Scratch Activity: Fundamental Coding Concepts

CS4S - Introduction Workshop @ the University of Newcastle

# Introduction

The fundamental *Coding* concepts we will introduce you to in this activity are:

* *Algorithms and Programs*
* *Sequencing*
* *Repetition*
* *Variables*
* *User Input*
* *Branching*

These will be explained during the activity, but you can also find definitions and examples on the [Glossary page of the workshop website](https://cs4s.github.io/intro-2017/glossary/).

# The Project

We will create a Scratch project that combines all the concepts mentioned in the previous section. You can see the finished project [on the MIT Scratch website](https://scratch.mit.edu/projects/173543499/).

Click the green flag to run the program, answer the dancing cat's question and see what happens.

What happens when you try the following?:

* Answer the cat's question with a small number (for example: 10)
* Answer the cat's question with a negative number (for example: -1)
* Put in a very large number (for example: 500)

# Algorithms and Programs

In this activity, we will take an *algorithm* and create a *Scratch program* that follows that *algorithm*. An algorithm is a precise and step-by-step procedure that is written for a computer to follow. Algorithms are often compared to recipes, as recipes also involve detailed, step-by-step instructions.

Usually when people talk about *algorithms*, they are referring to the instructions that the computer will follow. These instructions could be created in a variety of different ways. For example: written out with paper and pen or in a flowchart diagram. A program is an algorithm that has been written in a *Coding* language (for example: *Scratch*) and can be run by a computer.

For example: we may write the following steps for someone to follow (an algorithm), to draw a square:

* draw a horizontal line 10cm in length anywhere on a piece of paper
* rotate the piece of paper clockwise by 90 degrees
* draw another horizontal line 10cm in length, starting from the end of the last drawn line
* rotate the piece of paper clockwise by 90 degrees
* draw another horizontal line 10cm in length, starting from the end of the last drawn line
* rotate the piece of paper clockwise by 90 degrees
* draw another horizontal line 10cm in length, starting from the end of the last drawn line

However, these instructions are not really a *program* until they are written in *Code*, as shown in the *Scratch* blocks below:



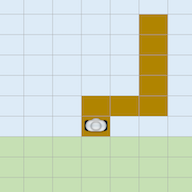
To create *programs*, you not only need to know how to create precise and step-by-step instructions that a computer can follow to complete a task (which is what we call *Computational Thinking*), but also be able to take those instructions and put them in a language that the computer can understand (which is *Coding*).

In the next section, we will look at some *Computational Thinking* concepts, that are essential for understanding how to write *algorithms*, and then use them for *Coding* our Dancing Cat program.

# Sequencing

## Why is Sequencing Important?

One of the most important concepts in this activity (and *Coding* in general) is *sequencing*. Computers not only need precise instructions to complete tasks - these instructions need to be in the right order. For example, say we have a robot that we are *Coding* to navigate a jetty over water, as shown in the image below.



In the image above:

* The robot is the silver shape
* The brown squares represent the jetty
* The blue squares represent the water
* The green squares represent land

To help the robot move along the jetty, we first want the robot to move up by one square and then move to the right by one square. But what would happen if we told the robot to follow these instructions the other way around? The robot would move right by one square, then move up by one square. When we *Code* computers, they will do exactly what they are instructed to do - so in this example the robot would end up in the water!

## Sequencing in Scratch

After clicking the green flag in *Scratch*, the steps in a *Scratch program* will run from top to bottom through a stack of blocks. For example, after clicking the green flag, in the stack of Scratch blocks below, the cat will:

1. say *Hello* for two seconds
2. say *How are you going?* for two seconds
3. say *That's good!* for two seconds



You can also get the cat to do the same thing, by dragging away the "when green flag clicked" block and clicking the blocks, as shown below:



## The Dancing Cat

Now, we will start creating our Dancing Cat *Scratch program*. During this section we will focus on how blocks in Scratch are run in sequence.

Create a new *Scratch program* and name it "Dancing Cat".

In the Scripts Area, add the following blocks:



Click the green flag and see what happens.

The Cat should move to the right in one movement. This is often unexpected - people usually expect the Cat to move in separate steps or to do 2 movements across the screen.

The *move* blocks above do not mean that the Cat will take 40 separate steps but that the Cat will move 40 pixels to the right in one movement. Also, while we have two separate *move* blocks in the code above, the Cat does not wait between these blocks. So the Cat moves 80 steps in total, without pausing.

Remember that when *Coding*, the instructions we give to a computer must be precise and direct. To get the Cat to pause before the second *move* block, we must add a *wait* block.

Change the code in your *program* so that it looks like this, by adding a *wait* block.



Click the green flag and see what happens.

Now the Cat will move 40 steps to the right, pause for half a second and move another 40 steps to the right.

We want to make the Cat dance, rather than just move to the right. To do this, change your stack of blocks to this instead:



Click the green flag. Now the Cat will move to the right and then move back to its original position.

## Adding Sound

Now, instead of using *wait* blocks, we are going to play a drum each time the Cat moves.

The different *Sound* blocks work in different ways. For example, create another stack of blocks, separate to your other blocks, that looks like this:



Click on the stack of blocks and you should hear one "meow" sound - although you might have expected two. The *play sound* block is like the move steps block - there is no pause between them.

To hear two meows you could change the stack of blocks to this instead:



Instead of making the Cat meow, however, we are going to play a drum every time the Cat moves. To do this, we will change our stack of blocks following the when green flag clicked block to this:



Note that, instead of using the *wait* block after the cat moves to the right, we can play a drum and the Cat will wait until that drum noise has finished before moving back to the left.

In the next section, we will look at how we can make the Cat repeat this dance over and over again.

## Checking Your Understanding

Now that we are the end of the *Sequencing* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website.

### Sequencing: Exercise 1

Look at the Scratch code below.



How many times will the Cat pause before moving again? What are the total number of steps that the Cat will move?

### Sequencing: Exercise 2

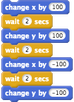
Now, the Scratch code is changed to this:



What will happen when the stack of blocks is clicked? Will the Cat move to the right, then the left, the right and the left again? Or will the Cat appear to not move at all?

### Sequencing: Exercise 2

You ask a student to explain how the Cat moves around the canvas when it follows a stack of *Scratch* blocks. This is their *program*:



The student explains that, when the top of the stack of blocks is clicked, the Cat will:

*Move to the right, wait for a couple of seconds, then move up a bit. It will wait again for a couple of seconds, then move to the left. It will wait again and finally move back down to where it was at the start.*

Is this explanation correct?

# Repetition

## What is Repetition in Coding?

Another important Coding concept is *repetition*. Computers are very good at repeating actions, whether you want them to repeat something two times, one hundred times, or even to repeat something over and over forever.

There are many examples of repetition in real life, as well as in *Code*. For example, when you swim 10 laps in a pool, you are repeating the lap 10 times. You would also be repeating the actions that make up the stroke (for example: freestyle) over and over as you swam the lap.

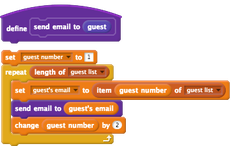
Another example of *repetition* could involve repeating actions for a list of people. You may organise a birthday party and decide to write personalised emails to all of the guests. To send out all these emails, you may follow these actions until you have worked your way through the guest list:

1. Look at the next guest on the list
2. Write an email addressed to that guest
3. Go back to step 1

In *Coding*, we use loops to repeat actions. *Scratch* has three different types of loop blocks: *forever*, *repeat* and *repeat until*. Every programming language has some way of allowing *repetition* of actions, which are usually loops, but we won't worry about that in this activity.

## An Example Program that uses Repetition

The Scratch blocks below show the above steps (*algorithm*) as a *Scratch program*. Don't worry about trying this out in your own *Scratch program* - just read the blocks below and see how they relate to the 3 actions above. The example also includes blocks and concepts that we haven't explored. These blocks and concepts will be explained later in the tutorial, but I will also briefly explain what the blocks do in the paragraph below the blocks.



In the blocks above, the big purple block is a *custom block* that would have blocks to create and send an email to a party guest (which aren't shown in the example above).

The *guest list* is a list of party guests' emails. We could, for example, have a list of three party guests' email addresses in this list: *bob.smith@gmail.com*, *rita.green@gmail.com*, *joan.blue@gmail.com*. The *guest number* is a variable (we will look at variables later in this activity) that changes by 1 each time the repeat loop runs its actions. The *guest's email* is another variable, which will be used to keep track of each of the *guest's email* that the invitation will be sent to.

The way this Code in the blocks will work is:

**Before the repeat loop:**

* The *guest number* variable will be 1

**The first time the instructions in the repeat loop are followed:**

* The actions in the *repeat loop* will run for the first time
* The *guest's email* variable will become the first item of the *guest list* (*bob.smith@gmail.com*)
* An email will then be sent to bob smith's email address
* The *guest number* will change by 1 (to 2)

**The second time the instructions in the repeat loop are followed:**

* We go back to the first block after the *repeat* block and the actions will run for the second time
* The *guest's email* variable becomes the second item of the *guest list* (*rita.green@gmail.com*)
* An email will then be sent to rita green's email address
* The *guest number* will change by 1 (to 3)

**The third (and final) time the instructions in the repeat loop are followed:**

* We go back to the first block after the *repeat* block and the actions will run for the third time
* The *guest's email* variable becomes the third item of the *guest list* (*joan.blue@gmail.com*)
* An email will then be sent to joan blue's email address
* The *guest number* will then change by 1 (to 4)

In total, the actions in the repeat block above will run three times for this example, as the number of email addresses in the guest list is three.

## Loops in Scratch

Recall that, as mentioned above, *Scratch* has three different types of loop blocks: *forever*, *repeat* and *repeat until*. In this section of the activity, we will focus on the *forever* and *repeat* blocks.

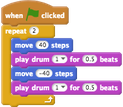
Change the blocks that makes your Cat dance to look like the blocks below:



This will make the Cat dance forever (or until you click the red stop icon next to the green flag).

We don't want the Cat to dance forever, and so we will use a *repeat* block instead.

Change the blocks so they look like this instead:



We are now using a *repeat* loop block, instead of a *forever* loop block. The *repeat* block has a slot for a number (2 in the example above), which determines how many times the actions in the loop repeat. Now the Cat repeats the movement back and forth twice and the drum sound is played 4 times. Try changing the 2 in the *repeat* loop block to different numbers and running the program. What happens when you use larger numbers (for example: 10). What happens when you use numbers like 0 and -1 in the *repeat* block and run the program?

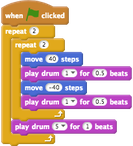
## Nested Repetition

We can also put loops inside loops - this is called *nesting loops* or *nested repetition*. For example, create a separate stack of blocks to your blocks that make the Cat dance, that look like this:



Click the stack of blocks and count the drum beats. You should hear 8 drum beats. Why does this happen? The *repeat (4)* block repeats the actions inside the loop four times, these actions being the *repeat (2)* loop, which in turn plays the drum noise 2 times. Therefore, the drum noise is played 8 times in total. So, when you have two nested repeat loops, the actions inside the inner loop block (*play drum (1 v) for (1) beats* in the example above) will repeat for the number in the outer *repeat* loop (4 in the example above) multipled by the number in the inner *repeat* loop (2 in the example above).

We are now going to add an extra step to our Dancing Cat program. Now, we are going to change the dance so that the Cat will move back and forth twice, a hi-hat noise will be played for 1 beat and the Cat will then repeat these steps. To do this, we will need to use *nested loops*. The program to make the Cat dance should be changed to look this:



Now when you click the green flag, the Cat should:

1. move to the right and play a snare drum noise
2. move back to the left and play a snare drum noise
3. repeat the last 2 steps
4. play a hi-hat noise

Then the Cat will go through steps 1-4 again.

## Checking your Understanding

Now that we are the end of the *Repetition* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website

### Repetition: Exercise 1

Look at the blocks below.



How many times will the snare drum noise play?

### Repetition: Exercise 2

You ask a student to explain what the Cat will do when you click the stacks below.



The student says that the Cat will move to the right 5 times when you click on the stack of the blocks. This is an incorrect explanation of the above blocks for two reasons, why is that and what will actually happen?

### Repetition: Exercise 3

Look at the blocks below.

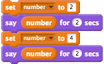


How many times will the Cat move to the right? How many times does the outer loop repeat and how many times does the inner loop repeat?

# Variables

## What are Variables?

In *Scratch*, and *Coding* in general, a *variable* is a name for a value (for example: a number like 5) that can change. For example, look at the blocks in the below image.



In the example above, we have a *variable* called *number* that changes as we move through down the stack of blocks. What happens when you click this stack of blocks?

1. The *number* variable's value is set to 2.
2. The cat will say the value of *number* (2 at this point) for 2 seconds
3. The *number* variable's value is then set to 4
4. The cat will say the value of *number* (4 at this point) for 2 seconds

*Variables* are an essential *Coding* concept that are used in most programs. For example, if we made a basketball game in *Scratch* and wanted to keep track of each team's score, we would need to use two *variables*: 1 number *variable* for each of the teams. These *variables* (let's call them *Home Team Score* and *Away Team Score*) would start with a value of 0 and then every time one of the teams scored a basket, the value of the appropriate *variable* would be increased by 2.

There are different types of *variables* in different *Coding* languages and sometimes *variables* work differently in other languages. However, in this activity we will focus on how they work in *Scratch*.

*Variables* can take on three different types of values in *Scratch*:

1. **Boolean**: the value of the variable is either *true* or *false*
2. **Number**: the value of the variable is an integer (a whole number like *2* or *3*) or a decimal number (for example: *1.34* or *2.5*)
3. **String**: the value of the variable is a piece of text (for example: *Hello* or *Dan*)

This is important, because the different blocks in *Scratch* expect variables with specific types of values. For example, the blocks below expect a *variable* with a *Boolean* value to be placed in the empty slot/s:



Notice that in the above blocks, the slots where the blocks can be placed are shaped the same.

In the next example below, the blocks pictured expect *variables* with *Number* values to be placed in the empty slot/s.



Notice that, like the *Boolean* value example, the empty slots are the same shape as each other (although they are more circular than the *Boolean* values slots).

In the next example below, the blocks expect a *variable* with a *String* (text) value:



These all have the same shaped empty slots as well. However, for *String* values, the empty slots are rectangular shaped.

Knowing the type of values that should be used in the different blocks can be important when trying to find out why Code is not doing what you expect to do. For example, read the blocks in the stack below, which involves three different variables (*Introduction*, *Name* and *Result*):



This could be an example of some Code that a student puts together that is supposed to have the Cat introduce themselves. You might expect that, because we are adding *"Hello. my name is "* and *"Dan"*, that the value of the *Result* variable would be: *"Hello, my name is Dan"*. Instead, because the add block expects *Number* values - not *String* values, the value of the *Result* variable is actually *0*. To make the sentence *"Hello, my name is Dan"* with the *Introduction* and *Name* variables we would use the *join* block instead, as shown below:



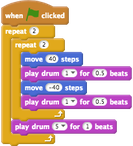
## Adding a Step Counter to Our Program

Let's imagine that our cat wants to know how many movements it makes when it is dancing. This would work similarly to a pedometer that keeps track of how many steps you take each day. We will create a *variable* called *number of movements* that will keep track of how many movements the cat makes.

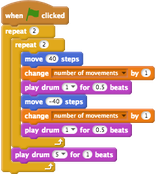
To create the *variable*, follow these steps:

1. Go to the *Data* blocks section of the *Scripts* tab
2. Click the *Make a Variable* button
3. Name the variable *number of movements*
4. Tick the *For this sprite only* checkbox
5. Click the *OK button*

We want to add 1 to this *variable* every time the Cat moves, using the *change variable* block Our Dancing Cat program is shown below. How many *change variable* blocks should be added below and where do you think those blocks should be placed?



Two *change variable* blocks should be added to your Dancing Cat program and each of these should be placed after one of the *move* blocks, as shown below:



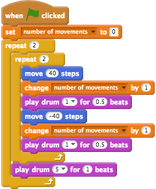
Click the green flag and watch the *number of movements* variable's Stage monitor. Notice that the *number of movements* *variable* increases by 8 every time the Cat does its dance from start to finish.

## Resetting the Movement Counter

If you click the green flag a few times, you will notice that the *number of movements* keeps increasing. For example, if you have clicked the green flag three times, the *number of movements* *variable* will be 24 (8 times 3).

We now want to reset the movement counter at the beginning of our program. Setting the value of a *variable* at the beginning of a program is called *initialisation*, which is good *Coding* practice. Another common example of *initialisation* is that when you play a video game and it ends, when you start a new game your score will be reset to 0.

To reset our movement counter, we will add a *set variable* block after the *when green flag clicked* block, as shown below:



Now, run the Dancing Cat program a few times and you will notice that each time the *number of movements* *variable* will start at 0 and finish at 8.

## Repeating using a Variable

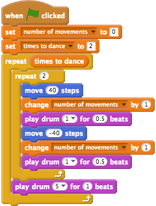
*Variables* can be placed in any of the blocks that have an empty slot. For example, we can use a *variable* in the *repeat* block, which will determine how many times the actions in that *repeat* block are repeated. In this part of the activity, we will create a *variable* called *times to dance*

Create a variable called *times to dance* by following these steps:

1. Go to the *Data* blocks section of the *Scripts* tab
2. Click the *Make a Variable* button
3. Name the variable *times to dance*
4. Tick the *For this sprite only* checkbox
5. Click the *OK button*

To use our *times to dance* variable in our repeat loop, we will need to add two blocks. The first block will be a *set variable* block, where we will set the *times to dance* *variable* to the number of times we want the Cat to dance. The second block will be the *times to dance* *variable*, which can be found in the *Data* section of the Scripts tab.

You can see the stack of block with these two blocks added below:



Change your *program* so that it includes these blocks and matches the stack above. Notice that if you change the *2* in the *set times to dance* block and click the green flag, the Cat will repeat its dance that many times (unless you use 0 or a negative number).

In the next section, after the *Check Your Understanding* questions, we will look at *user input*. We will change our *program* so that the person using the Dancing Cat *program* can tell the Cat how many times it should dance.

## Check Your Understanding

Now that we are the end of the *Variables* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website.

### Variables: Exercise 1

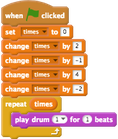
Have a read of the code blocks below:



There are five *variables* in the code blocks above. What are they named, what type (*Boolean*, *Number* or *String*) are they and what will their values be after the last block?

### Variables: Exercise 2

Read the blocks below, which involve a *variable* (*times*) that is used in a *repeat* block.



What is the value of the *times variable* after the last block in the stack above? How many times does the drum noise play?

### Variables: Exercise 3

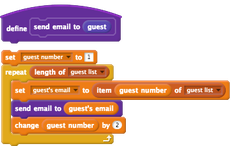
A student has created a short program as part of a bigger game project. They currently have the following blocks:



The student wants to change the blocks so that every time they press the space bar, the value of the *time space bar pressed* variable decreases by 1. How could they change the blocks above to do that?

### Variables: Exercise 4

You have decided to use the blocks from the example given in the *repetition* section, where you had a guest list and where sending invitation emails to your guests. Let's say that you tried to remember the blocks from the previous example and ended up with the following stack of blocks:



You send out the invitations using this Scratch program but you only recieve RSVPs from half of the invited guests. Why could this be?

# User Input

## What is User Input?

*User input* is a term that means that the person using the program (the user) interacts with the program, usually by (but not always) using a keyboard/mouse.

For example, you might register for an event (for example: a workshop) using an online form. The *user input* in this example would be the details you would enter into the form (such as your name and email address). Another example of user input is controlling a character in a videogame with a joystick. The videogame would have some *Code* that would take that input from the joystick (for example: moving the joystick to the right) and then perform some action (in this example: moving the character to the right).

In *Scratch*, there a few different ways to get *user input*. Examples of *user input* in *Scratch* include:

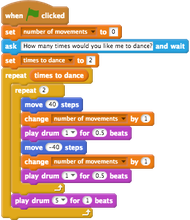
* Asking the user a question with the *ask and wait* block in the *Sensing* section.
* Detecting when a user has pressed a key, with the *when key pressed* block in the *Events* section and *if key pressed?* block in the *Sensing* section.
* Detecting when a user has clicked on a sprite (for example: when a user clicks on the Cat) with the *when this sprite clicked* block.

In this section of the activity we will focus on the first example: asking a question.

## Adding a Question to Our Program

We are going to make our Cat ask the user how many times they would like the Cat to dance.

The Cat will need to ask this question before the *repeat* block. The stack of blocks below include the *ask and wait* block.



To make your stack of blocks match the above image, make sure that you:

* Drag an *ask and wait* block from the *Sensing* section of the *Scripts* tab and place it after the *set number of movements to 0* block.
* Replace the text in the *ask and wait* block with: How many times would you like me to dance?

As we already have a *variable* - called *times to dance* - that is used to determine how many times the Cat will perform its dance, we will now set that *variable's* value to the answer given by the user. For example, if the user answers the Cat's question with 2 - we want the value of the *times to dance variable* to become 2.

There is a special block in Scratch called *answer*, which you can find in the *Sensing* section of the *Scripts* tab. When we use an *ask and wait* block, the *answer variable* will be set to the value given by the user to this question. For example, read the stack of blocks below:



Say that you click the stack of blocks above and the Cat asks: What day of the week is it? If you then typed in the name of a day, for example: Monday, the Cat would then say: Yes! It is Monday

Now, we will use the answer given to the cat (the *user input*) to set the value of the *times to dance* *variable*. To do this, drag the *answer* block from the *Sensing* section and replace the 2 with the *answer block*. Your *set times to dance* block should now look like this:



Click the green flag button, answer the Cat's question and watch the Cat dance. Does the Cat move the number of steps you expect?

After the Cat has finished its dancing, the value of the *number of movements* should be: the value of the *times to dance* variable multiplied by 4.

## Check Your Understanding

Now that we are the end of the *User Input* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website.

### User Input: Exercise 1

A student has written the following *Code*:



What will happen when you type *John* in the answer box and press enter?

### User Input: Exercise 2

A student has the following stack of blocks, which are part of a bigger game project. They say that the *Code* is not working properly but they don't know why.



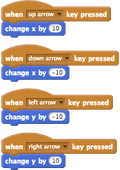
The student also shows you an example of what is confusing them:

* The student clicks the green flag and the *program* starts
* The student enters their name in when the Cat asks their name (for example: *Dan*) and presses enter
* The Cat says: New player = *Dan* and the *program* ends
* The student asks their friend (who is named *Ben*) if they want to play the game
* Ben clicks the green flag and the Cat says: New player = *Dan*, then asks: What's your name?

The student wants the Cat to ask the player's name then say: New player = *player name* but the *program* does not work this way. What do you think is causing this problem to happen and how it can be made to work as the student expects it to?

### User Input: Exercise 3

Another student is creating a game where the user has to navigate a Cat through a maze. They are currently working on the *user input* Code, shown in the blocks below, and are finding that when they try and move the Cat with the arrow keys, the Cat is not moving as the student expects.



What will they need to change to get the Cat to move as expected? Hint: when the user presses the right arrow key, will the Cat move to right? Why or why not?

# Branching

## What is Branching?

*Branching* in Coding refers to the computer following different instructions based on certain conditions. *Branching* is also sometimes called *selection*.

There are many examples of *branching* in real life. For example, say that you are hosting a barbecue and a friend has offered to buy the bread from the supermarket. You have heard that there has been a shortage of loaves of bread in the local area and so you give them the following instructions:

* If there are loaves of bread at the supermarket, buy a loaf of bread
* If there's no loaves of bread at the supermarket, buy 3 packs of breadrolls instead

This example is *branching* because, depending on the condition (the availability of loaves of bread), different instructions are followed.

In *Scratch*, the main blocks to use for branching are the *if then* and *if then else* blocks. There are also many examples of branching in *Scratch* programs, as they are an essential *Coding concept*. One example of *branching* in *Scratch*, that might be part of a bigger *Scratch* project, could be blocks that say different messages depending on how many lives a character has left in a video game.



This stack of blocks above could run every time a character loses a life (for example: by falling in water) and the *number of lives* *variable* is consequently decreased by 1. When the character has 0 lives left, the message: Game Over! will appear. Otherwise, the message about the *number of lives* left will appear.

## Dealing with Negative Numbers

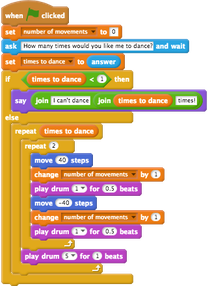
We will use *branching* to deal with what is called *invalid input*. *Invalid input* is *user input* that is nonsensical or that causes the program to do unexpected things. In this example, we will look at a specific *invalid input*: answering the Cat's question about how many times to dance with a negative number.

What happens if we answer the Cat's question (*How many times would you like me to dance?*) with a negative number (for example: -2) or 0?

If you try it and see what happens, you may notice that the Cat doesn't do anything. This is because the Cat can't dance a negative number or 0 times.

We will add *branching* to our Dancing Cat program, so that the Cat tells the user that they need to input a positive number.

Look at the *Scratch* blocks below, where an *if then else* block, a *say* block and two *join* blocks have been added to the *program*:



Change your Dancing Cat *program* to match the above. Once you have made these changes, answer the Cat's question with a number like 0 or -2. The Cat should say that it cannot dance that many times and refuse to dance.

If you enter a number larger than 0, such as 4, the Cat will follow the instructions (the dance steps) that are inside the *else* block.

## Adding an Upper Limit to the Times to Dance

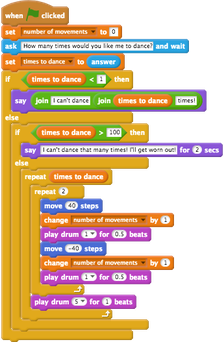
Now, let's say that the Cat will not only refuse to dance 0 or for a negative amount of times, but that it will also refuse to dance more than 100 times - because if it dances this much it will get worn out!

To add this upper limit we are going to have to use *nested branching*. *Nested branching* in *Scratch* involves having *if then* and/or *if then else* blocks inside of other *if then* and/or *if then else* blocks. This is similar to the *nested loops* we looked at earlier, where we have *repeat* blocks inside other *repeat* blocks.

We want to change our Dancing Cat *program* so that the following happens:

* The Cat asks how many times you want it to dance
* We check if the user answers with 0 or a negative number
* If the answer is 0 or a negative number, the Cat tells the user they can't dance that many times and the *program* ends
* Otherwise (if the answer is positive and not 0), we then check if the answer is larger than 100
* If the answer is larger than 100, the Cat tells the user they can't dance that many times and the *program* ends
* Otherwise, the answer is not larger than 100, the Cat then dances for the number of times the user answered with and then the program ends

To turn this *algorithm* into *program code* we need to add another *if else then* block, as shown in the blocks below:



Have a look at the bullet points above that describe the *algorithm*, the *Scratch* blocks and compare the two. Can you see how these are related to each other, and how the different instructions will be followed based on the *user input*?

## Check Your Understanding

Now that we are the end of the *Branching* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website.

### Branching: Exercise 1

Look at the *program* below that asks the user to enter a secret password, in order to get access to some secret documents.



What will happen when the user answers the question (*What's the secret password?*) with: *12345*?

### Branching: Exercise 2

A student creates a game and decides that only people aged 5 to 95 are allowed to play it. They create the following stack of blocks.



What will happen if the game player answers the question (*What's your age?*) with 10? What will happen if the player is aged 4? Or 96?

The student also notices that, regardless of what age the player is, the message: *Let's start the game!* appears. The student only wants this message to appear if the player is the right age (aged between 5 and 95) to play the game. To get the blocks to work this way, they will have to move the last *say* block. Where should the *say* block be moved?

### Branching: Exercise 3

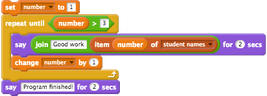
You have a list of 30 student names that has been sorted by their scores in a recent quiz, in descending order. The first student name is the name of the student who scored the highest mark in the quiz. The second student name is the name of the student who scored the second highest mark in the quiz, and so on.

You would like to create a *program* where the Cat congratulates the students who scored the three highest marks in the quiz. The three highest-scoring students' names are: *Sophie*, *Tim*, and *Mary*, and consequently they are the first three names in our *student names* list variable in the blocks below.

You find out about the *repeat until* block, which you think you will be able to use to create this program. The *repeat until* block combines a few of the *Coding* concepts in this activity, including *Repetition* and *Branching*.

Unlike the *forever* and *repeat blocks* however, actions inside a *repeat until* block repeat until some condition becomes true. An example of repeating an action until a condition is true in your everyday life is eating a meal until you're full. When eating cereal, you may repeat the same action (for example: eating a spoonfull of cereal) until you are full. Another example would be swimming laps in a pool until you are too tired to continue swimming.

To create the program where the Cat congratulates the top three students, you create the following stack of blocks:



Can you explain, step-by-step, what happens when these blocks run with a *list variable* called *student names*? The first three students in the *student names* list are: *Sophie*, *Tim* and *Mary*.

# Conclusion

You have now reached the end of this activity - good work!

You have learned about the following fundamental *Coding* concepts in this activity:

* Algorithms and Programs
* Sequencing
* Repetition
* Variables
* User Input
* Branching

While the *program* that we have created is relatively simple (in comparison to a video game like *Space Invaders* or *Tetris*, for example) you can see that every concept listed above was involved in the design of the program. As you learn more about *Coding* and see more *programs*, you are likely to see every one of these concepts present in most *programs*.

It is important to remember and understand these concepts because they are foundational to *Coding* and *Computational Thinking*. You may also notice that these are the terms used through the national curriculum's [Digital Technologies content descriptions](http://v7-5.australiancurriculum.edu.au/technologies/digital-technologies/curriculum/f-10?layout=1) and we hope that learning from this activity helps you when you read these.

If you would like to work through this activity again to practice these concepts or to refresh your memory, this activity will be available indefinitely on the workshop website. We have also created a glossary webpage on the workshop website, that you can use to review some of the meaning of the terms introduced in this activity.