularity again:

Q: Why might some think the right-hand script is better than the left-hand?
A: It separates movement and collisions into two stacks. These are two separate problems which should arguably be solved separately. This will make the code easier to maintain if new features are added to either aspect.

3.3 The code below controls a sprite going round a maze. If the sprite touches the side of the maze (the colour blue), it returns to its starting position of -150, 150.

Unfortunately, the sprite sometimes touches the walls of the maze and returns to the start when the player doesn't expect.

```
when [flag] clicked
forever
  if <key [right arrow] pressed?> then
    move (5) steps
    point in direction (90)
  if <key [left arrow] pressed?> then
    move (5) steps
    point in direction (-90)
  if <key [up arrow] pressed?> then
    move (5) steps
    point in direction (0)
  if <key [down arrow] pressed?> then
    move (5) steps
    point in direction (180)
  if <touching color [ ] ?> then
    go to x: (-150) y: (150)
```

What mistake has the programmer made?

The sprite moves before pointing in the correct direction.

This will make it move 5 steps in the previously selected direction, causing it to touch colour blue if it happens to be close.

3.4 In this example, the sprite is meant to return to the centre of the maze when it touches the sides (coloured blue); however, it only does this sometimes.

What mistake has the programmer made?

```
when [flag] clicked
forever
  if key right arrow pressed? then
    point in direction 90
    move 5 steps
  if key left arrow pressed? then
    point in direction -90
    move 5 steps
  if key up arrow pressed? then
    point in direction 0
    move 5 steps
  if key down arrow pressed? then
    point in direction 180
    move 5 steps
    if touching color [ ] ? then
      go to x: -150 y: 150
```

The if touching colour block is <u>inside</u> the final if statement, meaning it will only ever be executed if the down arrow is pressed.

3.5 In this example, the sprite **never** returns to starting position, even if it touches the walls of the maze (coloured blue).

What mistake has the programmer made?

```
when [flag] clicked
forever
  if key right arrow pressed? then
    point in direction 90
    move 5 steps
  if key left arrow pressed? then
    point in direction -90
    move 5 steps
  if key up arrow pressed? then
    point in direction 0
    move 5 steps
  if key down arrow pressed? then
    point in direction 180
    move 5 steps

when [flag] clicked
if touching color [blue] ? then
  go to x: -150 y: 150
```

The if touching colour block is only executed at the very start of the game i.e. the point that the flag is clicked (a common mistake).

It therefore needs to be inside a forever loop.

3.6 Now make up a buggy question of your own and pass it to your neighbour.

Learners' own answers here.

4: Get the Picture?

Concepts introduced

- Problem decomposition and modularisation using procedures
- Timers/clock component

Scratch commands introduced

- Motion
  - go to <location or object>
  - point in direction <n>
- Control
  - broadcast <message> and wait
- Pen
  - clear
  - pen up, pen down
  - set pen color to <colour>
  - change pen color by <n>
  - set pen size to <n>
  - change pen size by <n>

Computational Thinking themes

- Abstraction
  - repetition, including nested repeat
  - position: x & y coordinates
- Algorithms
- Decomposition
  - building complex shapes from simpler ones
- Pattern Recognition
  - The Rule of Turn
  - use of iteration to build complex patterns
- Generalisation
  - The Rule of Turn

Objectives

Learners should be able to:

- understand the use of an algorithm to develop a solution to a problem;
- become more familiar with translating an algorithm into code;
- become more familiar with using variables;
- use procedures to create a more modular program.

Materials

- Screencast: **Graphics**

Task 1 : Shaping up

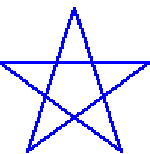
Display **Graphics** screencast to learners, or allow them to watch it on their own.

Learners should be encouraged to work out on paper the steps required to create the heptagon and triangle. It may be useful for one learner to direct a partner drawing the shapes on paper to encourage them to "**think like a computer**".

IDL Mathematics: Some learners will get the triangle wrong because they turn 60° (the internal angle) instead of 120° (the external angle).

Before going on to the next page, try to elicit from learners The Rule of Turn from what they have done already.

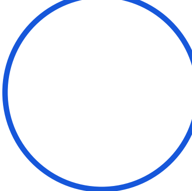
Task 2: You're a star!



The Rule of Turn will work here, but learners need to work out that they turn full circle ***twice*** in the creation of a star, so each turn is 144°.

Having one learner command another to draw (or even walk it through in the classroom) may help consolidate this.

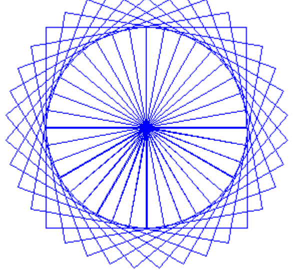
Task 3: Circle



Learners may spot that this is not a true circle, but a 36-sided polygon[3].

Task 4: Circular pattern

Learners should be encouraged to experiment with this task, changing repeating shapes and pen colour.



Some may have played with the geometric toy Spirograph™ when younger, which they may be able to relate to.

IDL There is clearly a great deal of mathematical content here. Teachers may wish to relate this to the maths that learners have already covered, or to liaise with their maths teacher in advance.

[3] A triacontakaihexagon.

Nesting

- In Task 4, we saw one **repeat** loop inside another – this is called a **nested** loop.
- In this case, the program starts the outer **repeat**, then enters the inner repeat, which carries on until it's finished. The outer repeat then carries on and so on.
- Add a wait 0.1 sec command in your code to see this happening more slowly.

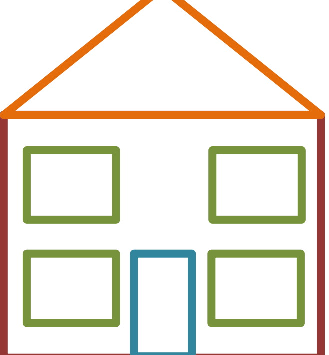
Tutors may wish to go over the panel above with learners, demonstrating Task 4 with the appropriate wait commands to slow down execution.

Extension 1: The main event

Task 4: Circular pattern is designed to be coded using procedures. However, it also provides an opportunity to show the use of the broadcast and broadcast and wait commands.

Note that using the broadcast command will not work properly, because the outer loop begins its next iteration before the inner loop has finished. This is a good way to demonstrate the difference between these two concepts – and it is the basis of "Did You Understand" Q3.4 later in this lesson.

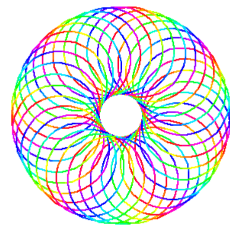
Extension 2: Our house



Learners would benefit from using squared paper to plan out their house. They should also use an equilateral triangle for the roof.

Learners should be encouraged to create the windows using their own procedure blocks.

Extension 3: Mmm… doughnuts



This will likely involve some trial and error, as the size of circle the learners choose as the building block might cause the pen to hit the edge of the screen, distorting the image.

Extension 4: The Olympic Rings[4]

This is a challenging exercise which will stretch the most able learners! Points they should consider when creating an algorithm include:

- Plan out on squared paper and note the distance between the centre points of each circle.
- Setting the pen size and colour at the start
- Use of **pen up** and **pen down**
- Changing colour of pen between each circle

Learners should be advised that they are not expected to make rings overlap/interlock like they do in the official logo.

[4] The Olympic rings symbol is reproduced by kind permission of the International Olympic Committee. The Olympic rings are the exclusive property of the International Olympic Committee (IOC). The Olympic rings are protected around the world in the name of the IOC by trademarks or national legislations and cannot be used without the IOC's prior written consent.

Did you understand?

4.1 The program from the screencast is shown below. Suggest any way(s) in which it could be made more efficient.

```
when [flag] clicked
go to x: -200 y: 0
clear
show
pen down
square
square
square
```

```
define square
repeat 4
    move 100 steps
    turn ↻ 90 degrees
pen up
move 150 steps
pen down
```

As hinted at in the screencast, the most obvious example of increasing code efficiency here would be to introduce a repeat 3 to make square repeat three times. Other changes could include:

- **Moving the pen down from the end of the square procedure to the beginning of it**
    - **Then removing the pen down command from the main program**
- **Hiving off the movement between squares into a separate procedure (as well as suggesting why this is a good idea).**

4.2 Look at the program below.

Write down the order in which the stacks are carried out after the green flag is clicked (number them in order 1, 2 and 3).

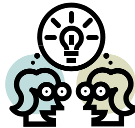
Number	Stack
3	define square repeat 4 move 100 steps turn ↻ 90 degrees
1	when clicked go to x: 0 y: 0 point in direction 90▾ clear set pen size to 2 pen down pattern
2	define pattern repeat 36 square turn ↻ 10 degrees

Now describe what the code will do.

Stack 1: **Initialisation: sets the starting point of the sprite, clears the stage, sets the pen size and puts the pen down. It then calls up the 'pattern' procedure.**

Stack 2: **The 'pattern' procedure: calls up the 'square" procedure and waits for completion; it then turns 10 degrees. This process is repeated 36 times.**

Stack 3: **The 'square' procedure: draws a square of side length 100 steps.**



4.3 Look at the code examples below.

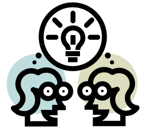
a) How many times will sprite move 10 steps? **12**

Why? **The outer loop causes the 4 passes of the inner loop to be completed 3 times: 3 x 4 = 12.**

```
repeat 3
  move 10 steps
  turn 120 degrees
  repeat 4
    move 10 steps
    turn 90 degrees
```

b) How many times will sprite move 10 steps? **15**

Why? **12 times in the inner loop + 3 times in the outer loop: 3 x (1 + 4).**



4.4 Discuss the following examples from real life. Write an “algorithm” for each one! _

a) Getting ready for school

Wake up, make breakfast, eat breakfast, get washed, get dressed, gather things, leave house.

b) Making breakfast

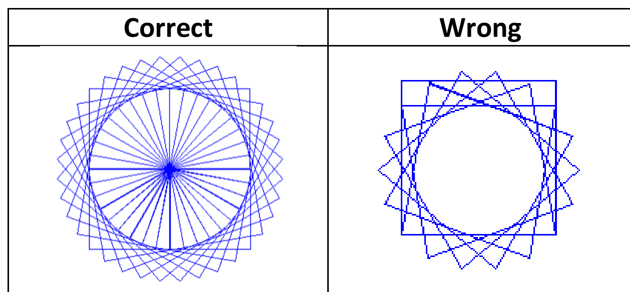
Get bowl, get milk, empty cereal into bowl;
Boil kettle, put bread in toaster, put teabag in pot, etc.

Note that this is one of the stages in part a), showing that successive decomposition can take place. Therefore one procedure can be broken down into further procedures, etc.

In the case of tea & toast, this could even be used to relate to parallel processing i.e. whilst waiting for the toaster to pop up or the kettle to boil, we get the other ingredients, for example. In this respect, we could have separate Scratch stacks for these stages that get executed at the same time.

4.5 In the **Storytime** activity you used the broadcast event to send a message between a sprite and the stage.

In this example, a programmer is using a broadcast event instead of a procedure to create a circular pattern of squares like the one labelled “Correct” below. Unfortunately, it always goes wrong, displaying the pattern labelled “Wrong”.



Look at the programmer’s code opposite. What mistake have they made?
Hint: it’s something to do with how fast the computer works.

This is a difficult question which is likely to challenge learners.

The programmer should have used broadcast and wait instead of just broadcast to draw the square.

The top stack is carrying out turn 10 degrees before when I receive square is yet complete.

Now enter the code above and run it to see the mistake for yourself. Once you have done this, create a procedure to draw the square and use it in the first script instead of the broadcast square block.

What does this tell you about the way that the broadcast command works compared to a procedure?

Procedures work more like a broadcast and wait event than a broadcast
i.e. when a procedure is called up the code execution is directed to the procedure. The effect of this is that code from where the procedure is called waits for the procedure to complete.

4.6 Now make up a buggy question of your own and pass it to your neighbour.

Learners' own answers here.

Did you understand (Extension 3 only)?

4.7 A programmer tries to draw a doughnut like the one in Extension 3. Unfortunately, it just draws lots of circles on top of each other.

```
when [green flag] clicked
clear
pen up
go to x: 0 y: -30
pen down
point in direction 90
repeat 36
    repeat 36
        move 10 steps
        turn ↻ 10 degrees
    pen up
    move 10 steps
    turn ↻ 10 degrees
    pen down
```

What mistake has she made?

Don't worry if you can't see it straight away – this is tricky! If necessary, enter the script into Scratch and run it to help you understand what's going on.

By moving 10 steps and turning 10 degrees at the end of every circle, she is effectively only going part of the way around the circle and then drawing another circle from that point.

She must change the program so it moves anti clockwise 10 degrees each time in the outer loop to ensure it is ready to draw the circle at a new starting point or moves by more than 10 steps between each circle

As suggested by the italicised text accompanying the question this is a very challenging question. Students may benefit from entering the code and running it to see it for themselves. Addition of wait blocks inside each loop to slow down the apparent speed of execution may also help.

5: Forest Archery Game

Concepts introduced

- Game creation
- Collision detection
- Loops
- Conditional statements
- Variables
- Random numbers

Computational Thinking themes

- Abstraction
  - repetition
  - position: x & y coordinates
- Algorithms
- Decomposition
  - Breaking down game into main components
- generalisation
  - use of a variable to keep score and time

Scratch commands introduced

- Motion
  - **glide <n> secs to x: <n> y: <n>**
  - **point in direction <n>**
- Control
  - **when sprite <sprite name> clicked**
  - **stop all**
- Sensing
  - **mouse x, mouse y**
  - **touching <sprite name>**
- Variables
  - **set <variable name> to <n>**
  - **change <variable name> by <n>**

Objectives

Learners should be able to:

- understand the use of an algorithm to develop a solution to a problem
- become more familiar with translating an algorithm into code
- use conditional statements

Materials

- Screencast: **ForestArchery**

Introduction

Display **ForestArchery** screencast to learners, or allow them to watch it on their own.

Alternatively, tutors could demonstrate the game to learners and try to elicit an algorithm from them.

Task 1: Designing the solution

Learners should be encouraged to create their code from the algorithms below, rather than watching the screencast again.

Point out to learners the use of **nesting** again – but this time with an if block inside another one, rather than a repeat.

In this case **if the sprite is touching the target sprite**

only gets carried out **if the mouse button is down.**

Task 2: Hit and miss

This introduces **if…else** i.e. if something is the case, do *this*…else/otherwise do *that.*

Elicit from learners other examples in real life of if...else
e.g. traffic lights – if at green go, else stop.

Task 3: Against the clock

This introduces a timer **variable** to learners' programs. The variable should appear on the screen (so should have a **meaningful name** and be **ticked** in the blocks area).

Discuss with learners different approaches to this e.g. repeat until timer=0 , and the problem of using a forever block (timer becomes negative).

Task 4: Bullseye!

The simplest way for learners to code this is to have lots of **if…then** statements (not nested). Whilst this is not particularly efficient – every **if** has to be evaluated even if one has already been executed successfully – it will be easier at this stage for learners to understand, compared to lots of nested if statements.

Scratch does not yet support a **case** (aka **switch**) statement.

Task 5: Stay positive!

In this task, learners simply have to add a condition – in the correct place – that tests for **if score>0** when subtracting a point for a miss.

Suggested Activity **Show learners Archery Champion beta 1.0 – a slick, professional-looking archery game created in Scratch v1.4 http://scratch.mit.edu/projects/Shanesta/9710.**

Variables

Go over the following panel with learners:

Variables

In this game, we introduced the idea of keeping a score using a variable block.

A **variable** is a space in a computer's memory where we can hold information used by our program – just like storing things in a box.

We should always give a variable a sensible **name** that tells us **what kind of information is stored in it** – just like putting a label on the box to tell us what's inside.

To create a variable in Scratch, we make a variable block.

Once a variable is created, the information stored inside it can be **set** or **changed** (that is, varied – hence the word "variable").

The use of variables is a key Computational Thinking concept (abstraction/ generalisation) and important for learners to understand.

A variable can be explained as a part of computer memory in which we store a value. We then label it – like a box – so that we can remember what's stored inside.

Suggested Activity

Pass out envelopes with names written on the front such as "age", "favourite colour" "pet", etc. Inside each one, have a piece of paper with a value written on it. Take out each value and put in a new one to demonstrate assignment of a new value to the variable.
Use this activity to discuss the importance of creating meaningful variable names so that we can recognise them easily in our program. Ask learners:

Q: Why store information in variables? Isn't it more complicated?

A: No! Variables make it **easier** for us to use and change information in a program. Explain how variables allow us to **generalise** our code.

For example:

- Consider a game that displays a score which changes throughout the game. *Would we have a separate program line for every possible score in a game? No, we'd have one program line, but refer to the variable. A variable is the only real way of storing this information.*

Extension 1: A Mazing cool feature

If learners no longer have their own Maze program, they can use the exemplar **Maze1**. Ask learners where this script is best located (the stage) and why (its scope is program-wide).

Extension 2: A Harder Maze

Learners will enjoy creating their own maze using Scratch's graphic tools.

One issue is that any maze should be easily navigable, so they may have to reduce the size of their explorer sprite.

When creating the bonus sprites, it is easiest if each bonus is set to detect touching the explorer (rather than have a complex script in the explorer), these could be set to show and hide after random periods of time.

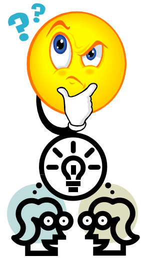
Extension 3: Do I get a prize?

Remind learners that to make a variable appear to the screen, simply tick the box next to the variable in the blocks area. Clicking on the variable displayed on the stage will show either its **value** or **value & name**.

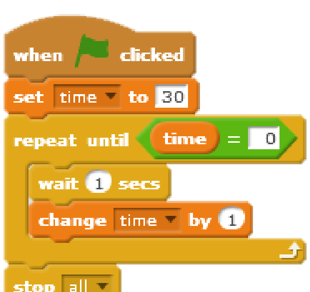
Extension 4: Now you see it…

This is a relatively simple extension using show / hide blocks. A further extension here could be to have bonus sprites re-appear in different locations. This would require a check that the sprite is actually within the maze area before showing it again, using if touching colour then .

Did you understand?



5.1 Look at the script below to make a timer variable count down from 30 to 0.



Will it work? **No.**

Explain your answer **The *time* variable is changing by + 1 each time around the loop (it is increasing in value). It needs to be -1.**

5.2 Now make up a buggy question of your own and pass it to your neighbour.

Learners' own answers here.

Scratch Project

Analyse

- Encourage learners to think about how their project could link with work they're doing elsewhere. In order to remain focussed, allow a maximum of 15 minutes to look at the Scratch gallery at https://scratch.mit.edu/explore/ .

Now discuss your ideas with your teacher.
Once you have agreed on your project, describe what it will do below.

- The main issue to look out for here is to ensure that the scope of the project is achievable for the learners concerned.

Design (Screen)

- One or two simple labelled sketches should suffice here. The purpose of this is to force learners to think through their project and how it will function.

Design (Code)

- Stress to learners that they should take time to get their algorithms right before coding. Remember the ancient proverb: *hours of coding can save minutes of design!*

Implement

- Remind learners about use of **meaningful identifiers** for sprites/costumes/sounds and variables.
- Stress to learners that they should be working from algorithms.

Test

- Learners should perform self-testing and get their classmates to test. They should describe the bugs that were found and how they were fixed (or not).

Document

- Learners should show they have considered how to write a short, snappy description of their project, including its main features.

Evaluate

- Encourage learners to discuss and reflect honestly on their work.
- Encourage learners to look at their code to see if they could make it more elegant (refer back to "Lazy or Smart?").

Maintain

- Suggest to learners the maxim:
  "A computer program is never finished – we just stop developing it any further."

Congratulations!

Encourage learners to continue their Scratch development at home.

Point out that they have only "scratched" the surface and that there are many more functions in Scratch!

Appendices

Appendix A: Learner Tracking Sheet

Name: ____ Class: ____________________

Stage	Progress * (D, C or S)	Date completed	Comment
Introduction			
Lessons			
1: Scratching the Surface			
2: Story Time			
3: A Mazing Game			
4: Get the Picture?			
5: Forest Archery Game			
Project			
Analysis			
Design			
Implementation			
Testing			
Documentation			
Evaluation			
Maintenance			

PROGRESS *

Developing Where the learner is working to acquire skills or knowledge.

Consolidating Where the learner is building competence and confidence in using the skills or knowledge.

Secure Where the learner is able to apply the skills or knowledge confidently in more complex or new situations.

Appendix B: Sample Code

Lesson 1: Catwalk

```
when [flag] clicked
forever
    move (10) steps
    play drum (36▼) for (0.2) beats
    next costume
    if on edge, bounce
```

Lesson 1: Frere Jacques

```
when [flag] clicked
forever
    repeat (2)
        play note (60▼) for (0.5) beats
        play note (62▼) for (0.5) beats
        play note (64▼) for (0.5) beats
        play note (60▼) for (0.5) beats
    repeat (2)
        play note (64▼) for (0.5) beats
        play note (65▼) for (0.5) beats
        play note (67▼) for (1) beats
    repeat (2)
        play note (67▼) for (0.25) beats
        play note (69▼) for (0.25) beats
        play note (67▼) for (0.25) beats
        play note (65▼) for (0.25) beats
        play note (64▼) for (0.5) beats
        play note (60▼) for (0.5) beats
    repeat (2)
        play note (60▼) for (0.5) beats
        play note (55▼) for (0.5) beats
        play note (60▼) for (1) beats
```

Lesson 2: Bad Joke

Girl sprite

```
when [flag] clicked
say [Hey, I've got a joke for you!] for (3) secs
wait (3) secs
say [My dog's got no nose!] for (3) secs
wait (3) secs
say [Terrible!] for (2) secs
```

Boy sprite

```
when [flag] clicked
wait (3) secs
say [Okay - let's hear it!] for (3) secs
wait (3) secs
switch costume to [boy4-shrugging v]
say [How does it smell?] for (3) secs
wait (2) secs
switch costume to [boy4-laughing v]
say [<Groan>!] for (3) secs
```

Lesson 3: A Mazing Game (including extensions)

Explorer (Cat) sprite

```
when [flag] clicked
forever
  if key right arrow pressed? then
    point in direction 90
    move 5 steps
  if key left arrow pressed? then
    point in direction -90
    move 5 steps
  if key up arrow pressed? then
    point in direction 0
    move 5 steps
  if key down arrow pressed? then
    point in direction 180
    move 5 steps
  if touching color [ ] ? then
    go to x: -150 y: 154
```

```
when [flag] clicked
go to x: -150 y: 154
show
set Timer to 2
forever
  wait 1 secs
  change Timer by -1
  if Timer = 0 then
    say You lose! for 2 secs
    stop all
```

Friend (2nd cat) sprite

```
when [flag] clicked
show
forever
  if touching Cat ? then
    say Thank you for saving me! for 3 secs
    hide
    stop all
```

Enemy (Dragon) sprite

```
when [flag] clicked
point in direction 90
forever
  move 5 steps
  if on edge, bounce
  if touching [ ] ? then
    play sound Gong until done
    stop all
```

Stage

```
when [flag] clicked
forever
  play sound Xylo1 until done
```

Lesson 4: Get the Picture?

Sample scripts used to draw shapes in this lesson:

```
define triangle
repeat 3
    move 100 steps
    turn ↻ 120 degrees
```

```
define square
repeat 4
    move 100 steps
    turn ↻ 90 degrees
```

```
define pentagon
repeat 5
    move 100 steps
    turn ↻ 144 degrees
```

```
define circle
repeat 36
    change pen color by 10
    move 10 steps
    turn ↻ 10 degrees
```

```
define pattern
repeat 36
    square
    pen up
    turn ↻ 10 degrees
    pen down
```

```
when [flag] clicked
go to x: 0 y: 0
point in direction 90
clear
set pen size to 2
pen down
pattern
```

Lesson 5: Forest Archery Game

Stage

```
when [flag] clicked
set Score to 0
set Time to 20
repeat until (Time = 0)
    wait 1 secs
    change Time by -1
stop all
```

Sight

```
when [flag] clicked
forever
    go to x: mouse x y: mouse y
    if mouse down? then
        if touching Target ? then
            change Score by 1
            play sound Pop
            say Hit! for 0.5 secs
        else
            change Score by -1
            play sound Rattle
            say Miss! for 0.5 secs
```

Target

```
when [flag] clicked
forever
    glide pick random 0.1 to 0.8 secs to x: pick random -240 to 240 y: pick random -180 to 180
```