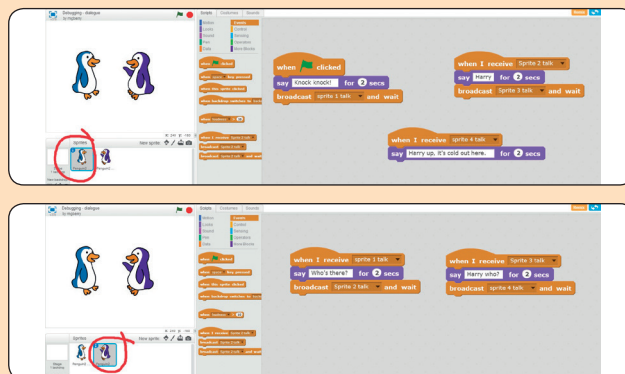




## 1 About this unit

- Software:** Scratch 2.0, Screencast-o-matic (if appropriate)
- Apps:** Snap! in the web browser (Scratch requires Adobe® Flash® Player, which is not available on iPad)
- Hardware:** Laptop/desktop computers, microphone (if appropriate)
- Outcome:** Debugged Scratch scripts and explanatory screencasts (if appropriate)



## UNIT SUMMARY

In this unit, the children work with six example Scratch projects. They explain how the scripts work, finding and correcting errors in them, and explore creative ways of improving them. The children learn to recognise some common types of programming error, and practise solving problems through logical thinking.

## CURRICULUM LINKS

### Computing PoS

- Debug programs that accomplish specific goals.
- Use sequence, selection, and repetition in programs; work with variables and various forms of input and output.
- Use logical reasoning to explain how some simple algorithms work and to detect and correct errors in algorithms and programs.

### Suggested subject links

- **English:** Programming emphasises a precise use of language and, in traditional, text-based programming languages, the importance of correct spelling and punctuation.
- **Maths:** This unit develops skills in logical reasoning and problem solving that can be applied right across the programme of study.
- **Science:** The work in this unit links to the requirements for working scientifically; in particular, making systematic and careful observations, and using results to draw simple conclusions and suggest improvements.

## TRANSLATING THE COMPUTING PoS

- Much of the work, and fun, in programming lies in spotting and correcting mistakes, known as 'bugs'. The process of finding and fixing bugs is called 'debugging'. In this unit, the children will *debug programs that accomplish specific goals*.

- The more complex a program is, the more likely bugs are to occur. Debugging and developing others' projects is a great way for pupils to *use logical reasoning to explain how simple algorithms work and to detect and correct errors in algorithms and programs*.
- The example scripts provided for this unit make use of *sequence, selection and repetition, variables and forms of input and output*.
- For more information about the different types of bug, see the unit poster.

## LEARNING EXPECTATIONS

This unit will enable the children to:

- develop a number of strategies for finding errors in programs
- build up resilience and strategies for problem solving
- increase their knowledge and understanding of Scratch
- recognise a number of common types of bug in software.

The assessment guidance on page 30 will help you to decide whether the children have met these expectations.

## VARIATIONS TO TRY

- Pupils could develop their debugging skills by fixing the code they and their classmates develop. You may be able to cover the ideas in this unit by linking it with *Unit 3.1 – We are programmers*.
- You don't have to use the example scripts provided on the CD-ROM – you could use your own scripts, or your pupils' scripts, instead.
- Pupils could debug and/or develop programs downloaded from the Scratch website, which are all covered by a Creative Commons BY-SA licence, i.e. they can be reused as long as the original author is credited and the resulting projects are shared on the same basis.

## 2 Getting ready

### THINGS TO DO

- Read the *Core steps* sections of *Running the task*.
- Decide which software/tools are most accessible/appropriate for use with your class. Scratch is recommended and the example scripts provided are all built using Scratch 2.0.
- Download your chosen software/tools (see *Useful links*), or ensure pupils have access to the Scratch website. They do not need to register for accounts.
- Watch the *Software in 60 seconds* walkthroughs. The walkthroughs are not directly related to bug fixing but provide a useful reference point.
- Work through the example scripts provided on the CD-ROM (or online) and have a go at debugging them.
- Think about the individuals and groups you have in your class. Could you use any of the *Extensions* on pages 24–29 to extend your more able children?

Could you use any of the suggestions in *Inclusion* (see below) to support children with specific needs, e.g. SEN or EAL? Have you considered how a Teaching Assistant will support you and the children, if one is available?

- Ensure you have sufficient computers/laptops/tablets and other equipment booked in advance.
- Decide whether you need evidence of pupils' debugging – are corrected scripts sufficient, or do you want pupils to record screencasts? It is useful for pupils to be able to record an explanation of how they improved the scripts, but the time taken will detract from their programming.

### THINGS YOU NEED

- Computers/laptops/tablets loaded with the software you have chosen
- Internet access
- Screen recorder software and microphones, if appropriate.



### CD-ROM RESOURCES

- *Software in 60 seconds* – Scratch (1–5)
- *Software in 60 seconds* – Introduction to Snap!
- Six Scratch scripts (with bugs) for children to work on
- Unit poster – Different types of bugs
- Pupil self-assessment information



### E-SAFETY

- Pupils don't need accounts to download Scratch 1.4 or to use Scratch 2.0 or Snap! online.
- If pupils do register for accounts, they need to give a parent's or carer's email address, so you should check with parents or carers that they're happy for their children to do this.
- Once registered, pupils can share their corrected and refined programs with the global Scratch community in a safe online space. Alternatively, pupils can upload their completed projects to the school's learning platform or blog.
- If pupils upload screencasts of their solutions, make sure you take the usual precautions to protect their identity.
- If pupils use the web for research (see *Extensions*), ensure all usual internet safety protocols are in place.



### INCLUSION

- Scratch has several languages built in (use the globe icon at the top of the screen).
- This unit uses maths skills. You may want to give extra support to pupils who struggle with maths.
- Some pupils may benefit from working with a partner, particularly for the last couple of steps.



### USEFUL LINKS

#### Software and tools

- Scratch is free open source software. Download Scratch 1.4 from [http://scratch.mit.edu/scratch\\_1.4](http://scratch.mit.edu/scratch_1.4) or use Scratch 2.0 online at <http://scratch.mit.edu/projects/editor>.
- Snap! is free open source software. Use online at <http://snap.berkeley.edu/snapsource/snap.html>.
- Screencast-o-matic is a free online screen recorder using Java: [www.screencast-o-matic.com/screen\\_recorder](http://www.screencast-o-matic.com/screen_recorder).

#### Online tutorials

- Introduction to Scratch 2.0: <http://scratch.mit.edu/help/videos>.
- Suggested solutions for each Scratch bug: <http://youtu.be/grMMY2LSKFI>.

#### Information and ideas

- Miles Berry's Scratch project directory: <http://scratch.mit.edu/studios/306100>.
- There are many further debugging challenges on the Scratch site. See [http://scratch.mit.edu/search/google\\_results?q=debugging](http://scratch.mit.edu/search/google_results?q=debugging) and <http://scratch.mit.edu/studios/219583>.

### 3 Running the task – We are bug fixers

**Software:** Scratch 2.0, Screencast-o-matic (if appropriate) **Apps:** Snap! in the web browser (Scratch requires Adobe® Flash® Player, which is not available on iPad)

**Hardware:** Laptop/desktop computers, microphone (if appropriate) **Outcome:** Debugged Scratch scripts and explanatory audio files or screencasts (if appropriate)

## Core steps

### Step 1: Spotting and correcting off-by-one bugs

#### RESOURCES

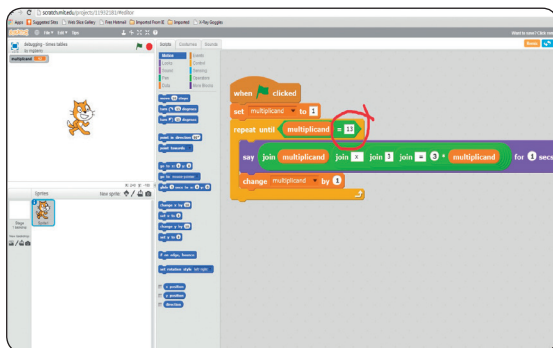


- Scratch multiplication script



- Scratch multiplication script: <http://scratch.mit.edu/projects/11932181>

#### POSSIBLE OUTCOME FOR THIS STEP:



- Share the *Learning expectations* (see page 22) and explain the success criteria.
- Revise the term ‘algorithm’: an unambiguous procedure or precise step-by-step guide to solve a problem or achieve a particular objective.
- Ask the pupils to recite their three-times table. What did they say? Could they phrase the instruction more clearly? Ask them to write an algorithm for reciting the three-times table, e.g. ‘Starting from  $1 \times 3$ , multiply each counting number by three, up to and including  $12 \times 3$ .’
- Show the children the Scratch multiplication script on the CD-ROM. It is meant to read the twelve-times table up to  $12 \times 3$ . What’s wrong with it? (It stops at  $11 \times 3$ , because it repeats *until*  $12 \times 3$ , but not *including*  $12 \times 3$ .) Explain that ‘off-by-one’ errors like this are common mistakes in programming.
- Ask the pupils to compare the Scratch script to their algorithm. Can they find the *script* block that needs changing so that the cat says the three-times table all the way through?
- Ask the pupils to fix the program by editing the script and testing the changes they make.
- If time allows, you could ask the pupils to record a screencast explaining how they debugged the program.

## Extensions

### SCHOOL

- Once pupils have got the script working properly, they could look at ways to improve the program, such as allowing users to choose the times table, or improving the graphics.
- Some pupils could have a go at rewriting the script without using Scratch’s  $() \times ()$  block, e.g. using repeated addition for working out the product – perhaps using a running total variable and adding three each time round the loop.

### HOME

- Syntax errors are bugs in which the spelling, punctuation or ‘grammar’ of a program isn’t quite right. It’s hard to make this sort of mistake with Scratch, but the project at <http://scratch.mit.edu/projects/11932059> has similar problems. It’s meant to draw a set of ten squares, one inside the other: can the pupils fix it?

## Step 2: Spotting and correcting performance bugs

### RESOURCES

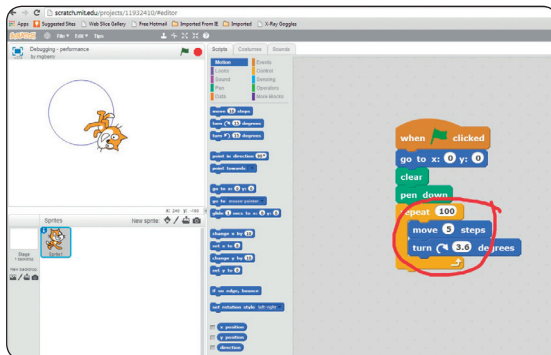


- Scratch circle script



- Scratch circle script: <http://scratch.mit.edu/projects/11932410>

### POSSIBLE OUTCOME FOR THIS STEP:



- Ask the pupils to draw a circle on a piece of paper. Discuss the differences between how different children drew these. Did some draw much more slowly than others? Did any use compasses or objects to draw round? Were the faster pupils less accurate than the slower ones?
- Remind the pupils what an algorithm is (unambiguous, step-by-step instructions). Invite them to write an algorithm for drawing a circle, in terms of the steps or rules they followed. Compare the algorithms that they have written.
- Head out into the playground or gym, if you can, and practise the following possible approach to drawing a circle.

Repeat until you get back to the start:

- Walk forward one step.
- Turn right a bit.

- Ask the pupils to look at the Scratch project at <http://scratch.mit.edu/projects/11932410> and try to explain how it works.
- Run the script. What do the pupils think? The program is meant to draw a circle, or at least a close approximation to one. Did it work? Did it work well? Try to elicit a response of: 'The program takes too long to draw the circle.'
- Ask the pupils to look for ways to improve the performance of the program while keeping the size of the 'circle' the same. Ask the pupils to experiment with changing the numbers. (It is not necessary to explain what the angle values mean as pupils can discover for themselves how changing this affects the shape drawn.) At what point does it become obvious that their shape is no longer a circle?
- The pupils could record a screencast or audio file explaining how the program works and what they did to improve its performance.

### SCHOOL

- Ask the pupils to make further changes to the program to create more interesting shapes or patterns, or to explore other blocks in Scratch's *Blocks* palette.
- Set the pupils the challenge of changing this script so that it draws a circle centred on the cat's initial position, with the cat returning back to the start at the end.

### HOME

- The pupils could research the origins of the term 'bug' for mistakes in software.
- The pupils could explain the term 'algorithm' to their parents or carers, and show them their algorithm for drawing a circle on paper.



# Core steps

## Step 3: Spotting and correcting multi-thread bugs

### RESOURCES

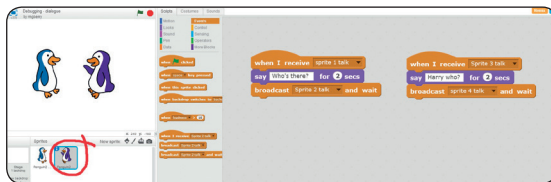


- Scratch penguin script



- Scratch penguin script: <http://scratch.mit.edu/projects/11932160>

### POSSIBLE OUTCOME FOR THIS STEP:



- Ask the pupils to think about occasions when everybody in the class is doing something at the same time, such as collecting things from their trays or tidying up. What sort of problems do they encounter? How can these be avoided? Could the pupils write an algorithm for tidying up the classroom at the end of the day? Would this algorithm work if all pupils followed it at the same time?
- Explain that in modern computers, several things can be happening at once, and that sometimes this can cause difficulties – when one process races ahead without another catching up, or when several processes are all waiting for the same shared resource.
- Show the pupils the Scratch project listed in *Resources*, which is meant to be two penguins telling a joke. Ask the pupils to identify the problem with this project and try to fix it. It may be helpful to introduce pupils to (or remind them of) the `say () for () secs` and `wait () secs` blocks, and/or the `broadcast ()` and `when I receive ()` blocks. (See Step 5 of Unit 3.1 – We are programmers.)
- The pupils could record a screencast explaining how the program works and what they did to fix it.
- The pupils could adapt their corrected script for other jokes or characters.

# Extensions

## SCHOOL

- The penguin sprites could be animated while talking – each has a second costume to allow this.

## HOME

- Pupils could try more debugging exercises in Scratch that explore working with multiple sprites, e.g. <http://scratch.mit.edu/projects/10437040/>, <http://scratch.mit.edu/projects/10745531/> and <http://scratch.mit.edu/projects/10745563/>.

## Step 4: Spotting and correcting conceptual bugs

### RESOURCES

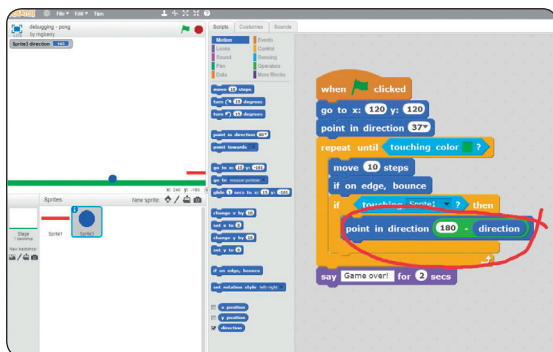


- Scratch 'Pong' script



- Scratch 'Pong' script: <http://scratch.mit.edu/projects/11932263>

### POSSIBLE OUTCOME FOR THIS STEP:



- Ask the pupils to describe what happens if they roll a ball towards a wall, either straight on or at an angle. Can they draw a picture to show how the ball would bounce? Do the experiment, changing the angle of impact. Are the pupils surprised? Can they come up with a rule to describe what happens?
- Explain that often, programs have bugs because the programmer hasn't fully understood the idea of what's supposed to happen in the program – the bug lies in the concept for the program rather than the code. These sorts of logic bugs can be tricky to find and fix.
- Let the pupils play the simple 'pong'-style game listed in *Resources*. Do the pupils notice anything odd about the game? They should spot that the ball doesn't bounce back correctly when it hits the bat (it always bounces off in the direction it came).
- Ask the pupils to study the script to work out how the game works, and then identify which block has the bug (if necessary, remind them they need to have the ball sprite selected to see the appropriate script). Ask them to correct the bug, drawing on their knowledge of how a ball bounces from discussions at the start of the session. The correct solution is to use *point in direction* ( $180 - \text{direction}$ ). Use your judgement about how much scaffolding to provide.
- You might want pupils to record a screencast explaining how the program works and what they did to fix it.
- You could ask the pupils to make further changes to the game, perhaps allowing a number of lives, and keeping score.

### SCHOOL

- The pupils could convert this into a two-player game, using two bats, each controlled by different keys, e.g. cursor arrows and the W and S keys, with 'out of bounds' on either side of the board, and perhaps a score for how many faults each player makes.

### HOME

- The pupils could draw up a list of bugs or ideas for improvements they've spotted in programs they've used, including on the web or on smartphones.

# Core steps

## Step 5: Spotting and correcting arithmetical bugs

### RESOURCES

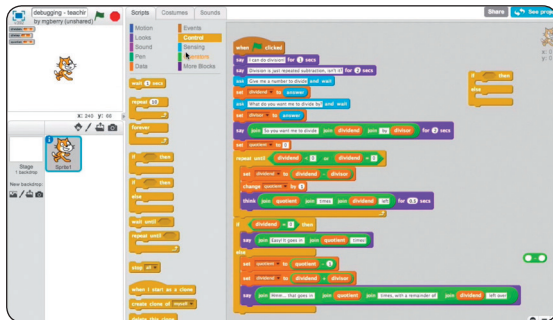


- Scratch division script



- Scratch division script: <http://scratch.mit.edu/projects/13550313>

### POSSIBLE OUTCOME FOR THIS STEP:



- Ask the pupils to practise their division in pairs. One pupil could give a number, the other the number to divide by, and both could work out the answer. Ask the pupils to consider how they worked out the answers to the division questions. Can they write their method as an algorithm? For example, 'Start with the number to be divided (the dividend). Count how many times you can subtract the divisor until you get to zero.' Another example might be 'Look up the answer on the times table and the number you multiply by tells you the quotient.'
- Are there any divisions their algorithm won't work for? A test plan (a list of divisions to try) would be useful to see if there were any special cases when their algorithm didn't work. Check if the pupils' algorithms work for dividing where there is a remainder, e.g.  $7 \div 3$ . What about when the children try to divide by zero?
- Show the pupils the project listed in *Resources* and ask them to explain to each other how this program works. Ask them to test the program. Does Scratch always give the right answer when the divisor 'goes in' exactly? What about when it doesn't?
- Ask the pupils to edit this project so that Scratch will work out remainders.
- Ask the pupils to edit the script to use Scratch's built-in *division* block, and to test the program again. Were they surprised by what happened?
- The pupils could record a screencast explaining what the script does and how they have used logical reasoning to improve it.

# Extensions

## SCHOOL

- The pupils could make further changes to the algorithm so that when the number doesn't divide exactly, the answer is displayed as a fraction or a decimal rather than as a whole number with a remainder.
- The pupils could improve the project in other ways too, perhaps making this the basis of a calculator project.

## HOME

- A similar bug is present in the division game project at <http://scratch.mit.edu/projects/11932022>. Does this game work the way it should? Ask the pupils to debug this script.

## Step 6: Spotting and correcting resource bugs

### RESOURCES

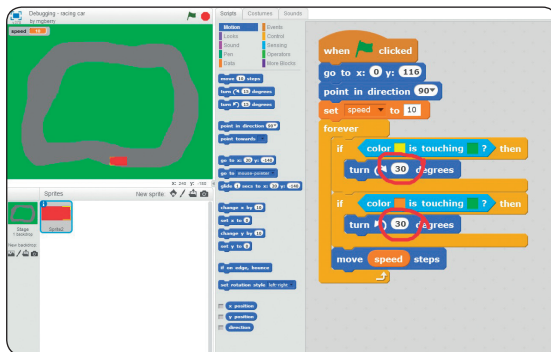


- Scratch racing car script
- Pupil self-assessment information



- Scratch racing car script: <http://scratch.mit.edu/projects/11932304>
- Video clip 1: Creating a driving simulator: [www.bbc.co.uk/programmes/p016j4g5](http://www.bbc.co.uk/programmes/p016j4g5)
- Video clip 2: Simulating Formula 1 racing: [www.bbc.co.uk/programmes/p016612j](http://www.bbc.co.uk/programmes/p016612j)
- Video clip 3: Google's self-driving cars: [www.youtube.com/watch?v=cdgQpa1pUUE](http://www.youtube.com/watch?v=cdgQpa1pUUE)

### POSSIBLE OUTCOME FOR THIS STEP:



- Show the pupils the video clip about creating a driving simulator. The clip ends with the pupils discovering what appears to be a bug in their scripts. The pupils might also enjoy watching the second clip, about simulating Formula 1.
- Show the pupils the Scratch project referenced in *Resources*, which uses the same code as the scripts in the video.
- Ask the pupils to study the script and explain how it works. What happens when the speed is increased? At what speed does the car go off the track? Does it matter where the car starts or which direction it's facing? Does the track shape make a difference?
- Can the pupils explain why this script seems to work for low speeds but breaks for high speeds? Can they think of a way to fix this? What could they change? Be aware that this is quite hard, owing to limitations in Scratch, but the pupils will still benefit from exploring this, and thinking of ways to work around these limitations.
- The pupils could record a screencast explaining how the program works and what they've done to improve it.
- Show the pupils the clip about Google's self-driving cars. The pupils should discuss what an algorithm for driving would be like. This is an example of where it's better to think of algorithms as sets of rules, e.g. if the road ahead is clear, then accelerate up to a safe speed; if the speed is greater than the speed limit, then stop accelerating or brake until the speed is below the speed limit, etc. It might be easier to do this in terms of the rules that drivers should follow. What events do they encounter? How should they react to them? The pupils should think about how important it would be to debug the program driving the car, and what the consequences of bugs in this software might be.
- Finally, pupils should evaluate the success of their work.

### SCHOOL

- Some children could convert the program into a driving game, perhaps by adding another car under the player's control.

### HOME

- The pupils could write a blog post about the programs they've studied, the different bugs they've encountered, and how they've fixed them. The pupils could also observe their parents or carers driving, and try to work out some of the driving 'algorithms' they use.



## Assessment guidance

Use this page to assess the children's computing knowledge and skills. You may wish to use these statements in conjunction with the badges provided on the CD-ROM or community site and/or with your own school policy for assessing work.

### ALL CHILDREN SHOULD BE ABLE TO:

- Correct 'off-by-one' errors in loops
- Improve the performance of the circle-drawing program
- Get the dialogue in the joke program to work in sequence
- Experiment with the speed variable and other factors in the racing car simulator

### BADGE



### COMPUTING PoS REFERENCE

- Debug programs that accomplish specific goals
- Debug programs that accomplish specific goals
- Debug programs that accomplish specific goals
- Work with variables: use logical reasoning to detect errors in programs

### MOST CHILDREN WILL BE ABLE TO:

- Describe how the times-table program works
- Describe how the circle-drawing program works
- Describe how the two joke scripts work together
- Correct the 'Pong'-style game so the bounce is more realistic
- Describe how the racing car simulator works



- Work with variables
- Use logical reasoning to explain how some simple algorithms work
- Use logical reasoning to explain how some simple algorithms work
- Debug programs that accomplish specific goals, including simulating physical systems
- Work with variables

### SOME CHILDREN WILL BE ABLE TO:

- Explain how they debugged the times-table program using logical reasoning
- Explain the connection between the number of steps, step size and turn in the circle-drawing program
- Explain how they corrected the joke program
- Describe how the 'Pong'-style program works
- Suggest explanations for the bug in the racing car simulator



- Work with variables
- Use logical reasoning to explain how some simple algorithms work
- Use logical reasoning to correct errors in programs
- Use logical reasoning to explain how some simple algorithms work
- Debug programs that accomplish specific goals, including simulating physical systems: work with variables

### PROGRESSION

The following units will allow your children to develop their knowledge and skills further.

- Unit 4.1 – We are software developers
- Unit 5.1 – We are games developers

## 5

## Classroom ideas

Practical suggestions to bring this unit alive!



### DISPLAYS AND ACTIVITIES

- Screenshots of the scripts before and after debugging could form a display.
- Encourage the pupils to step through the programs they study to get a better feel for what's going on – they can take turns to role-play the sprite, following the instructions as another pupil reads them.
- The Computer Science Unplugged resources at <http://csunplugged.org/routing-and-deadlock> and <http://csunplugged.org/sorting-networks> provide great, classroom-based activities linked to examples of more complex types of algorithms.



### WEBLINKS

- The Wikipedia article on software bugs provides a good introduction: [en.wikipedia.org/wiki/Software\\_bug](http://en.wikipedia.org/wiki/Software_bug).
- Michal Armoni and Moti Ben-Ari have an excellent, free book exploring computer science concepts through Scratch. See [stwwww.weizmann.ac.il/g-cs/scratch/scratch\\_en.html](http://stwwww.weizmann.ac.il/g-cs/scratch/scratch_en.html).
- See [www.bbc.co.uk/news/technology-18301670](http://www.bbc.co.uk/news/technology-18301670) for some primary pupils' experiences using text-based programming on the Raspberry Pi.
- Tips on debugging from StackExchange can be found at [programmers.stackexchange.com/questions/10735/how-to-most-effectively-debug-code](http://programmers.stackexchange.com/questions/10735/how-to-most-effectively-debug-code).



### VISITS

- A software developer might be willing to talk to the class about their work via video conference.



### BOOKS

- Agans, D. *Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*. (AMACOM, 2006)
- Badger, M. *Scratch 1.4 Beginners Guide*. (Packt Publishing, 2009)
- Butcher, P. *Debug It!: Find, Repair, and Prevent Bugs in Your Code*. (Pragmatic Bookshelf, 2009)
- Ford, J. *Scratch Programming for Teens*. (Delmar, 2009)
- Hahn, D. *The Alchemy of Animation*. (Disney Editions, 2011)
- Jonassen, D. *Learning to Solve Problems: A Handbook for Designing Problem-Solving Learning Environments*. (Routledge, 2010)
- LEAD Project, The. *Super Scratch Programming Adventure!* (No Starch Press, 2012)
- Metzger, R. *Debugging by Thinking: A Multidisciplinary Approach*. (Digital Press, 2003)

## 6

## Taking it further

When you've finished, you might want to extend the project in the following ways.

- Debugging has some connection with proofreading in English and 'checking your working' in maths, as well as overcoming problems in design and technology, and experimental approaches in science. Emphasise the application of logical reasoning across the curriculum.
- The skills the pupils learn in this unit should help them deal with problems they encounter when writing their own programs in other units.
- Pupils could help other Scratch users debug and improve their programs. If they have accounts on the Scratch website, they could create and publish improved versions of other people's programs.
- At the time of writing, the Simple English Wikipedia entry on debugging, at [http://simple.wikipedia.org/wiki/Software\\_bug](http://simple.wikipedia.org/wiki/Software_bug), is very limited. The pupils could expand on the material there.
- Open source projects (such as Moodle and Wordpress) welcome detailed bug reports from their users.