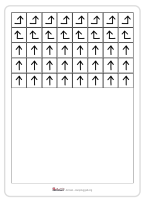
Sending a rocket to Mars

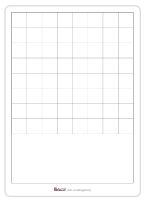
- **Duration:** 30 minutes
- **Ages 8 to 10:** Lesson 1

Printables



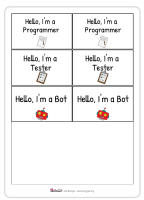
Arrows

Cut these squares out for students.



Grid

8 x 8 grid for students to use if no large grid available.



Job Badges

Cut these out and use them as badges for the three students doing the three jobs respectively. You might want to put them in a name holder on a lanyard.

Left and Right Cards

Cut these sheets out for students who are still learning their left and right.

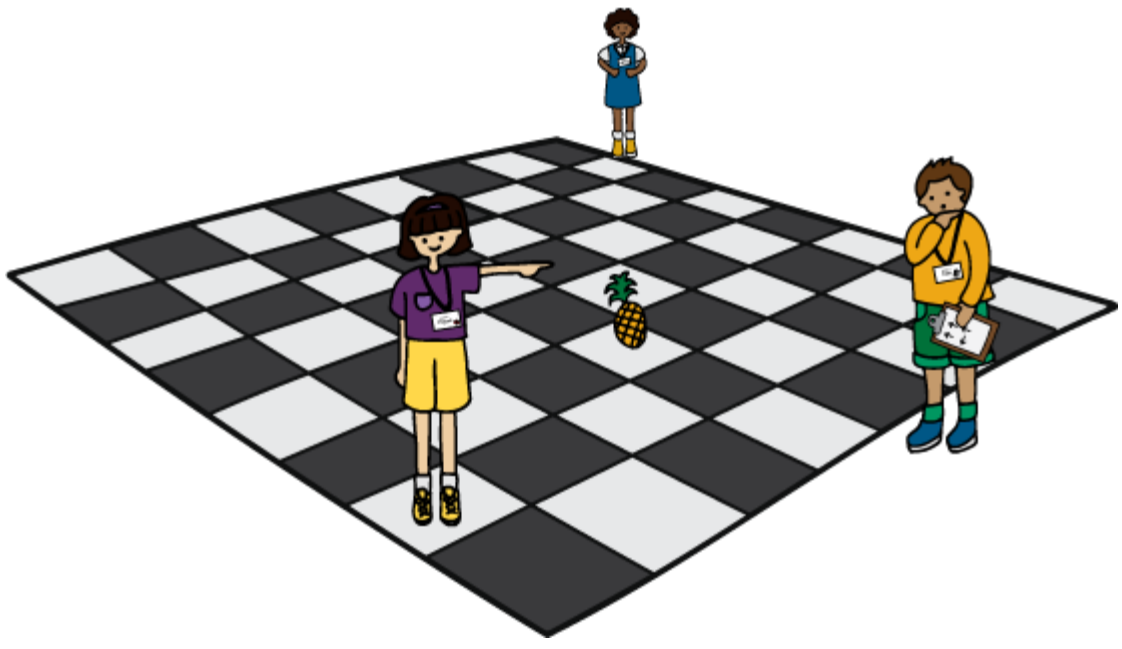
Classroom resources

- Blocks
- Clipboards
- Handheld whiteboards
- Paper
- Pens
- Whiteboard pens

Learning outcomes

Students will be able to:

- Actively listen to the feedback and debug the program.
  Computer Science: Programming | Literacy: Listening
- Communicate when the code isn't correct so that the person knows what needs debugging.
  Computer Science: Programming | Literacy: Speaking
- Explain why it's important to give very exact instructions.
  Computer Science: Programming | Mathematics: Geometry
- Give accurate instructions at a pace that the person can follow.
  Literacy: Speaking
- Give a set of precise instructions that programs an object to move efficiently from one point to another.
  Computer Science: Programming
- Identify where a bug has occurred and be able to correct the code to allow the object to move to the desired destination.
  Computer Science: Programming



This example is a starter lesson with an object in the middle. The girl on the white square is 'The Bot', the girl at the back is 'The Programmer' and the boy in the green and gold is 'the Tester'.

Key questions

Why is it important to give very clear instructions? Have you ever been given unclear instructions and ended up doing the wrong thing? Why do you think computers need clear instructions?

Lesson starter

Ideally this lesson should take place around a large grid such as:

- An outside painted chess board.
- Grids in your classroom carpet.
- Making masking tape grids one the floor in your classroom.
- Draw a chalk grid either in your classroom or outside.

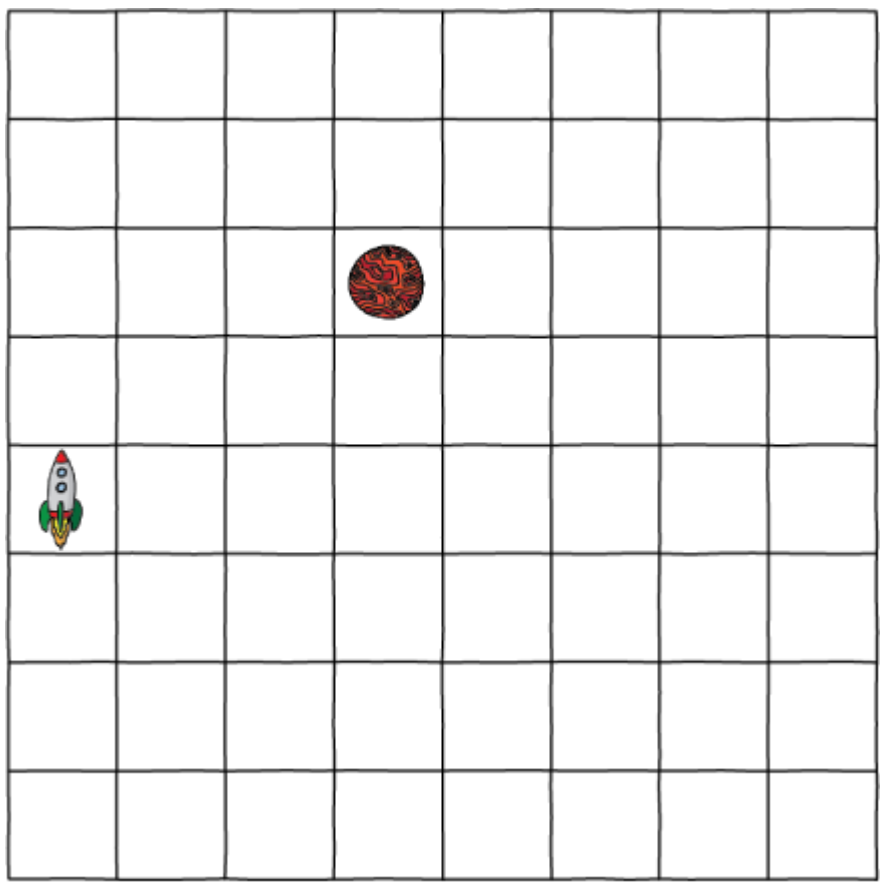
Ask for two volunteers and give yourself and them the roles of:

Role 1: The Developer (who writes the program) - The teacher will model this initially

Role 2: The Tester (who instructs the Bot and looks for bugs)

Role 3: The Bot (who runs the program)

Lesson activities



Teacher: "I'm going to be the programmer, but I'm going to need your help. We are programming the Bot, not just remote controlling it, because ALL the instructions are written before the Bot can follow those instructions."

"It's our job to write down clear instructions for the Bot, who is going to be (say the person's name). (Student's name) will be the Tester and is going to give the instructions to the Bot. The Tester will be on the lookout for bugs."

"First of all we need to decide, what programming language are we are going to use for this? I've chosen arrows to represent move forward, turn left and turn right."

"Debugging is fun because you get a chance to change your program after it's finished when you notice it's not working how

you thought it should."

"First of all I need to decide, what programming language are we are going to use for this? I've chosen arrows to represent move forward, turn left and turn right."

Teaching observations

There will be different ways to express the same instructions (such as drawing an arrow, writing "Forward", or using the printed arrows resource above), and the key is that we need to be consistent. The choices about the exact format of the instructions is what leads to different programming languages, and it's fine that this happens, as long as we know the meaning for the particular language we're using at the moment.

If students aren't sure about the left and right direction, you can print the "left and right hand cards", and stick them to their shoes or have them hold them in their hands.

Have the Bot act out the individual instructions: forward means step one square forward, and left and right mean a 90 degree turn on the spot in the square (not moving to another square).

Teacher: "We're going to write our own program that gets the rocket to fly to Mars. The goal is to get the rocket to the square that Mars is in. Let's write the first two steps on the board together." (Draw two forward arrows.)

Teaching observations

To work it out students may initially need to see the program in action, so place the arrows within the grid to demonstrate what the rocket will do. It's also a great technique to write the first few instructions of the code, test it and then add to it. If you are putting arrows on the grid, the turning arrows will also need a forward arrow in the same box, or you can put the forward arrows on the line between boxes, which makes it clearer what it does.

"So let's try that, and see what happens.

"Tester - could you please take these instructions and pass them onto the Bot. Be ready to underline what doesn't work when you see the Bot doing something that doesn't look right, and hand the whiteboard back to me to figure out how to fix the bug."

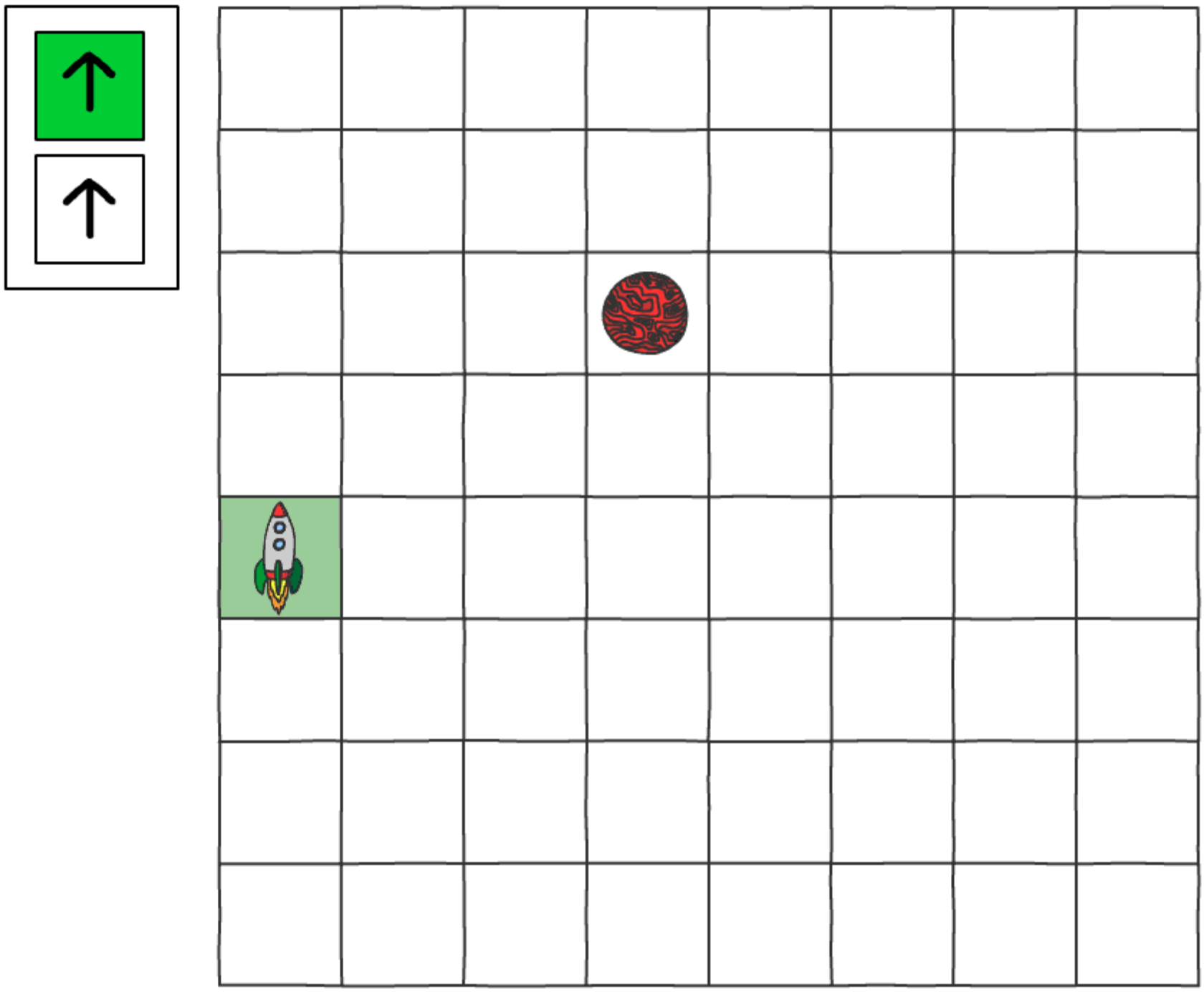
Teaching observations

A key point in this activity is that the instructions are all written before they are tested. We don't allow anyone to give additional instructions to the Bot; they must follow exactly what is written (which can sometimes be humorous if they head off in the wrong direction.) This is what happens when programming: you write instructions for a program, and when you run the program, they are all executed without the programmer intervening. A programmer needs to visualise what would happen when they are writing the instructions; during testing they will find out if what they envisaged was correct!

Teacher: "Bot - please pick up the rocket ready to receive the instructions for the tester." (The bot can carry a toy or token representing the rocket; or they can imagine that they are guiding it).

Tester then reads off the board: "move forward, move forward."

Rocket to Mars program:



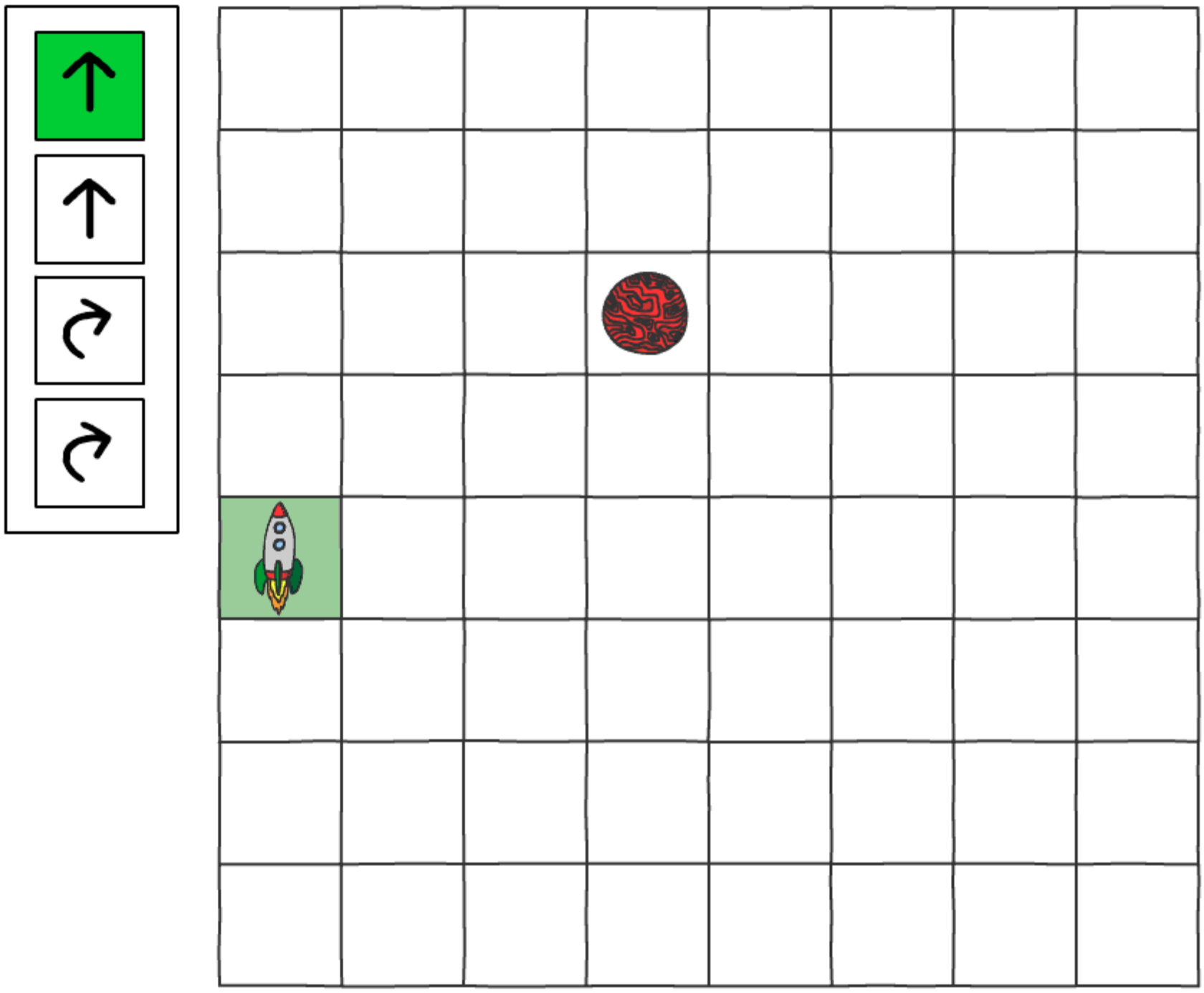
> **Teaching observations**
>
> At this point you could question whether or not a "Stop" instruction is needed. Students should be able to come up with the reasoning that the program stops automatically because there are no more instructions.

"Tester, did the program run as you expected it to?" Depending on the tester's response, if it did then carry on programming, otherwise fix what didn’t work and run that again. In this example the rocket should be in the square three to the left of Mars.

Now let’s add to it. What would we program next?

Point to where the next piece of code needs to be added and add turn right, turn right. (This is deliberately incorrect.)

Rocket to Mars program:



I think it’s ready to test now. Tester, please test my program (the programmer hands the program on the whiteboard to the tester and the bot should return to the starting square ready to rerun the program).

Teacher: Remember Tester, it’s your job to find any "bugs" in my program. A bug is when my program isn’t doing what was expected. Your job is to draw a line under the piece of code where you notice the instructions seem to be going wrong. You can stop the Bot at the point that you think there is a bug.

Tester then reads the instructions in the program off the board and the Bot executes them as they are read.

1. Move forward
2. Move forward
3. Turn right
4. Turn right

Teaching observations

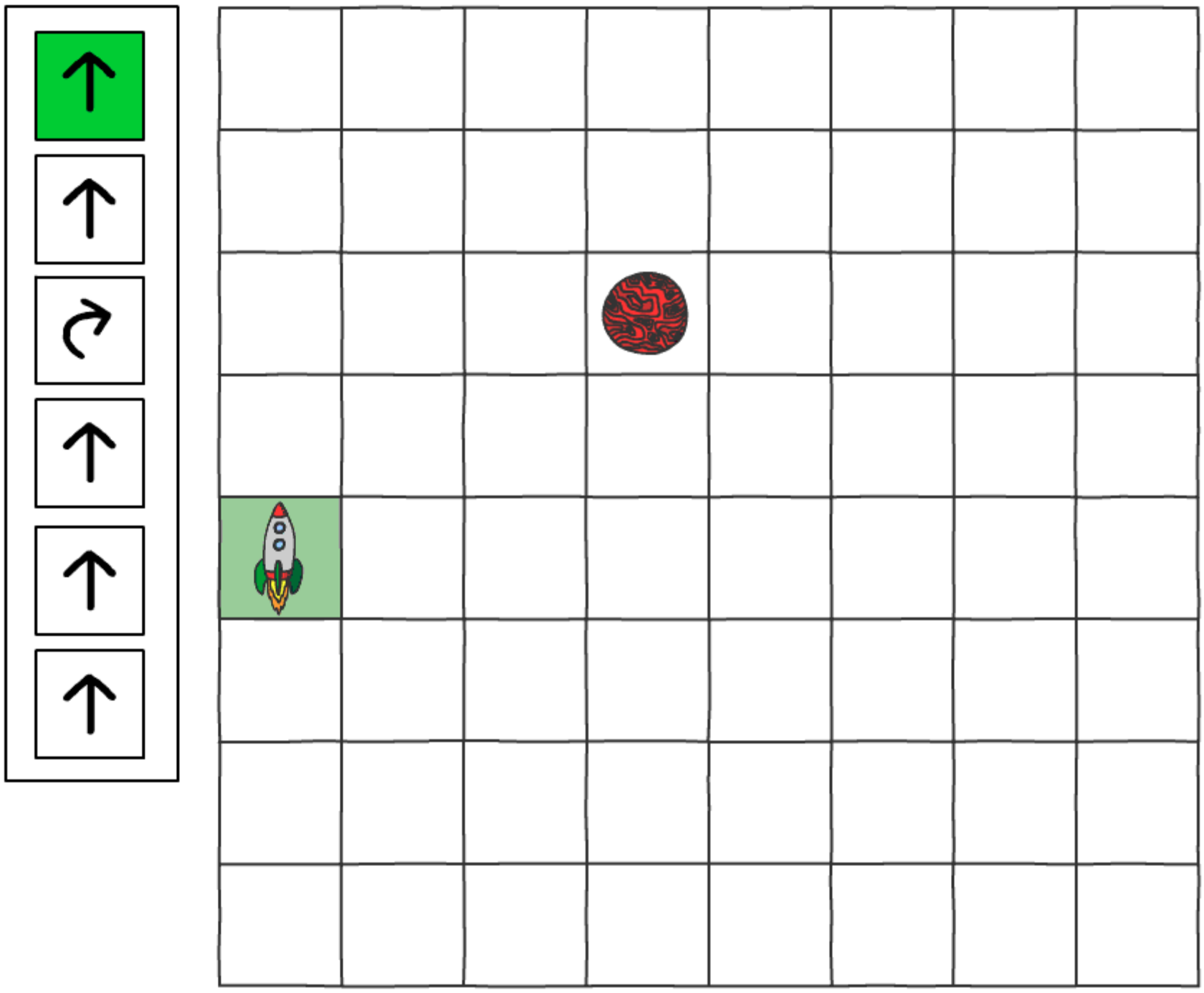
The tester should underline under the second turn right because the rocket will have turned around twice on the spot rather than turning once and going forward again. (Which is what is needed to get to Mars.)

Teacher: "Excellent, you found a bug! I love finding bugs, so I can start solving them. Now class, let’s work through this together to find my bug. Tester, you’ve done a great job finding it, but it’s the programmer’s job to find and fix the bug."

Teaching observations

If the class can’t identify how the program needs debugging then talk through each step and model it with the rocket. Did move forward make sense? Did move forward a second time make sense? Did turn right make sense? What about turn right again? No? Okay I think we found our bug. Let’s draw a line under that and think about what we would change it to? Move forward? Let’s test it.

Once the bug has been identified then ask the Tester to test again. Ask the Bot to pick up the rocket and go back to the start position, then the Tester reads them the instructions.



Did we successfully program the rocket to land on Mars? How do we know?

Teaching observations

We successfully programmed the rocket to land on Mars when the rocket and Mars are in the same square.

Are there other ways we could have programmed the rocket to get to Mars? (There will be lots of ways; for example, Right, Forward, Forward, Forward, Left, Forward, Forward will work.) Discuss the programming options and test each one. What if we want the rocket to get to Mars, and then come back safely?

Teaching observations

In programming, there are numerous ways to program the same thing. Some might be more efficient than others, but all are correct if they achieve the desired result. For example a student might program the rocket to go around the outside of the grid and then go and get to Mars. This is a correct solution, but there's a lot of extra code that's not necessary. More commonly, two programmers might come up with programs that take a similar amount of time to achieve the same result, in this case (for example, "Forward, Left, Forward, Right" and "Left, Forward, Right, Forward" get the Bot to the same place and orientation; both are equivalent programs; there is rarely a single solution that is the best one, and this means that if a student's work looks different to a model answer that might be available, it isn't necessarily wrong. If it achieves the intended outcome, but gets there in a different way it is still correct.

Teaching observations

Either set up your students to be working around the outside of your large grid, or you can use a smaller grid like a chessboard or the back of a 100s board (or print the grid provided with this lesson), in which case the bot moves a counter on the board instead of walking around the grid.

If you have multiple small grids, students can work independently in groups of three (programmer, tester, bot). If you are using a large grid, one group of students can work on their program while another tests theirs. Each group gets to try their program once, and then the next group has a turn while the previous one starts working on debugging their program.

Have the students choose their own two toys (one to be a space travelling object, the other to be the destination) and have them practise this task, as follows.

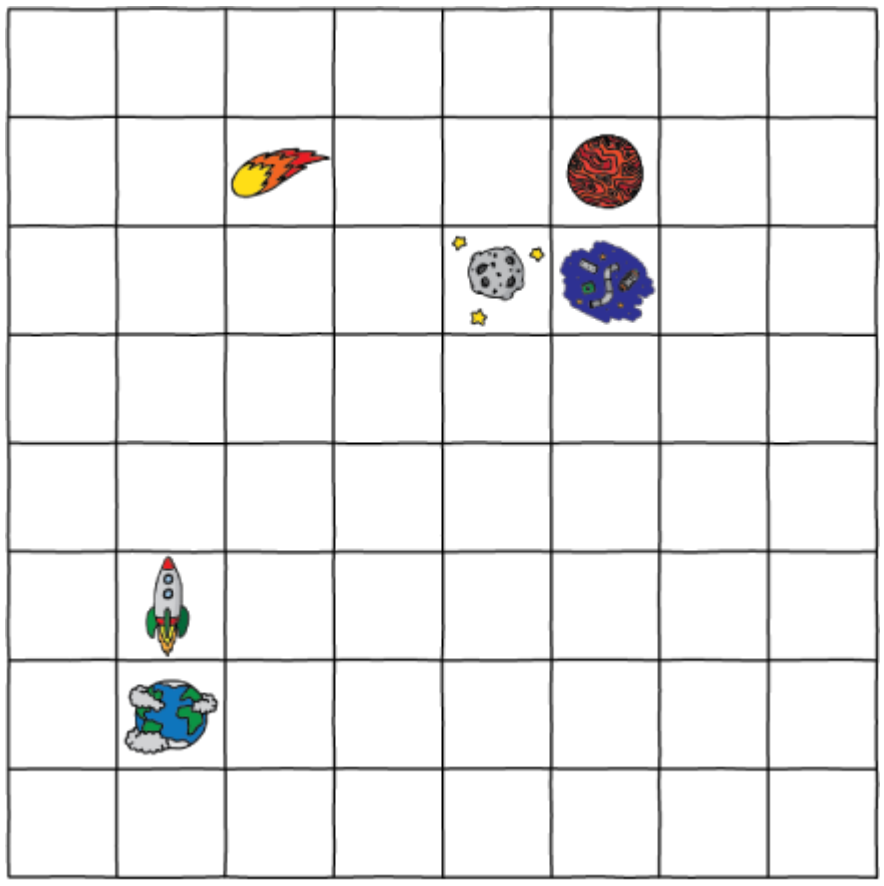
1. Place the traveller on a square on the edge of the grid, facing inwards.
2. Place the destination toy inside the grid.
3. The programmer writes down the program on a whiteboard.
4. The tester then takes the whiteboard and a different coloured whiteboard pen. The tester tells the Bot each instruction in the program. The tester puts a tick next to the code that is correct and underlines when the code is different to what the Bot should be doing. If this happens the Tester says "Stop" and the Bot stops and goes back to the start. The Tester gives the whiteboard to the Developer, who then debugs the code, and gives the Tester a revised version.
5. Repeat step 4 until the program is free of bugs and works as intended.
6. Change roles and move the Bot (space travelling object) starting point and the toy that represents the destination until everyone has had a turn.

Teaching observations

If you notice that your students need support to visualise how to write the instructions you could use the arrows (cards) provided on the ground, or for a small desktop grid, use a whiteboard marker or small arrows. This supports students to visualise what they want to program.

The next challenge

Add barriers to the grid so that the path is more complex because the Bot needs to avoid the barriers. This could be space junk and comets, or you could invent a new scenario for the grid.



Other challenges

Have groups program the trip without using the left hand turn (i.e. the only instructions are Forward and Right turn.) Scaffold the students to realise that a left hand turn can be achieved by doing three Right turn instructions. Then challenge them to program with a Left turn but no Right turn.

Ask if they can write programs with only the Right and Left turn instructions (i.e. no Forward instruction)? (This isn't possible, as you would only be able to turn around on one square.)

Teaching observations

Eliminating one of the turn instructions highlights that different instruction sets can achieve the same thing, although some may be more convenient than others (for example, there are many different programming languages, but they can all do essentially the same kind of computation; it's just that some are better suited to some purposes than others.)

Implementing fewer instructions simplifies building a computer, and this can make it either faster or cheaper. For example, a very simple computer might have an addition instruction, but no multiply instruction; but if you need to do multiplication it can be achieved by doing lots of additions. Many common processors these days tend to have a small number of simple instructions (they are called Reduced Instruction Set Computers, or RISC) because it's more efficient than having lots of instructions (Complex Instruction Set Computers, or CISC).

Applying what we have just learnt

It’s quite common to think that programming is some kind of special talent that people either have or don’t have, but this couldn’t be further from the truth! Like all skills, programming is something you learn through practise, making mistakes, and learning from them. The most important skill that programmers need is to be able to communicate with others, especially when

they are finding and describing bugs. Bugs happen all the time in programming, so being able to identify where the bug occurs and problem solving how to fix it is incredibly important. It doesn't matter how experienced you are at programming, there will always be bugs that need to be found, and fixed. That's why the word "debugging" is so important to programmers.

Lesson reflection

Now that you are all programmers, what did you think was the most challenging part of being a programmer?

Seeing the Computational Thinking connections

Throughout the lessons there are links to computational thinking. Below we've noted some general links that apply to this content.

Teaching computational thinking through CSUnplugged activities supports students to learn how to describe a problem, identify what are the important details they need to solve this problem, break it down into small logical steps so that they can then create a process which solves the problem, and then evaluate this process. These skills are transferable to any other curriculum area, but are particularly relevant to developing digital systems and solving problems using the capabilities of computers.

These Computational Thinking concepts are all connected to each other and support each other, but it's important to note that not all aspects of Computational Thinking happen in every unit or lesson. We've highlighted the important connections for you to observe your students in action. For more background information on what our definition of Computational Thinking is [see our notes about computational thinking].

Algorithmic thinking

Creating a sequence of instructions for this lesson exercises algorithmic problem solving, as it requires students to create an algorithm to accomplish a task. Computational algorithms are based on input, output, storage, sequence, selection and iteration. This exercise focuses on sequencing instructions.

Examples of what you could look for:

Are students able to see what the result will be of their program before it is executed? Did students successfully create a set of step by step instructions for the robot to follow?

Abstraction

In this lesson we have abstracted out writing a program and using a programming language to very basic instructions of move forward, turn left or turn right. Students write these instructions down in familiar words or symbols and give the instructions to the Bot verbally, which removes the need to know how to use a programming language and implement this on a computer. This supports students to understand how sequence works in programming, without being overwhelmed with technical terminology and tools.

Examples of what you could look for:

Can students see the stages of programming (designing the program, testing it, and debugging it), even though the environment is removed from physical devices?

Decomposition

In our everyday lives we don't often give instructions as specifically as saying "Turn right, take a step forward, take another step forward, turn right", and it seems much more straightforward to simply say "please go over there". But when we program we have to be very specific because we have to tell computers exactly how to do each thing and limit ourselves to the few instructions that they can follow. Students need to work on taking something they are familiar with, like "go to that square" and identifying the simplest individual steps that need to happen for someone to accomplish this.

Also, programs can be written incrementally; instead of trying to solve the whole problem, students were encouraged to write a few steps first, test these, and then add to them. Breaking a program down into smaller components makes the task less overwhelming.

Examples of what you could look for:

Do students work on the problem incrementally? Can they take the action of "going to the left" and break it down further to the instructions "turn left, move forward"?

Generalising and patterns

Many different patterns are likely to emerge when students write their programs, and when students recognise these patterns they can reuse some of them multiple times, rather than having to figure out each step again. For example, students may recognise that if they want to turn around 180° they can repeat “left” or “right” twice, and they can repeat this same sequence whenever they need to do this movement. Or if they need to move diagonally across the grid they can take the group of instructions “left, go forward, right, go forward” (right and left swapped depending on which way you’re going) and repeat this as many times as they need to cross the board.

Examples of what you could look for:

When students recognise patterns in their programs and in how they move the bot do they also see that they can reuse these instruction sets whenever they need the bot to make the same movement?

Evaluation

There is a clear way for students to evaluate their programs in this activity, which is simply to ask themselves “did it work?”. By testing their programs they can evaluate whether their instructions accomplish the task.

There are multiple programs students could write that will get the Bot to the goal square, but some of these are probably more efficient than others. This could be measure be either the number of instructions, or the amount of time used.

Examples of what you could look for:

Did students recognise that multiple solutions (many different sets of instructions) are possible? Were they able to compare some and evaluate which ones use fewer commands and are the fastest? Are they able to see that measuring the time taken is a practical measure of the performance of a program, but counting the number of instructions is more consistent? (As it doesn't depend on who has the Bot and Tester roles).

Logic

When students encounter bugs in their programs they will need to logically step through their instructions one by one to locate the bug. They will need to think about what they expect to happen when each instruction is followed, and if they do not get the result they expected they will need to identify what went wrong, why it went wrong, and how to fix it. This requires students to apply their logical thinking skills.

Examples of what you could look for:

Do students go through their instructions and predict what will happen each time one is executed? When they debug their instructions do they use a logical method to do this? Such as stepping through each instruction one by one and checking if what they expected to happen does happen, or comparing their program to previous version which didn’t seem to contain a bug?