Supplement to the book

"Advanced Scratch Programming"

(Author: Abhay B. Joshi)

Review of Computer Science and Scratch Concepts

Introduction

The projects covered in the book "Advanced Scratch Programming" by Abhay B. Joshi are all based on the CS and Scratch concepts listed below. I assume that you are already familiar with these concepts. If not, or if you want to brush up on these concepts, please refer to the brief description of each concept provided in this appendix. This is NOT a rigorous and comprehensive explanation of concepts, but only a quick summary.

The Internet is replete with study material, videos, and online courses on Scratch programming and Computer Science. I have listed below a few general Internet-based references that you could use to study Scratch and CS concepts in more detail.

Scratch Offline Editor:

This editor itself offers help on every feature of Scratch. Click on the "?" symbol in the upper right corner to open this help. You will find step-by-step tutorials, how-to guides, and explanation of every command block of Scratch.

Scratch Website (scratch.mit.edu):

This website is an authentic source of information related to Scratch. Click on the "Help" button to access user guides, frequently asked questions, help with scripts, and video tutorials.

ScratchEd Website (scratched.gse.harvard.edu):

This website is an online community for educators and it offers stories, discussions, and resources such as the Scratch curriculum guide.

Scratch Wiki Website (wiki.scratch.mit.edu):

This website contains a wide variety of articles by Scratchers for Scratchers, including advanced topics and tutorials.

YouTube videos:

There are literally hundreds of YouTube videos that you can view to understand basic concepts of Scratch. On the YouTube website, search for “Scratch programming” to get a list of useful videos.

Now, let’s take a summary view of each concept individually.

Algorithms

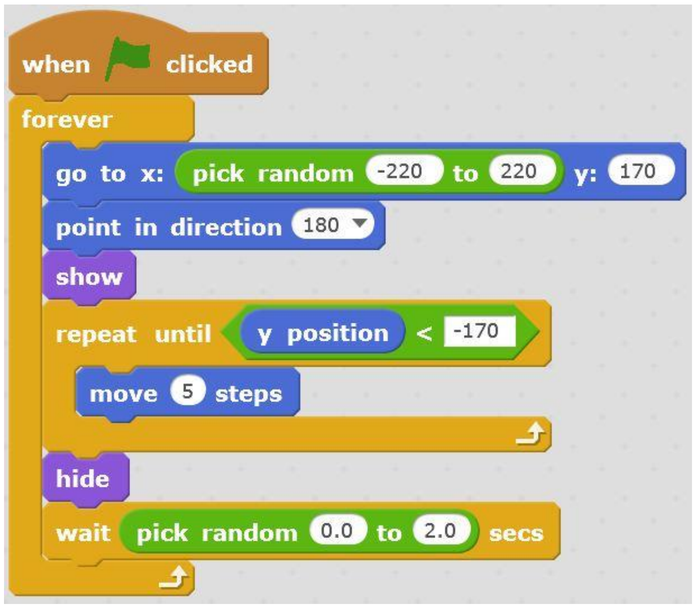
An *algorithm* is a step-by-step procedure that describes how a certain task can be performed. Strictly speaking, an algorithm has a formal structure (sometimes called the *pseudo-code*), which includes the initial and final states of the procedure, description of all inputs and the output, and so on. But, for the purpose of the programming projects in this book, we use algorithms rather loosely to create an informal *high-level description* of program steps, and help us in the process of creating the final Scratch scripts.

Here is an example:

<u>Algorithm to make a *rainfall*</u>:

```
Forever:
   Go to a random point on the top edge
   Face downward
   Become visible
   Fall slowly to the bottom
   Become invisible
   Wait for a random amount of time
End-forever
```

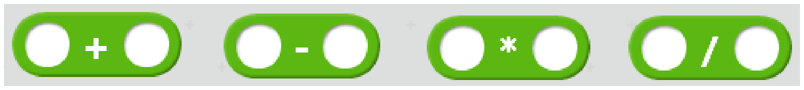
<u>Resulting Scratch script</u>:



```
when [flag] clicked
forever
   go to x: (pick random (-220) to (220)) y: (170)
   point in direction (180 ▼)
   show
   repeat until <(y position) < (-170)>
      move (5) steps
   end
   hide
   wait (pick random (0.0) to (2.0)) secs
end
```

Arithmetic operators (+, -, *, /) and expressions:

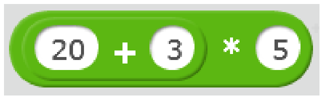
In the operators tab of Scratch, you will find the arithmetic operators:

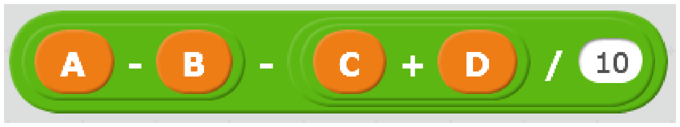


All of us know these operators.

They can be used to create complex arithmetic expressions, such as,
(20 + 3) x 5
(A – B) – ((C + D) / 10) where A, B, C, D are variables.

Equivalent Scratch expressions for these are shown below:

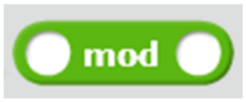




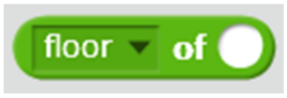
Arithmetic operators:

Scratch comes with several arithmetic operators other than the basic ones (discussed above). They are listed at the bottom of the "operators" tab of Scratch.

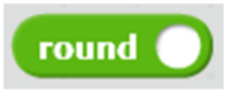
We will discuss a few of them here, especially those that we have used in our projects in this book.



Gives remainder of a division.
For example: 10 mod 5 = 0, 12 mod 7 = 5.



Gives the integer portion of a decimal number.
For example: Floor of 10.3 = 10, Floor of 7.8 = 7



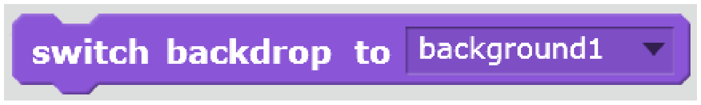
Rounds the given decimal to the nearest integer.
For example: Round 6.6 = 7, Round 9.3 = 9.

Backdrops – multiple:

The Scratch stage can have multiple backdrops. There are a number of ways to create a new backdrop:

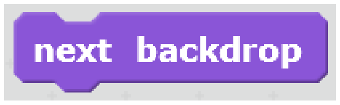
- Choose from the Scratch library
- Paint using the paint editor
- Upload an image from your computer
- Take a photo using your camera

Once you have multiple backdrops, either the stage or any of the sprites can change the backdrop any time using one of the following commands (listed under the "Looks" tab):



switch backdrop to background1

The stage can also use the following command to change backdrops:

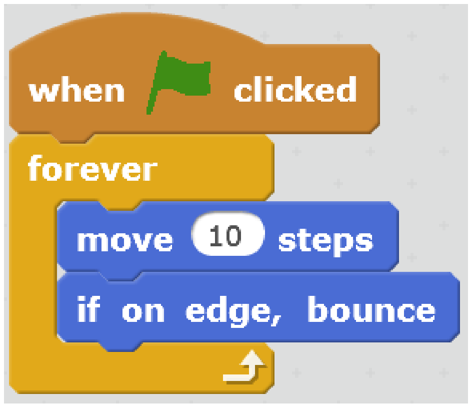


next backdrop

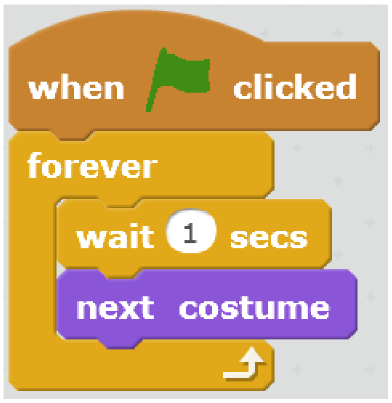
Concurrency - running scripts in parallel

Concurrency basically means doing many actions at the same time. People like you and me are doing multiple things at the same time, all the time! For example, when we take a walk, we listen to music or talk on the phone, watch for road signs and so on. All these actions happen concurrently.

Here is a simple example:



```
when [flag] clicked
forever
    move (10) steps
    if on edge, bounce
```



```
when [flag] clicked
forever
    wait (1) secs
    next costume
```

We have two scripts for the same sprite. The first script makes the sprite move around the screen. And the second script changes its costume every second.

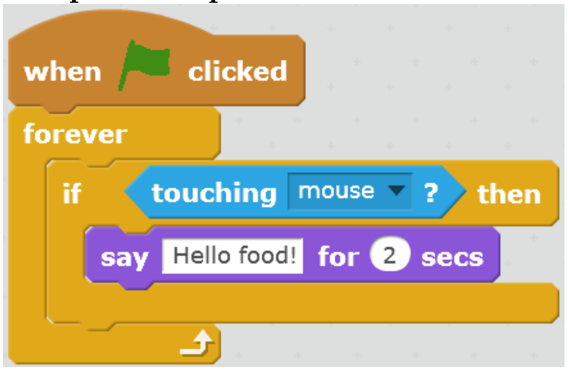
Now, because both the scripts start with the same signal, which is, “Green Flag clicked”, the sprite will appear as if it is performing both these actions – moving and changing costumes – at the same time, that is, concurrently.

Concurrency - race condition

Race condition is the result of a slight misunderstanding about how concurrency works. The scripts that run in parallel don't really run "simultaneously"; the CPU runs a little bit of each script at a time and runs them one after the other. Because this "little bit" is a really tiny portion of the task and because the CPU runs really fast, we get the impression that things are happening simultaneously.

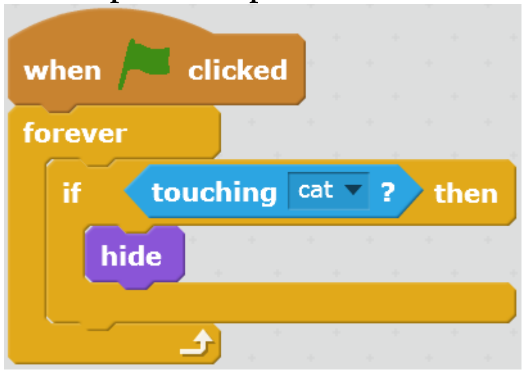
One good example of race condition in Scratch is when two sprites try to sense touch with each other. In the program below, when the cat touches the mouse, the mouse disappears (it is eaten!) and the cat grunts with satisfaction!

Cat sprite's script:



```
when clicked
forever
    if touching mouse ? then
        say Hello food! for 2 secs
```

Mouse sprite's script:



```
when clicked
forever
    if touching cat ? then
        hide
```

If the cat's script senses the touch first, there is a good chance that mouse will also sense it. But, if the mouse's script senses the touch first, it will hide immediately and it is possible that by the time the cat's script checks the condition it will likely miss the touch because the mouse is now invisible.

To avoid such race conditions, it is best to have only one sprite do the sensing. It could then send a message to the other sprite.

Conditionals (IF)

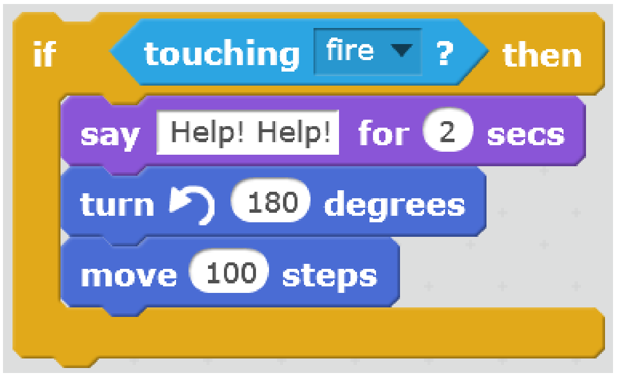
In Computer Science there is a special type of question called the TRUE-FALSE question. In this type the answer can only be either TRUE or FALSE. For example, "Is it raining?" Or, "Are you going home?" Or, "Is 25 a square of 5?" The answer to all these questions is either TRUE or FALSE. There is no other possible answer.

These questions are known as CONDITIONS in programming.

Scratch provides many such conditional questions and they are shown as diamond blocks. If you look under the SENSING tab, you will see several conditions. TOUCHING is a condition which basically is the question "Am I touching so and so?" There is another diamond block called TOUCHING COLOR which is the question "Am I touching this color?" Under OPERATORS there are diamond blocks that compare numbers.

In real life, we use the TRUE-FALSE questions to take some action. For example, if the answer to the question "Is it raining?" is TRUE (or YES), we might decide to take the umbrella to school.

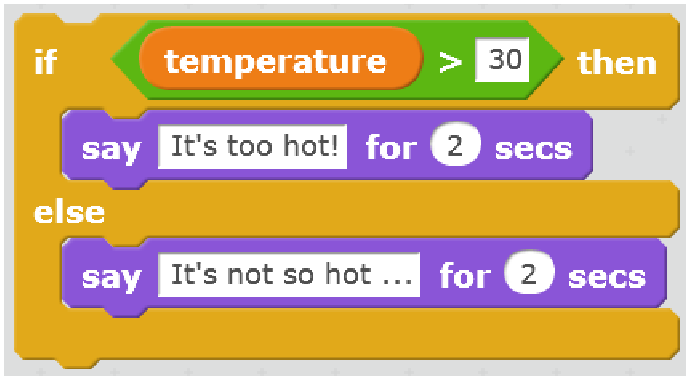
Similarly, in Scratch we have a command called IF that we can use if a condition is TRUE. In the example shown below, if the sprite is TOUCHING a sprite called FIRE, the sprite will say "Help! Help!" and then back off by 100 steps.



```
if touching fire ? then
    say Help! Help! for 2 secs
    turn ↺ 180 degrees
    move 100 steps
```

Conditionals (IF-Else)

This is a variation of the IF command described above. See this example:



```
if temperature > 30 then
    say It's too hot! for 2 secs
else
    say It's not so hot ... for 2 secs
```

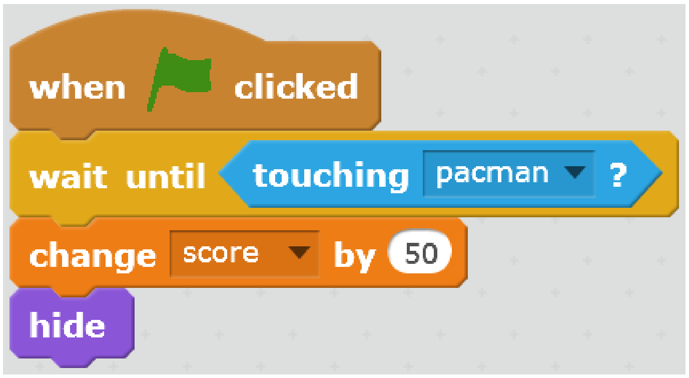
The condition in the above command is: “Is the temperature greater than 30?” If the answer is YES, the sprite says, “It’s too hot!” If the answer is NO, the sprite says, “It’s not so hot …”

So, the IF-Else command offers a fork in the process of decision-making.

Conditionals (Wait until)

Conditions, i.e. questions that only require a binary answer (yes/no, true/false), are used by another command in Scratch as shown in the example below.

Sometimes in our programs, we want a sprite to just wait for some condition to become true. For example, here is a program in which a "prize" sprite is just sitting to be eaten by the *pacman* sprite. When *pacman* touches the prize, it is supposed to disappear.

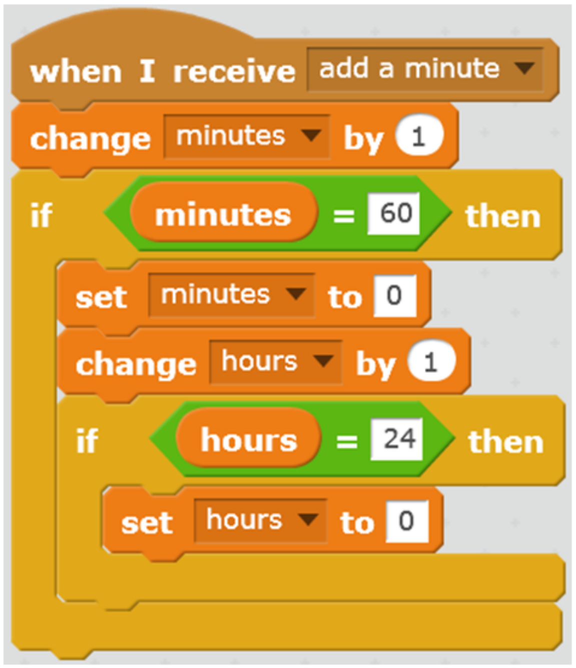


```
when flag clicked
wait until <touching pacman ?>
change score by 50
hide
```

WAIT UNTIL simply waits as long as the CONDITION is false. As soon as the condition becomes true, it stops waiting and the script moves to the next command.

Conditionals (nested IF)

The IF (and the IF-Else) command can be used in many variations. One variation called "nested IF" is shown below. Nested IF means having one IF command inside another.



```
when I receive [add a minute ▼]
change [minutes ▼] by (1)
if <(minutes) = [60]> then
    set [minutes ▼] to [0]
    change [hours ▼] by (1)
    if <(hours) = [24]> then
        set [hours ▼] to [0]
```

This script is for a digital clock.
It adds a minute to the time.

If the total minutes become 60, that's counted as an hour and counting of minutes starts again from 0.

If the total hours become 24, counting starts again from 0.

Costumes

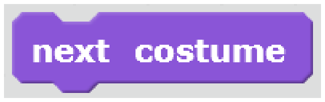
Animation, as you know, is an illusion of action or motion. We can make pictures of people, animals, and even things appear alive by making them do things.

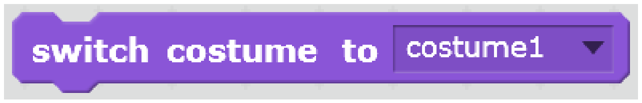
Motion is one way to create animation. But, there is one shortcoming in that. Motion commands make the whole sprite move. You cannot move its parts. There is no change of expression or anything like that.

Animation is basically a trick played on our eyes. When we see a succession of slightly changing pictures rapidly one after the other, for example, the pictures of a hand moving up, our eyes think that we actually saw a hand moving up!

In Scratch this series of pictures is called costumes. Every sprite can line up its costumes in the "Costumes" tab. Several Scratch sprites come with a few costumes of their own. In addition, you can draw costumes using the Paint editor, or you can simply import images from outside. You can also use a camera if your computer has one.

We use the following commands to actually use these costumes to create animation.



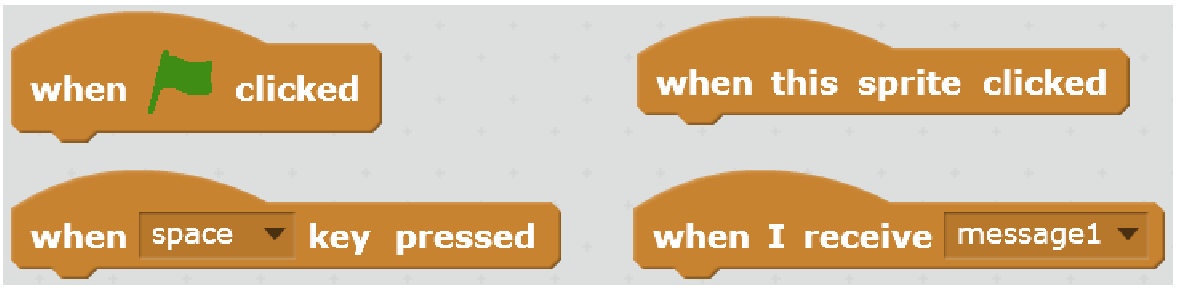


The "next costume" command will make the sprite change its appearance and look like the next costume in its list of costumes. When it reaches the end of the list it goes back to the first costume in the list.

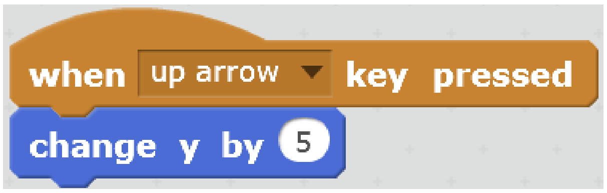
The command "switch to costume" allows the sprite to change to any costume in the list. This is handy when you don't have an orderly series of costumes, but a set of costumes that your sprite wants to use in no particular order.

Events

Events are outside happenings that create some sort of a signal. For example, the "ringing of a phone" is an event that tells us that someone is calling. "Traffic light turning red" is an event that signals that cars need to stop. Real world is full of events. Following are some of the commonly used events in Scratch:



Every sprite in your program will have scripts to respond to the events that that sprite is interested in. For example, the ball sprite in a game responds to the "when up arrow key pressed" event by moving up a little as shown below:



```
when up arrow key pressed
change y by 5
```

Events - coordinating multiple user events

This is not really a basic CS concept but a specialized idea. Sometimes you need to respond to a "combination" of events rather than to a single event. Here is an example:

In our "Tower of Hanoi" program, the user needs to move discs from one stack to another. For this, he/she needs to specify the *source* and *destination* stacks. And let's say we want to use the "when sprite clicked" event to point out the stacks. Obviously a single click won't be enough; we will need to coordinate multiple events (two in this case).

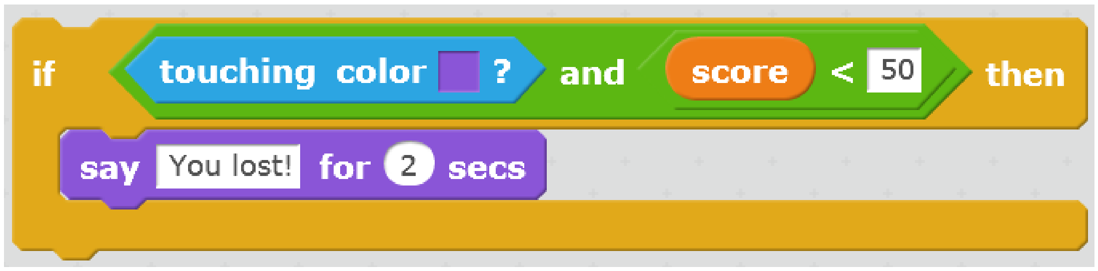
When the user clicks on a rod, the rod has no way to know whether it is the "source" or the "destination". It all depends on which rod was clicked first.

We can achieve this coordination by guiding the user through the process. We will prompt the user that he/she should click on a rod to pick the *source*, and then click on another rod to pick the *destination*.

We will use a separate sprite called "prompt" that will display these prompts by changing costumes. First, when a disc move is requested (by pressing SPACE BAR), this sprite will show the prompt "Click on the source rod". As soon as the user clicks on a rod, that rod's name will be recorded in a variable. The "prompt" sprite will then change its costume to say "Click on the destination rod". As soon as the user clicks on a second rod, that rod's name will be recorded in another variable. The "prompt" sprite will then change to a blank costume since the move is complete. It will also send out a broadcast to signal that the event combination is over.

Logic operators (AND, OR, NOT)

Conditions, i.e. Boolean questions that return yes/no or true/false, can be combined using these logic operators. For example, in the script below, the sprite will say "You lost!" only if it is touching the purple color AND if the score is less than 50.



```
if <<touching color [ ]?> and <(score) < [50]>> then
    say [You lost!] for (2) secs
```

The best way to understand how the logic operators work is to construct truth tables.

The AND operator

Condition 1	Condition 2	Combined effect (output)
False	False	False
False	True	False
True	False	False
True	True	True

As you can see, the output is true only when both conditions are true.

The OR operator

Condition 1	Condition 2	Combined effect (output)
False	False	False
False	True	True
True	False	True
True	True	True

As you can see, the output is true when at least one condition is true.

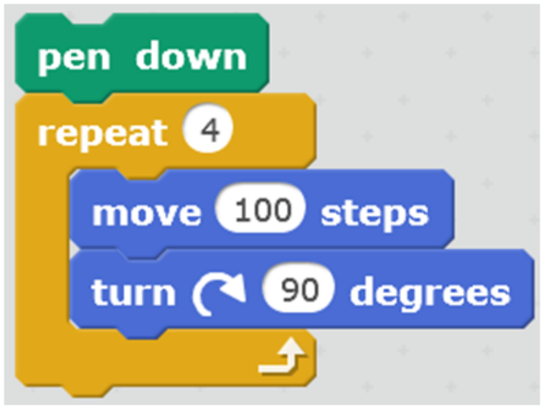
The NOT operator

Condition	Combined effect (output)
False	True
True	False

As you can see, the output is the exact opposite of the input.

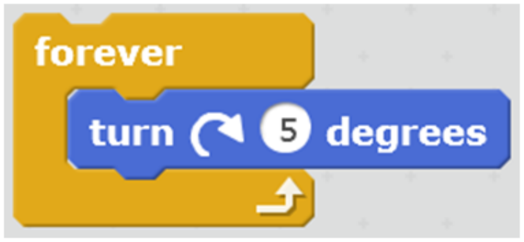
Looping - simple (repeat, forever)

Looping (also known as *iteration*) is the repetition of a sequence of commands. The "repeat" and "forever" commands in Scratch allow simple looping:



```
pen down
repeat 4
    move 100 steps
    turn ↻ 90 degrees
```

This will draw a square of size 100.

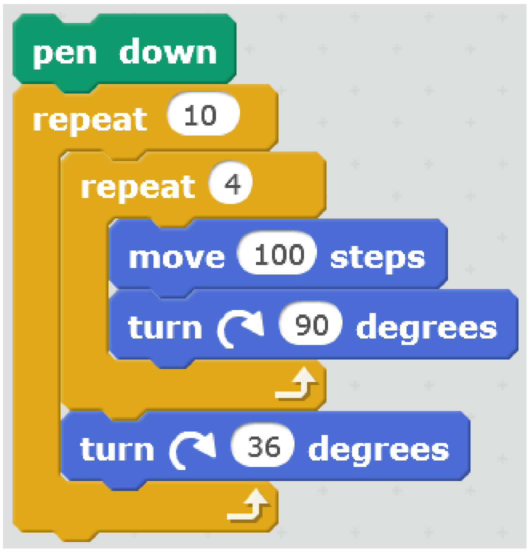


```
forever
    turn ↻ 5 degrees
```

The sprite will spin around itself forever.

Looping - nested

Nesting means having one loop inside another. See the example below:

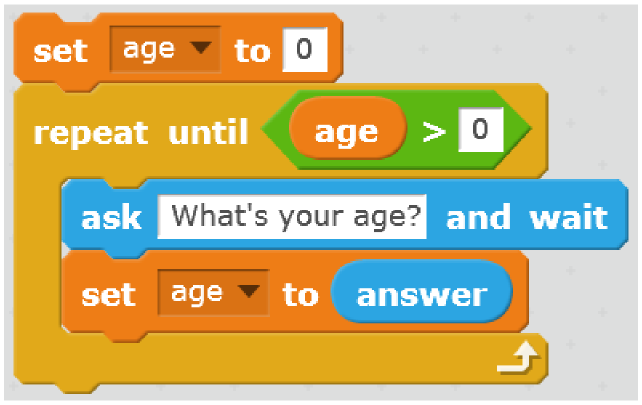


```
pen down
repeat 10
    repeat 4
        move 100 steps
        turn ↻ 90 degrees
    turn ↻ 36 degrees
```

The inside repeat loop draws a square, and the outside loop calls this square loop and then turns the sprite slightly. The result is a nice-looking flower of squares. Thus, nesting of loops can open up interesting opportunities.

Looping - conditional (repeat until)

In the simple looping covered above (Repeat and Forever), the repeat count is fixed beforehand. But, many times we may want to terminate the repetition based on a condition. For example, see the script below:



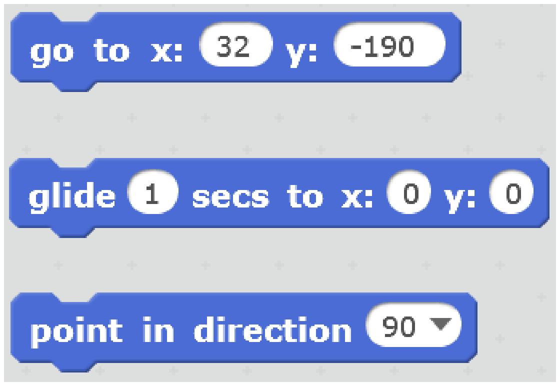
```
set age to 0
repeat until age > 0
    ask What's your age? and wait
    set age to answer
```

In this script, we want the user to enter his/her age. But, what if he/she enters a negative number? In order to ensure the age is a positive number, we need to continue asking the same question to the user until the user enters it correctly. "Repeat until" does exactly that: it asks the same question until age > 0.

Motion – absolute

Absolute motion refers to motion whose result *does not* depend on your current position and direction. For example, "Going to Washington DC" is absolute motion because you will end up in Washington DC no matter where you currently are.

The following example Scratch commands describe absolute motion because the resulting position or direction *does not* depend on the sprite's previous position or direction.

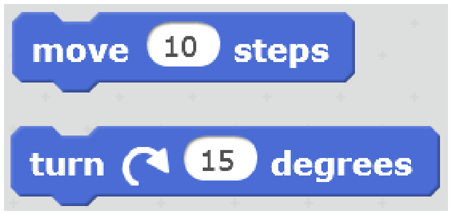


```
go to x: 32 y: -190

glide 1 secs to x: 0 y: 0

point in direction 90
```

Motion – relative

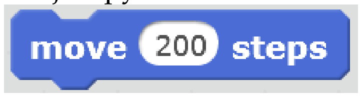
In contrast to absolute motion, relative motion depends on your current position and direction. For example, "Turn right" is relative motion because the resulting direction depends on your current direction.

The following example Scratch commands describe relative motion because the resulting position or direction *does* depend on the sprite's previous position or direction.



```
move 10 steps
turn ↻ 15 degrees
```

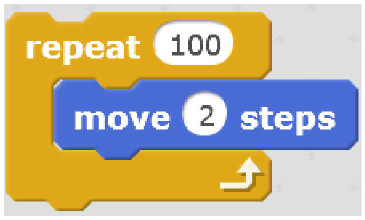
Motion - smooth using repeat

The Move and Turn commands are abrupt or jumpy in the sense that the sprite jumps to a new position. The transition is not smooth. So, 

```
move (200) steps
```

would make the sprite jump by 200 steps. But, what if we want the motion to appear smooth?

Looping comes to help! If you break down the total distance into multiple smaller steps and use Repeat, the motion appears smoother because each move takes a very short finite time.



```
repeat (100)
    move (2) steps
```

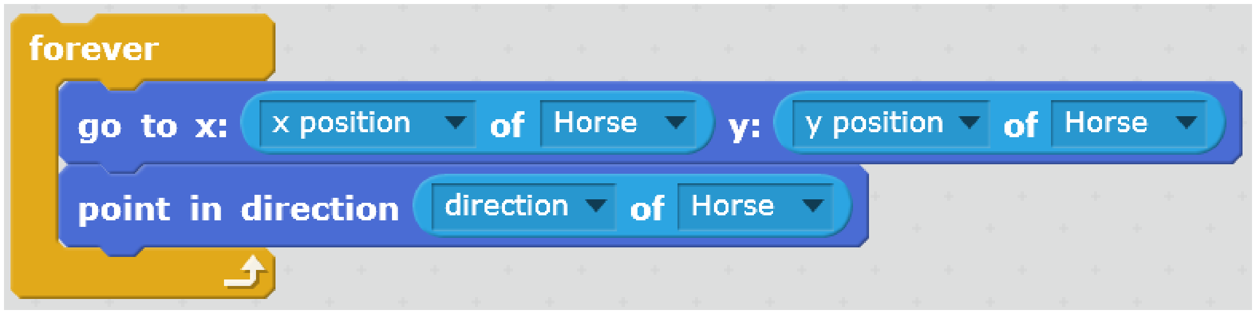
The above script still moves the sprite the same distance – 200 – but it is split into 100 jumps of 2 steps each. Since each move command takes a short time, 100 jumps would add up that time to make the motion look smooth. The speed of the motion would depend on the size of each move.

This same idea can be applied in other situations, such as, turning.

Motion - piggyback another object

There are occasions when we want one sprite to simply move along with another sprite. The two sprites cannot be combined into a single sprite because perhaps they cannot move together all the time. Think of, for example, an animation containing a horse-rider, who would ride the horse, but also be separate from the horse some times.

Creating such "piggyback" motion is quite straightforward, as shown below.



```
forever
    go to x: (x position of Horse) y: (y position of Horse)
    point in direction (direction of Horse)
```

The sprite that wants to piggyback (the rider in this case) continuously goes to the other sprite's (horse's) location and uses its direction.

Motion - direction and bouncing

Generally speaking, whenever an object bounces off a flat surface, its incoming angle (with the surface) is equal to its outgoing angle.

Let's see how this works in Scratch, where angles of sprites are given by the "direction" property. A sprite's direction is measured w.r.t. the North direction. So, if the sprite is facing north *direction* equals 0. If it's facing east, *direction* equals 90, and so on.

If you observe how the *direction* property is affected by bouncing, you will notice the following:

1. When a sprite bounces off the left or right edge (of the screen), its direction changes only in sign. So, 30 becomes -30, -110 becomes 110, and so on. (So, we can simply multiply *direction* by -1.)
2. When a sprite bounces off the top or bottom edge, and if its initial *direction* is A, after bouncing it becomes 180-A.

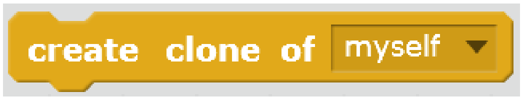
So, depending on what type of surface (horizontal or vertical) the sprite is bouncing off, the calculation of *direction* would be different.

OOP - creating instances using clones

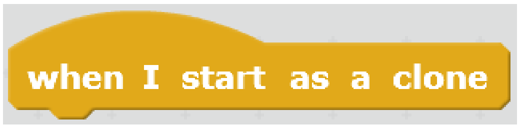
OOP stands for "object oriented programming". In OOP, there is a concept of a class, which defines the characteristics (i.e. its properties and its actions) of a collection of objects, and an object is an instance of that class. Using this idea, a program can have multiple instances of a class.

From Scratch's perspective, every sprite can be viewed as a class which contains its own data (variables) and methods (scripts). And the idea of creating instances of a class is implemented using a feature called "clones". So, clones are basically identical copies of a sprite which exist only at run-time, that is, they are created by the program and they vanish when the program stops running.

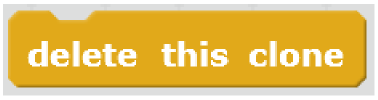
The following command creates a clone of a sprite:



Clones inherit (copy) all scripts of the parent sprite that begin with an event block (except "When Green Flag clicked"). In addition, every clone also runs (only once) scripts that start with this event:



All clones are deleted when you stop the program using the STOP command or by clicking the STOP button. Each clone may also delete itself using this command:

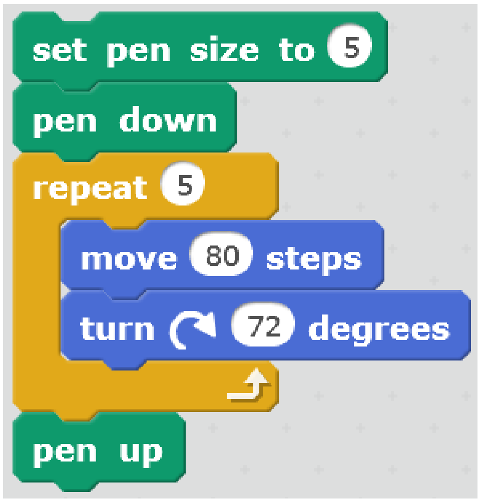


Since a clone is an identical copy of a sprite, it shares all its methods, but has its own copy of all its private variables. Each clone also has its own copy of built-in Scratch properties such as, *xposition* and *yposition*.

Pen commands

Every sprite in Scratch has a pen attached to it (at its center) and is able to draw on the background. The pen commands are listed under the Pen tab and they contain commands to put the pen down (after which the sprite will start drawing wherever it goes), pen up (after which the drawing will stop), set pen size, set pen color, and so on.

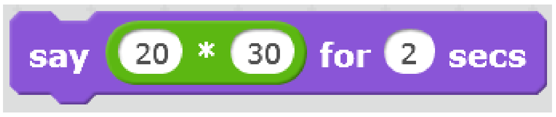
The actual drawing happens when the sprite moves using the motion commands. Using cleverly designed scripts, you can draw practically any type of geometric patterns. For example, the following script draws a thick pentagon:



```
set pen size to 5
pen down
repeat 5
    move 80 steps
    turn ↻ 72 degrees
pen up
```

Procedures

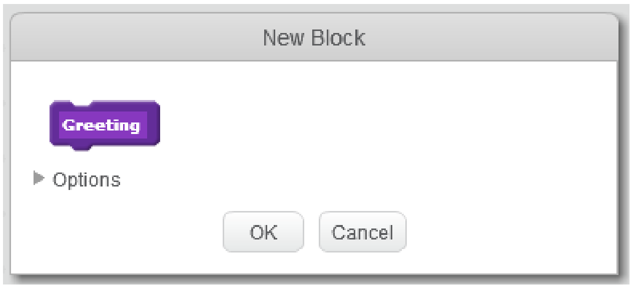
A Scratch *instruction* consists of a command block that we drag and drop in the script area, and which is then run by Scratch when we click on it. For example, the following instruction evaluates the expression 20*30 and shows 600 on the screen.



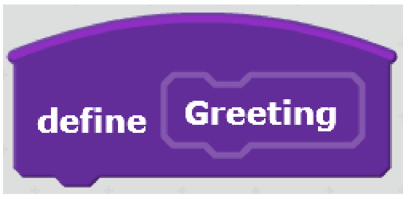
A Scratch instruction always carries out a specific, well-defined, and repeatable task. If you look carefully, you will notice that this instruction above contains the keyword "SAY" and the symbol "*" (for multiplication). Each of these words/symbols is called a Scratch *procedure*.

A *procedure* is like a recipe of how to do something. For example, MOVE is a procedure that knows how to move a sprite.

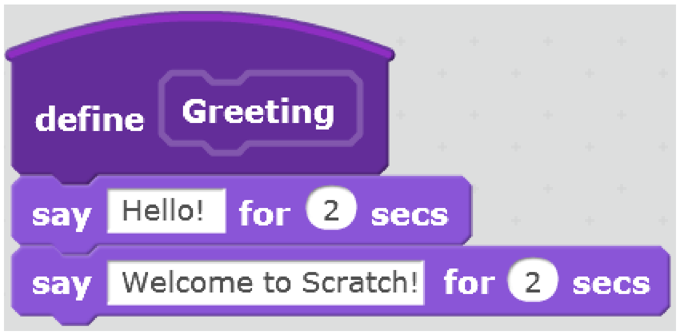
Scratch allows you to define your own procedures. Go to the "More blocks" tab and click on "Make a block". The following picture shows that I am creating a new procedure called "Greeting".



When you click Ok, you see this new block in the script area:

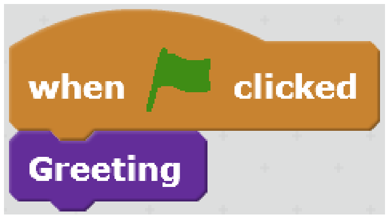


Below this you create a script by attaching existing Scratch command blocks. This is what I created:



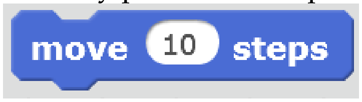
```
define Greeting
say Hello! for 2 secs
say Welcome to Scratch! for 2 secs
```

You now have a new procedure called "Greeting" which you can use in any script. For example:

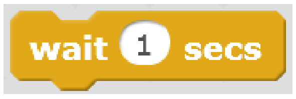


```
when [flag] clicked
Greeting
```

Procedures with inputs

The behavior of many procedures depends on how they are invoked. It depends on the *input* supplied. So, 

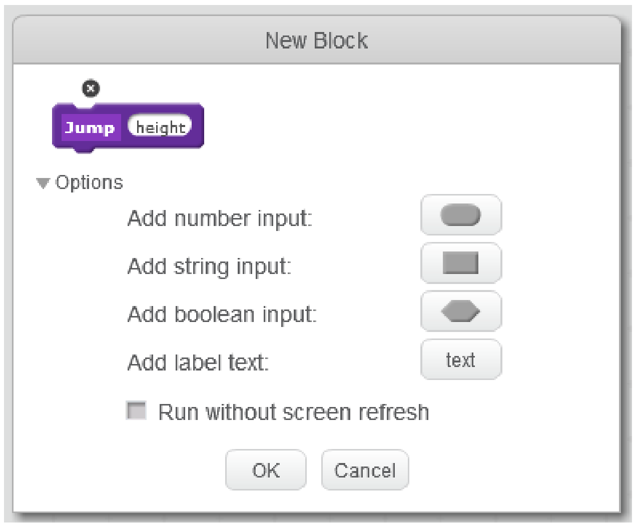
```
move 10 steps
```

can do its job only when you tell it *how many* steps to move. Or, 

```
wait 1 secs
```

command needs to know *how many* seconds to wait.

Scratch allows you to define your own procedures which take input. While creating a new procedure (as described above) if you click on "Options" you will see the following:



New Block

Jump (height)

▼ Options

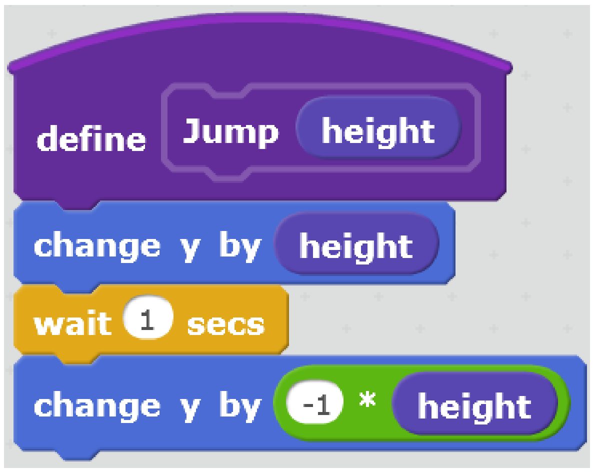
Add number input:	
Add string input:	
Add boolean input:	
Add label text:	text

☐ Run without screen refresh

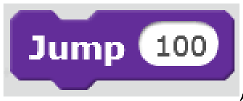
OK Cancel

In this case, I am creating a new procedure called "Jump" which will take a *number* input, which I have labelled as "height". The label has no importance really, other than giving a descriptive name to the input.

After clicking OK and attaching a script, I have a new procedure (with input) as shown below:



```
define Jump (height)
change y by (height)
wait 1 secs
change y by ((-1) * (height))
```

If I use this new procedure in a script, say, as 

, the sprite will jump up and down by 100 steps.

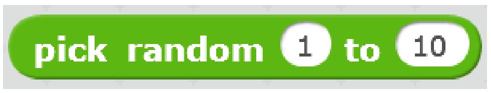
Random numbers

Several things in real life are unpredictable. Here are just a few examples:

```
Pick a ball out of a bag.
Roll a dice.
Toss a coin.
Decide what to wear to a party
```

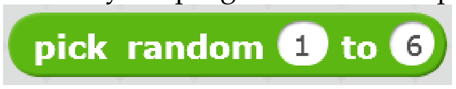
These acts are not completely random; they are random with constraints. The outcome in each of the above is one of a set of possible outcomes. Coin toss involves 2 possible outcomes; retrieving a ball from a bag depends on the number of balls; roll of a dice has 6 possible outcomes, and so on.

Scratch offers a simple way to use randomness in programs through the following operator:



pick random 1 to 10

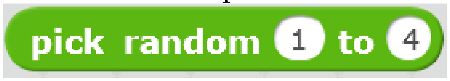
This operator returns a number from 1 to 10 – you can't predict what it will return. You can use any range (e.g. -100 to 100); the range can be in any order (e.g. 100 to -100); and it can even be a decimal range (e.g. 1.5 to 11.5).

The real key is to figure out how to use this simple operator to simulate the random events that you might want to use in your programs. For example, how would you simulate the roll of a dice? Simple: use  pick random 1 to 6 since there are 6 possible outcomes.

Random numbers - mapping to a set of things

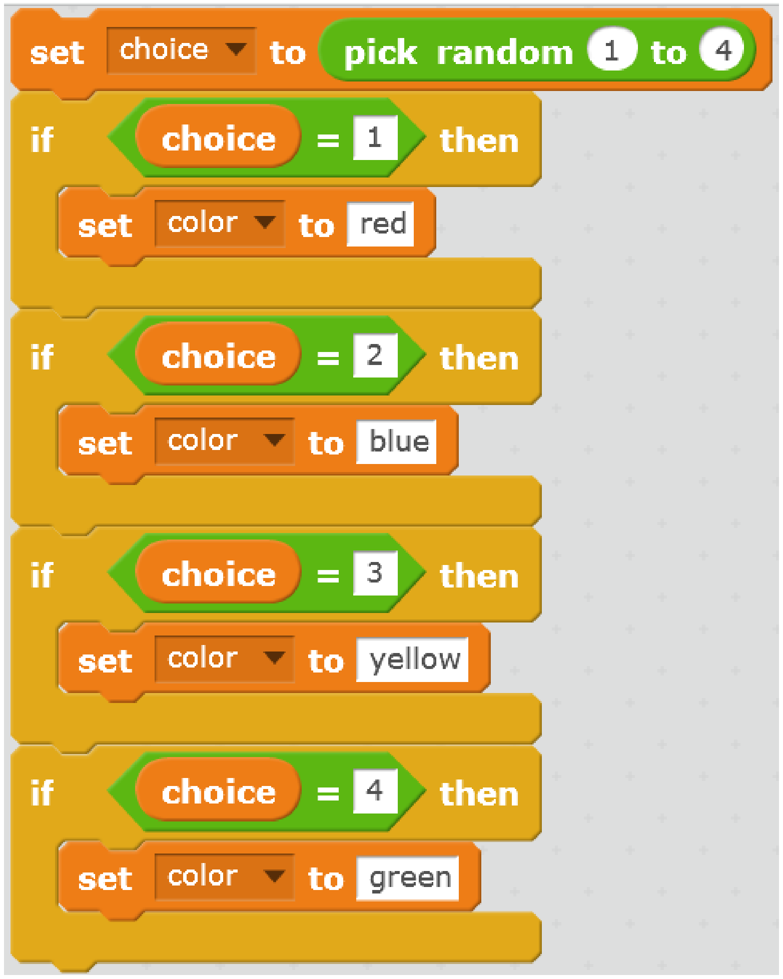
Sometimes you want to pick something from a collection of things, which is not necessarily an ordered list of numbers. For example, you want to pick a color from the set of 4 colors: green, red, blue, and yellow. How would you do that using the random operator?

Well, the first step is to realize that the collection contains 4 items. So we should use



```
pick random 1 to 4
```

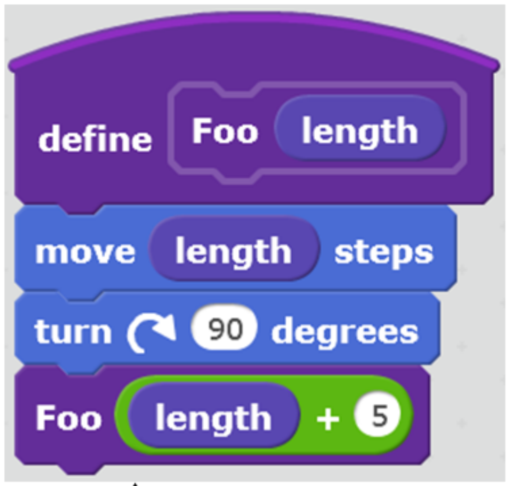
to pick from 4 numbers. Next, we can map each number to a color: so, 1 could map to red, 2 to blue, 3 to yellow, and 4 to green. Then, depending on what *pick random* returned, you would know which color to pick. Here is the script to show the entire scheme of things:



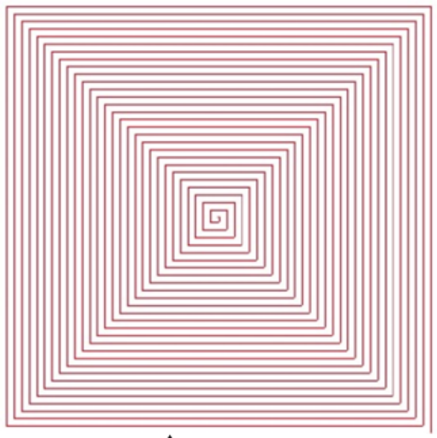
```
set choice to pick random 1 to 4
if choice = 1 then
    set color to red
if choice = 2 then
    set color to blue
if choice = 3 then
    set color to yellow
if choice = 4 then
    set color to green
```

Recursion

The idea of recursion is a property of procedures. Simply stated, a recursive procedure calls itself. For example, the following procedure is recursive:

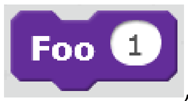


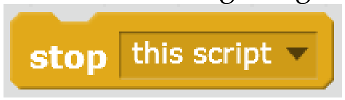
```
define Foo (length)
move (length) steps
turn ↻ (90) degrees
Foo ((length) + (5))
```



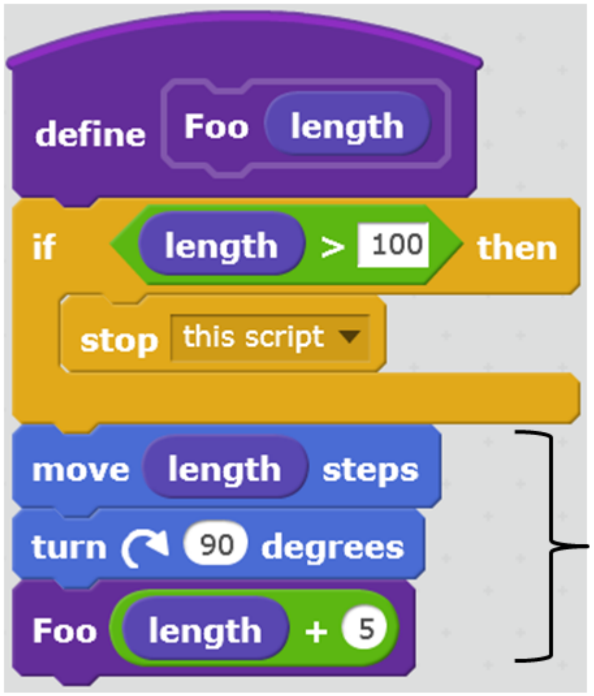
Recursive call to itself.

This is what will be drawn if you call this procedure.

If you call this procedure, say by using  `Foo (1)`, it will draw a rectangular spiral forever; it will never return because it will call itself indefinitely. What you will have is a program that will run forever!

Writing a program that never terminates is interesting but not very convenient. We would like to write recursive programs that do interesting things and terminate (i.e. stop) when their job is done. The Scratch command  `stop [this script ▼]` will stop the currently running procedure and return to the calling procedure. Thus, the infinite recursion will be broken.

Here is a modified recursive procedure that will not run forever:

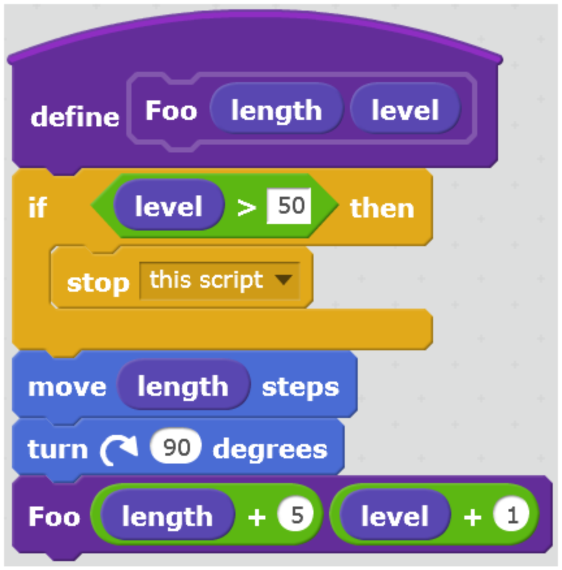


```
define Foo (length)
if <(length) > (100)> then
    stop [this script ▼]
end
move (length) steps
turn ↻ (90) degrees
Foo ((length) + (5))
```

When *length* becomes greater than 100 the procedure stops, thus breaking the recursion.

Otherwise it continues drawing the rectangular spiral.

There is another way to make recursion finite, that is, make it stop after some time. We can simply decide how deep the recursion should go. See the modified Foo procedure below:



```
define Foo (length) (level)
if <(level) > (50)> then
    stop [this script ▼]
end
move (length) steps
turn ↻ (90) degrees
Foo ((length) + (5)) ((level) + (1))
```

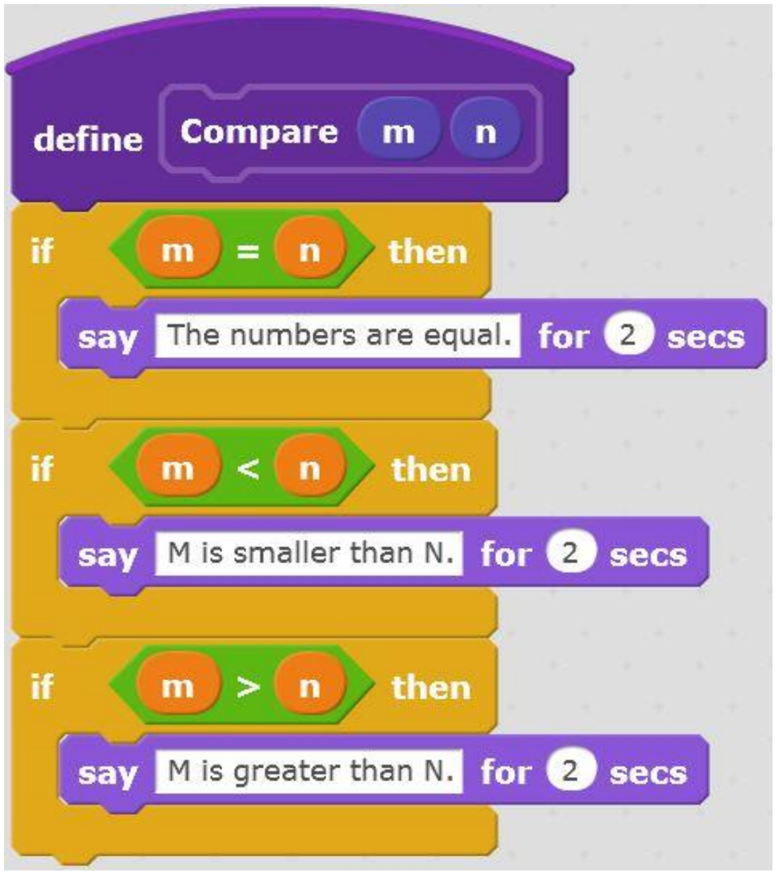
Here, let's assume *level* = 1 when we call Foo. It becomes 2 when Foo is called the 2nd time. It becomes 3 when Foo is called the 3rd time, and so on. "IF" will cause this recursive chain to terminate when *level* becomes 51.

Relational operators (=, <, >)

These operators compare two values and return true or false, depending on whether the comparison succeeded or failed. For example:
20 = 30 would return false
51 > -1 would return true, and so on.

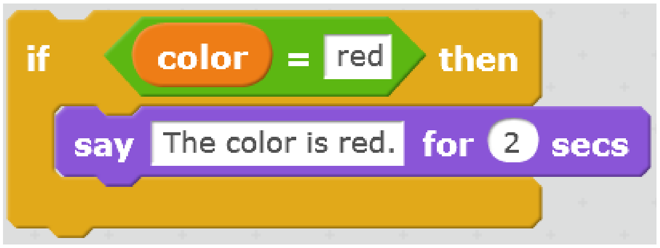
These operators are typically used by conditional statements, such as, IF, Repeat until, etc.

Here is a complete example:



```
define Compare m n
if m = n then
    say The numbers are equal. for 2 secs
if m < n then
    say M is smaller than N. for 2 secs
if m > n then
    say M is greater than N. for 2 secs
```

The "=" operator can also be used to compare strings:



```
if color = red then
    say The color is red. for 2 secs
```

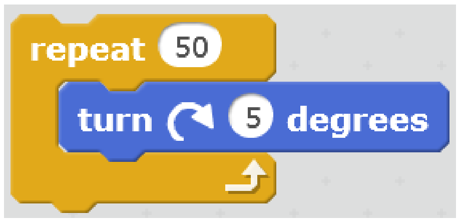
Scratch UI - special features

There are several features in the Scratch UI that are not commonly used. Some of them are listed below:

Turbo mode:

Every motion command contains a very small but finite delay. This delay actually helps in creating smooth animation because we can break down a long jump into multiple smaller jumps and then the series of jumps creates the impression that the sprite is moving smoothly.

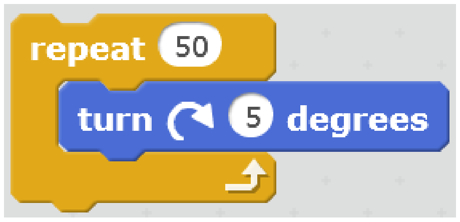
For example, the script below makes the sprite turn slowly around itself.



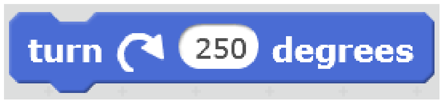
```
repeat 50
    turn ↻ 5 degrees
```

But, this delay can be problematic for some other occasions. For example, if you are drawing a complicated drawing, say a spiral, the number of moves is quite large and so, the cumulative delay can make the drawing really slow. For such purposes, the delay can be (practically) eliminated by using the turbo mode (click "Edit" and then click "Turbo mode"). Suddenly you will notice that your drawing has become much quicker.

With turbo mode on, the following two scripts will be identical in behavior.



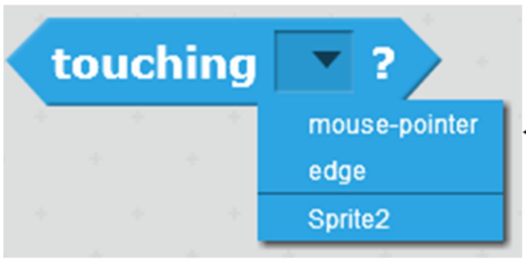
```
repeat 50
    turn ↻ 5 degrees
```



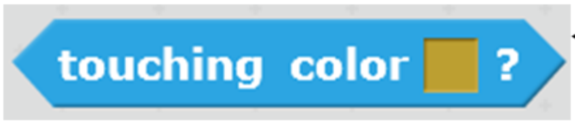
```
turn ↻ 250 degrees
```

Sensing touch

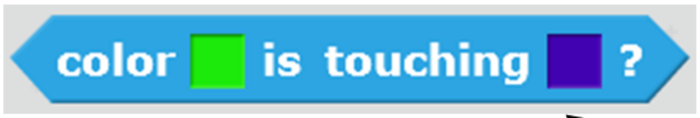
In Scratch, you can check if things are touching each other, and use that information in the conditional statements (such as IF, Wait until, Repeat until, etc.). Here are the sensing conditions:



A sprite can sense if it is touching another sprite, the mouse pointer, or one of the screen edges.

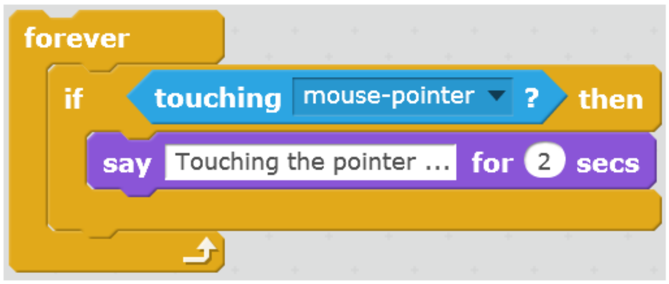


A sprite can sense if it is touching a color (could be part of any sprite or the stage).



A sprite can sense if its own color is touching another color.

An example of how sensing can be used in a conditional statement:



```
forever
    if touching mouse-pointer ? then
        say Touching the pointer ... for 2 secs
```

"Forever" is essential if you want to continuously watch for the touch.

Sequence

Scratch programs tell the computer precisely what to do, step-by-step. To create a program in Scratch, you need to think systematically about the order of steps. Unlike humans, computer programs do not change the order of the commands given to it.

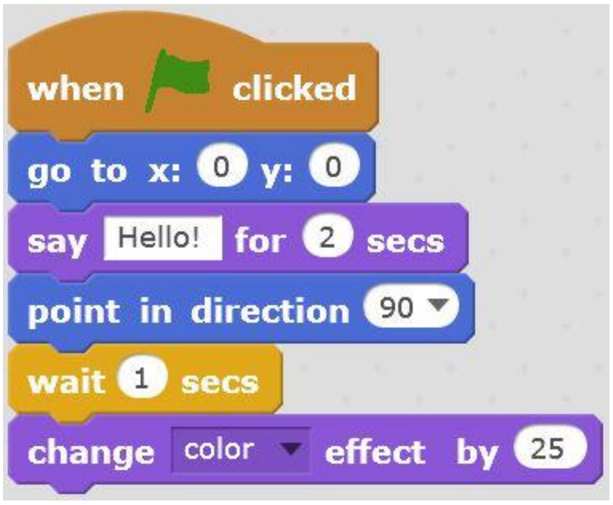
For example, let's say you were given a "To do" list as follows:

1. Get clothes from the laundry
2. Check mail in the post office box
3. Buy grocery

You may not follow this task-list exactly in the given order. You might go to the grocery shop first, or even skip getting the mail.

Computers do not have this freedom. They would, if given the above list, follow it exactly in the given order, one after the other, making sure every item was completed in a satisfactory manner before going to the next item. This concept is called *Sequence* in computer science.

Here is an example of a Scratch script that would run the commands one by one. The color will change at the very end of the program. The sprite will jump to the center of screen first. And so on.



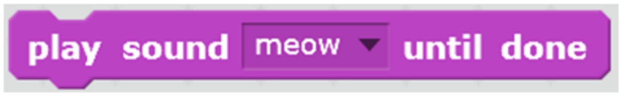
```
when 🏴 clicked
go to x: 0 y: 0
say Hello! for 2 secs
point in direction 90 ▾
wait 1 secs
change color ▾ effect by 25
```

Sounds - playing sounds

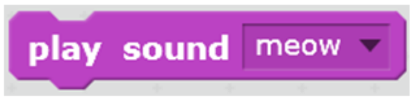
One of the exciting features of Scratch is the ability to play sounds. You might have noticed that every sprite has a tab called "Sounds". Under this tab, you can collect a list of sounds. You can import an existing sound from the Scratch library. You can also import your own sound file in MP3 format. If your computer has a sound recording device, you can record a new sound using the RECORD button.

Once you create a list of sounds for a sprite, you can play these sounds from any script of that sprite.

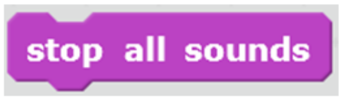
Here are the sound commands that you can use to play sounds.



This command waits until the entire sound file has been played and only then moves to the next command in your script.



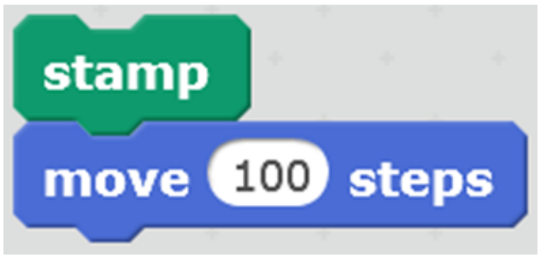
On the other hand, this command starts playing the sound and then immediately moves to the next command in your script. This is useful if you want to play a sound as background score.



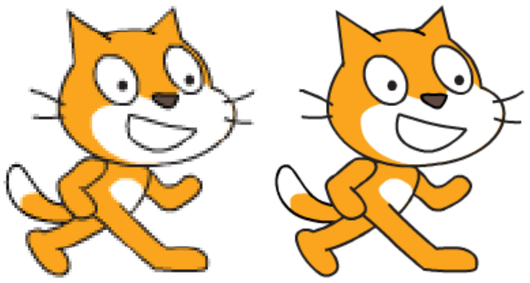
This command stops all sounds that might be playing at that time.

STAMP - creating images

The STAMP command leaves an image of the sprite on the screen. See the example below:

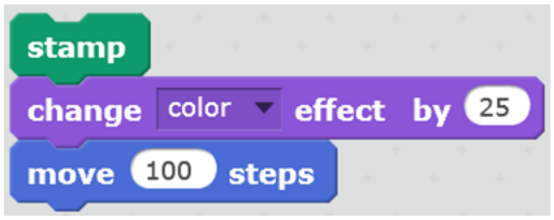


```
stamp
move (100) steps
```

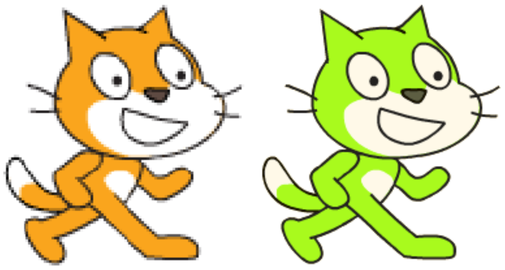


The first one is actually an image, and the second one is the sprite.

The STAMP command has no relation to the pen. So, to change the color of your image you must use the "change color effect" command. See the example below:



```
stamp
change [color ▾] effect by (25)
move (100) steps
```



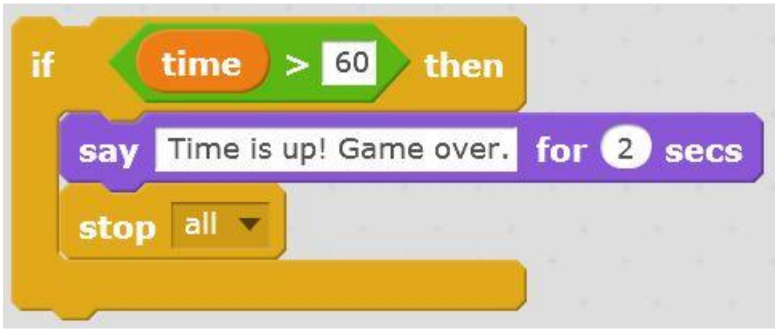
Once again, the first one is an image, and the second one is the sprite.

To erase the images created by STAMP, use the CLEAR command.

Stopping scripts

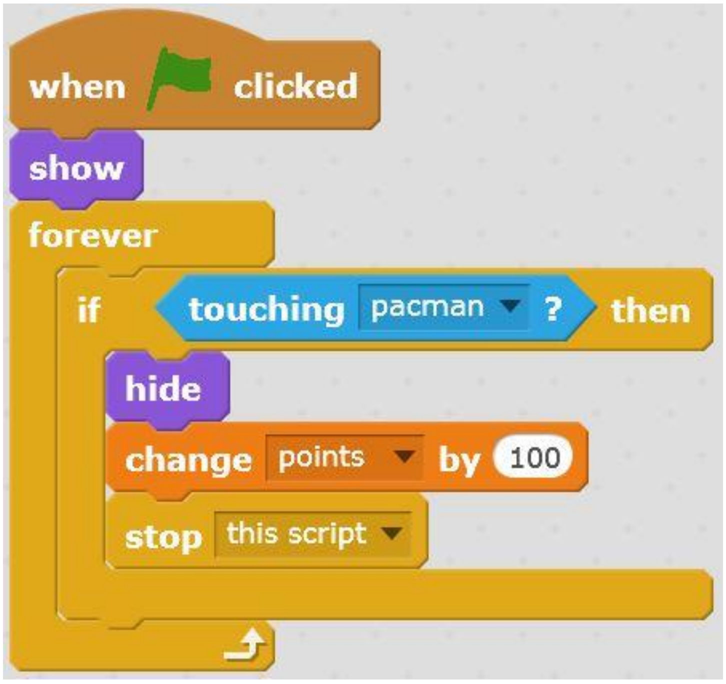
The STOP commands (under the "Control" tab) allow us to stop Scratch scripts in various ways.

The "STOP all" command stops all active scripts (including the current one). For example, the script below checks if the allotted time has been used up, and if so, it stops the game.



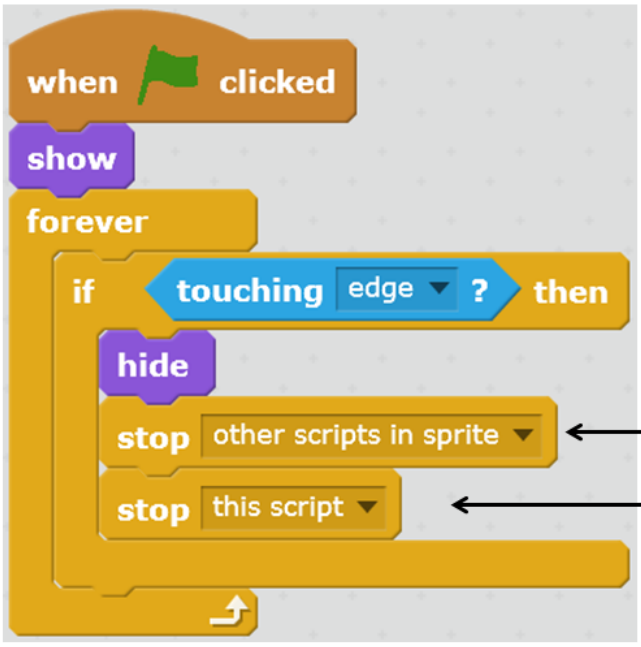
```
if time > 60 then
    say Time is up! Game over. for 2 secs
    stop all
```

The command "STOP this script" stops only the current script. In the following example, a "prize" sprite waits until it touches "pacman". Since it has nothing else to do afterwards, it hides and stops its script.



```
when [flag] clicked
show
forever
    if touching pacman ? then
        hide
        change points by 100
        stop this script
```

Finally, the command "STOP other scripts in sprite" is useful when you want to only stop scripts of the current sprite. If you want to stop all scripts of the current sprite you would need to do as shown in the following hypothetical script:



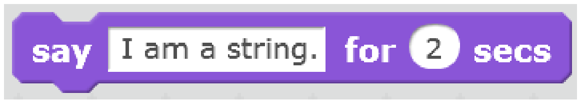
```
when [flag] clicked
show
forever
    if <touching [edge ▾] ?> then
        hide
        stop [other scripts in sprite ▾]
        stop [this script ▾]
```

This script waits until it hits any of the screen edges. Then it is supposed to stop all scripts of this sprite. It does that in 2 steps:

1. Stop other scripts.
2. Stop itself.

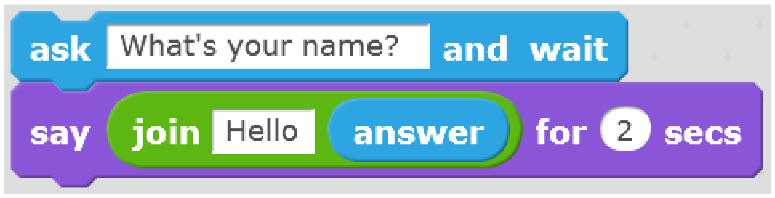
String operations (join, letter, length of)

A string is basically a sequence of alphanumeric letters:



```
say [I am a string.] for (2) secs
```

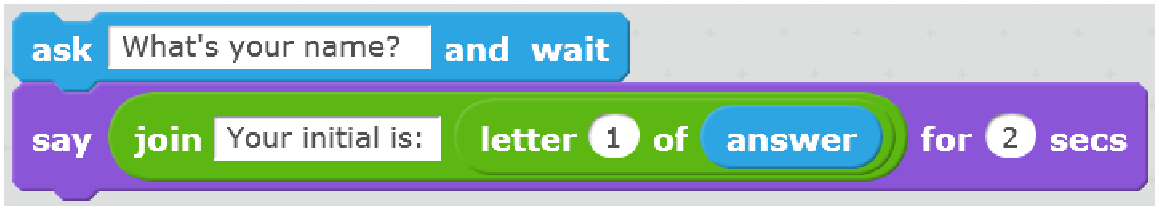
The join operator allows you to concatenate two strings:



```
ask [What's your name?] and wait
say (join [Hello] (answer)) for (2) secs
```

The above script asks you to type your name. If you type “George” the sprite will say “Hello George” on the screen.

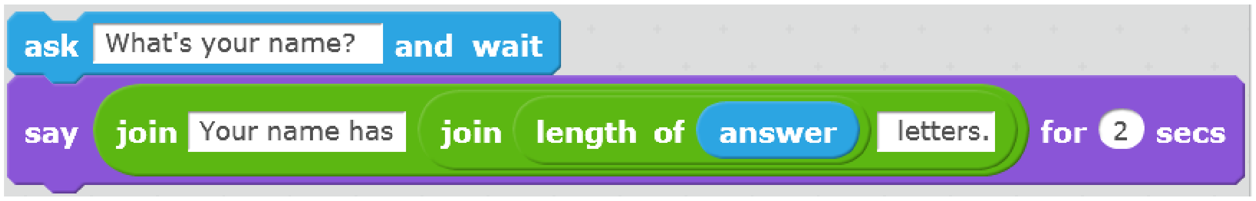
The “letter” operator lets you get an individual letter of the given string:



```
ask [What's your name?] and wait
say (join [Your initial is:] (letter (1) of (answer))) for (2) secs
```

The above script asks you to type your name. If you type “George” the sprite will say “Your initial is: G” on the screen.

The “length” operator tells you how many letters there are in the string:

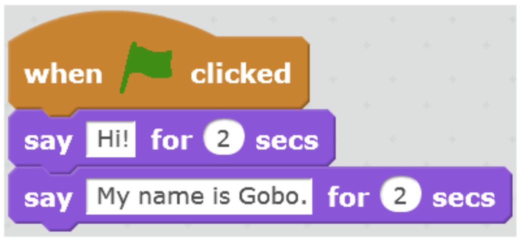


```
ask [What's your name?] and wait
say (join [Your name has] (join (length of (answer)) [letters.])) for (2) secs
```

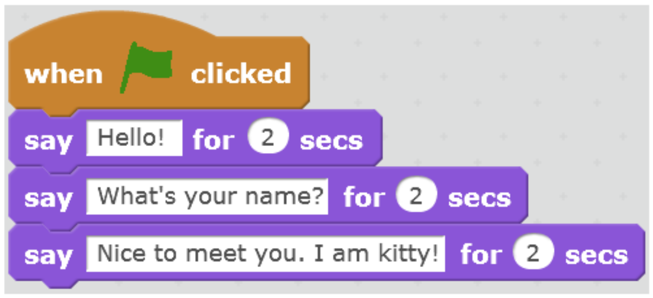
The above script asks you to type your name. If you type “George” the sprite will say “Your name has 6 letters”.

Synchronization using broadcasting

When you write a program that contains multiple sprites that interact with each other in some way, you need to worry about synchronization – which basically means proper coordination of their actions.



```
when [flag] clicked
say Hi! for 2 secs
say My name is Gobo. for 2 secs
```



```
when [flag] clicked
say Hello! for 2 secs
say What's your name? for 2 secs
say Nice to meet you. I am kitty! for 2 secs
```

To understand synchronization, let us look at an example, as shown above. Here, we have two sprites – Gobo and cat – talking with each other. If you run the program what will happen? They both will talk at the same time! There will be no proper sequence or coordination. That's not what you expect, right? To make it a proper dialog, the Gobo should say "Hi" first, then, the cat should reply by saying "Hello", and so on.

So we say that there is no *synchronization* in this program. Do you now see the meaning of synchronization?

You might have heard about the word "Broadcasting" in the context of radios and TV. When you tune into a radio station, you are actually listening to something that was broadcast from a radio station.

Broadcasting basically means sending a message to everyone who cares to listen. So, when someone gives a public speech, he/she is doing broadcasting.

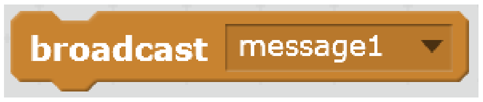
In Scratch broadcasting has a similar meaning. When you use the BROADCAST command, your sprite sends a message that goes to every sprite in your program and even to the stage.

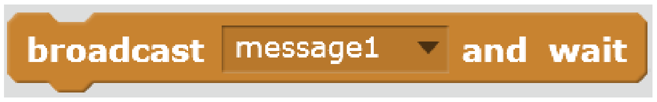
But, just like not everyone is interested in listening to the radio, not every sprite may be interested in hearing the message. So, if a sprite is interested in receiving that message it uses the command WHEN I RECEIVE.

WHEN I RECEIVE is an event and works just like WHEN GREEN FLAG CLICKED. When the message is received the script underneath that event runs.

Every time the exact same message is received the script runs again.

There are two flavors of the BROADCAST command:





In the first flavor, the sending sprite sends the broadcast message and immediately goes to the next command.

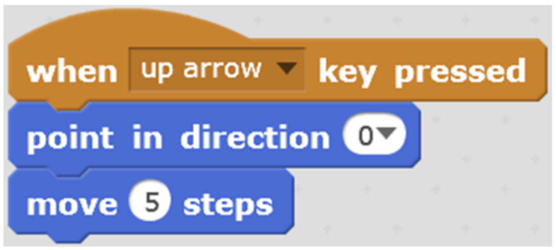
But, in the second flavor, the sending sprite sends the broadcast message and waits – it waits until all listening sprites have received the message and completely run each of their “WHEN I RECEIVE” scripts.

User events (keyboard)

The most interesting feature of Scratch is that it allows the user to interact with programs. Our programs need not be just animations that one has to watch, but, they can be interactive.

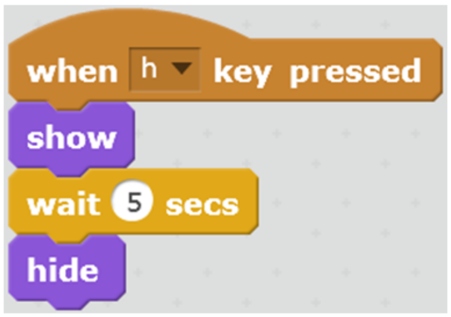
Scratch provides an event block called WHEN KEY PRESSED. It works similar to the event WHEN GREEN FLAG CLICKED. If you write a script under this event, it will run every time the user presses the specified key.

See these examples. In game programs arrow keys are typically used to move sprites. Here, the UP ARROW key runs a script in which the sprite moves 10 steps upwards.



```
when [up arrow] key pressed
point in direction (0)
move (5) steps
```

Move the sprite when an arrow key is pressed.



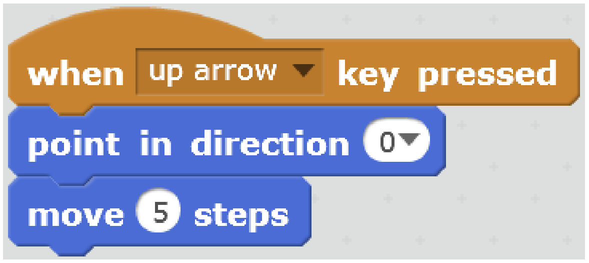
```
when [h] key pressed
show
wait (5) secs
hide
```

Show help screen when “h” is pressed.

In the second example, the h key makes the sprite visible for a short time – this sprite could possibly be a “HELP” screen which pops up for a short time and then goes away.

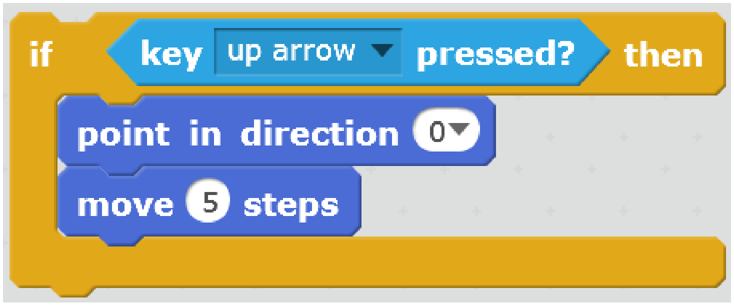
User events (keyboard - polling)

There are two ways to get keyboard input: one is using events (also known as “interrupt driven”), which we have already seen. The following is an event that signals a key-press and immediately runs the script. The command block “when -- key pressed” is called an “event handler”.



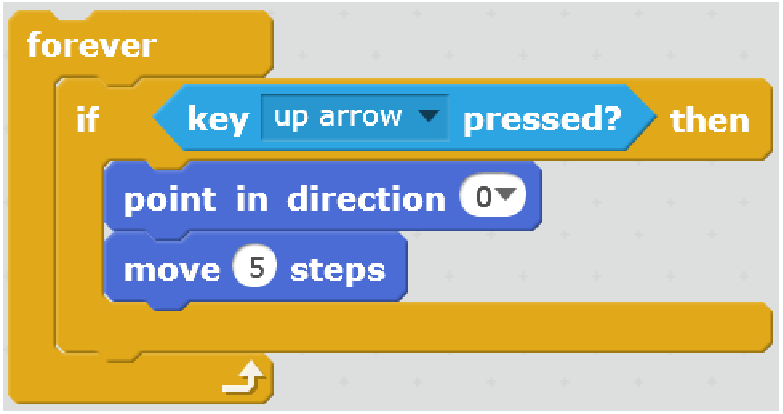
```
when up arrow key pressed
point in direction 0
move 5 steps
```

The other way is called “polling” in which the program actively checks if a particular key is pressed.



```
if key up arrow pressed? then
    point in direction 0
    move 5 steps
```

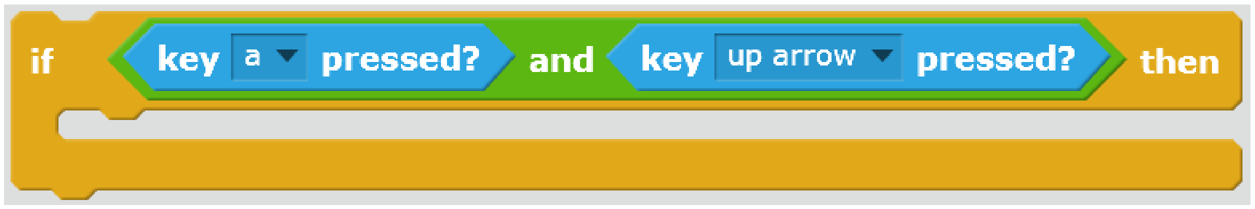
If you want to check this continuously you must use a forever loop:



```
forever
    if key up arrow pressed? then
        point in direction 0
        move 5 steps
```

It is possible in the “polling” method that some key-presses might be missed if their timing does not match with when the IF command checks for the key-press. Also, polling (in a Forever loop) runs continuously taking CPU time, whereas event handlers do not take CPU time, because they sleep and are “woken up” when the event happens. But the advantage of polling is that the program has full control on when to handle keyboard signals. When the program stops, polling also stops, whereas event handlers would run even if the program has stopped.

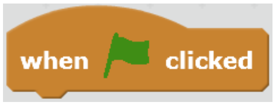
Another interesting advantage of polling is that you can combine multiple events as shown below:

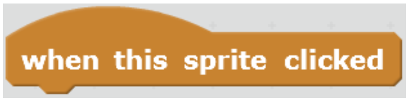


```
if <<key [a v] pressed?> and <key [up arrow v] pressed?>> then
end
```

User events (mouse)

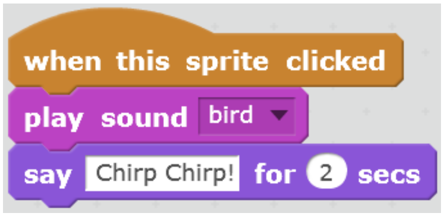
Here are the mouse event blocks that Scratch provides. You are already quite familiar with the event WHEN GREEN FLAG CLICKED.





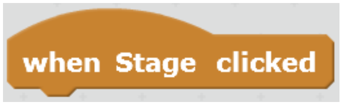
The other event is called WHEN THIS SPRITE CLICKED which invokes a script when you click on that particular sprite. Imagine a button sprite, for example, that you can activate through such a script.

In the script shown below, if you click on the bird sprite, the bird makes a "bird" sound and says "Chirp chirp".

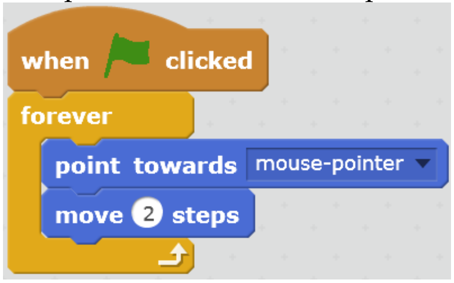


```
when this sprite clicked
play sound bird
say Chirp Chirp! for 2 secs
```

There is a similar event for the stage also:



Another interesting thing you can do in Scratch is that you can make a sprite follow the mouse pointer. The script shown below explains how this can be done. You make the sprite point towards the pointer and then take a small step. If this is done continuously, the effect is to make the sprite follow the mouse pointer.

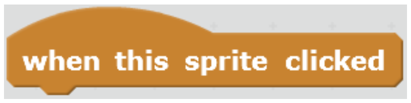


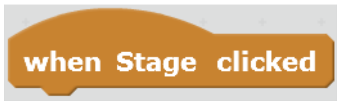
```
when clicked
forever
    point towards mouse-pointer
    move 2 steps
```

In fact, the same idea can be used to make one sprite follow another sprite.

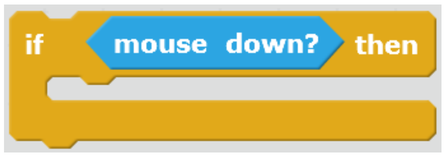
User events (mouse - polling)

Similar to keyboard interaction, mouse signals are handled in two ways: one is called events (or "interrupt driven") as shown below:

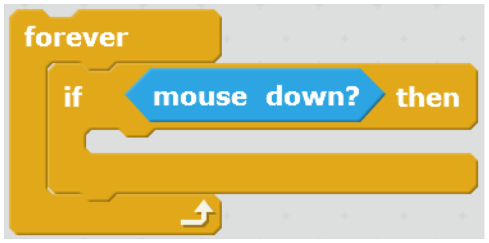




The other is called "polling" as shown below:



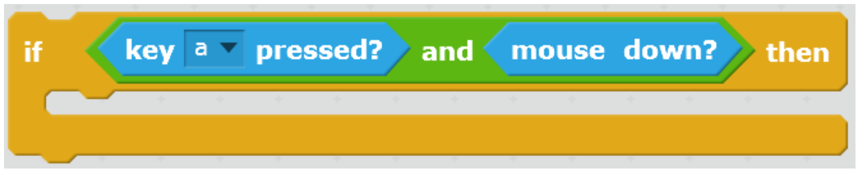
If you want to check this continuously you must use a forever loop:



Note that the condition is "mouse down" and not "mouse click".

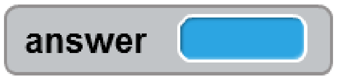
It is possible in the "polling" method that some mouse-downs might be missed if their timing does not match with when the IF command checks for the mouse-down. Also, polling (in a Forever loop) runs continuously taking CPU time, whereas event handlers do not take CPU time, because they sleep and are "woken up" when the event occurs. But the advantage of polling is that the program has full control on when to handle mouse signals. When the program stops, polling also stops, whereas event handlers would run even if the program has stopped.

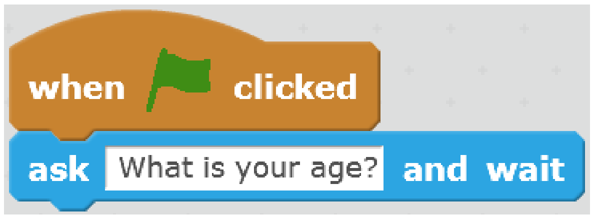
Another interesting advantage of polling is that you can combine multiple events as shown below:



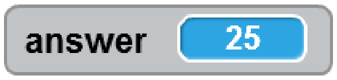
User input (ASK)

Sometimes you may want to ask the user to provide some textual information. The ASK command presents a text window and waits. The user can type his/her reply in this text window and press ENTER. Whatever has been typed is then saved in the "ANSWER" variable. See the script below in which the user types "25" in the text window.

Before running the script: 



```
when [flag] clicked
ask [What is your age?] and wait
```

After running the script: 

User input (buttons)

It is common to have push-buttons (or rather click-buttons) to allow users to interact with programs. See examples below:

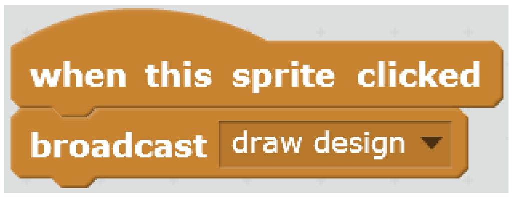
A click-button has the following properties:

- It is usually rectangular, oval, or circular.
- It has a label that describes what action it initiates.
- When you click on it, the specified action is performed.

To implement a click-button:

- Get a button sprite (you can use the built-in sprites or draw your own)
- Label it appropriately
- Use a "When sprite clicked" event to make the button active

Here is an example script for a "Draw" button:



```
when this sprite clicked
broadcast draw design ▼
```

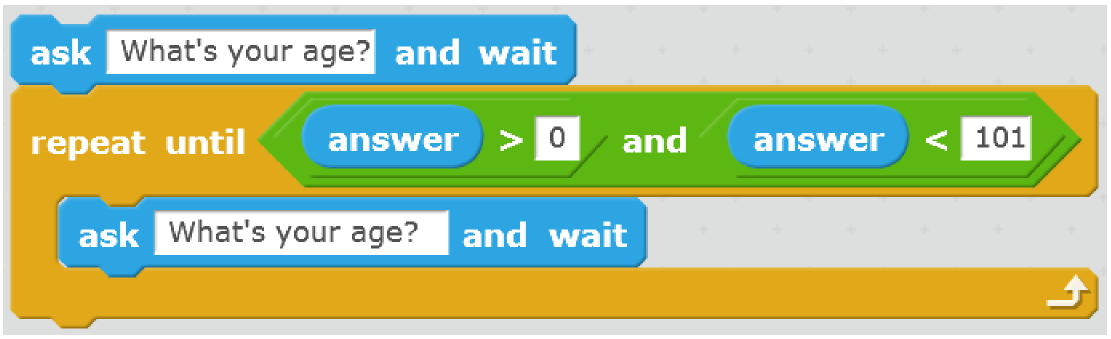
When clicked, the button sprite sends a broadcast message to let everyone know that it was clicked. The actual drawing would be drawn by some other sprite when it receives this message.

User input validation

The ASK command allows the user to enter input, which could be words or numbers. For example, you might ask the user to enter his/her height. It is essential to ensure that the user enters a valid input, in this case, a valid height. Something like -50 or "abc" would not be a valid height.

The program must have a way to validate the user input, and if it is not valid, go back and ask again. The following example shows how this can be done:

Let's say we ask the use for his/her age. We assume that the age cannot be less than 1 or more than 100.



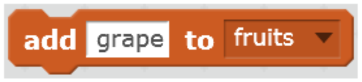
```
ask What's your age? and wait
repeat until < (answer) > 0 and (answer) < 101 >
    ask What's your age? and wait
```

The loop continues to ask the same question until the user answers it correctly.

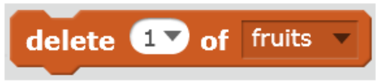
Variables – lists

A list variable is a type of variable that stores multiple pieces of information: words, numbers, or sentences.

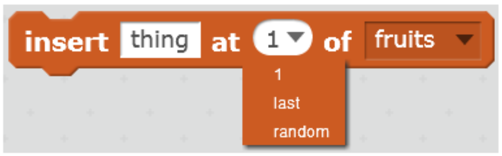
Click on “Make a List” under “Data” to create a new empty list variable, and then use the following commands to manipulate the list.



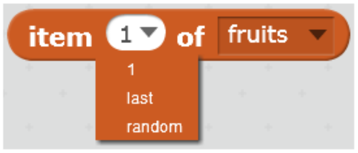
Add a new member at the end of the list



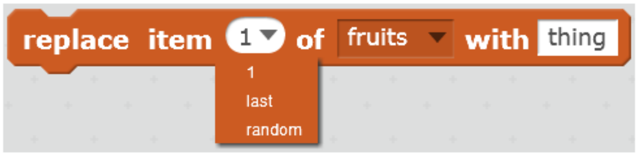
Delete a member



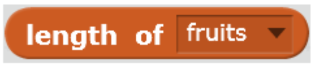
Insert a new member in the list (before the given location)



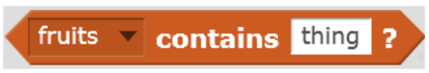
Get a member of the list



Replace by a new member

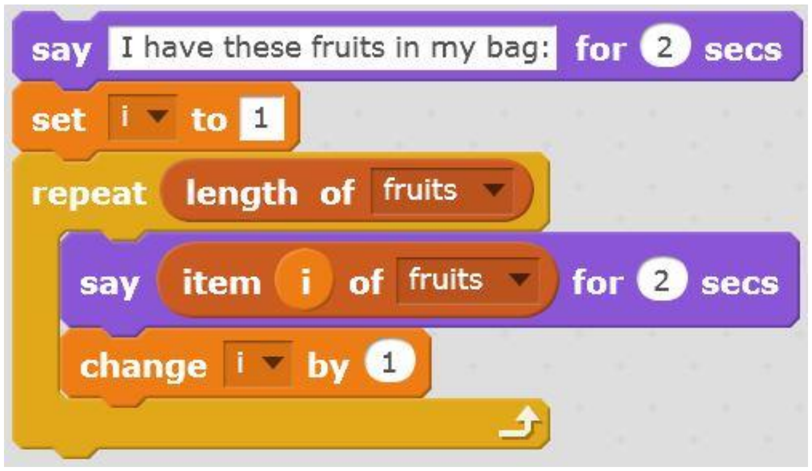


Get the size of the list



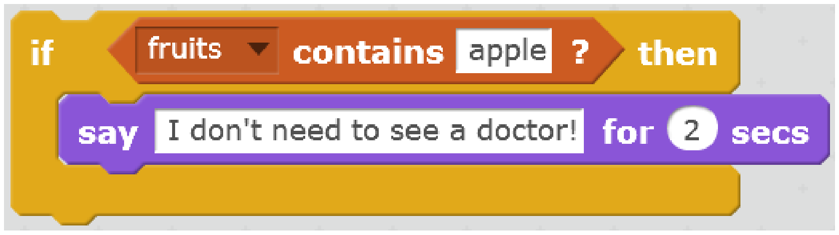
Does the list contain the given fruit?
This condition can be used in an IF statement.

Here is an example script that shows how the members of a list can be listed:



```
say [I have these fruits in my bag:] for (2) secs
set [i v] to [1]
repeat (length of [fruits v])
    say (item (i) of [fruits v]) for (2) secs
    change [i v] by (1)
end
```

The following condition checks if the given item is present in the list:



```
if <[fruits v] contains [apple] ?> then
    say [I don't need to see a doctor!] for (2) secs
end
```

Variables – numbers

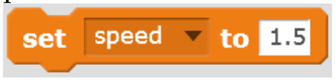
As you might know already, computers have memory. They use this memory to store information and do their work. For example, when you use a calculator to make calculations, the computer stores the numbers you type in its memory. This memory is called "temporary memory" because the information goes away if you shut down the computer.

And what is a variable? A variable is a location in this temporary memory, and each variable has a name. In Scratch, you can create a variable by clicking on the button "Make a variable". Once you create a variable it is available to your program to store information.

If you don't need a variable, you can delete it by right-clicking on the variable name and selecting "delete variable".

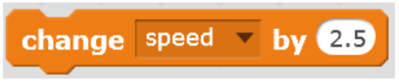
The word "variable" means something that can vary or change; indeed variables can contain any value. Scratch provides commands that you can use in your scripts to store information in variables and then change it later.

The SET command stores a value in a variable. The value can be any number – negative or positive, whole, or decimal.



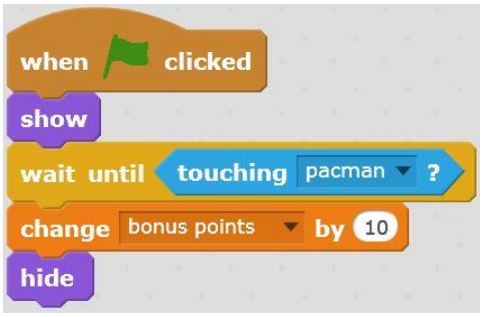
```
set speed to 1.5
```

Later, you can change the value by using the command CHANGE. The CHANGE command adds a number to the variable.



```
change speed by 2.5
```

Here is an example of how variables can be used. Here we have a script for the game of maze. We have a variable called "Bonus points". In this script, this variable is incremented by 10 if the packman touches the sprite – which is probably a PRIZE sprite.



```
when [flag] clicked
show
wait until touching pacman ?
change bonus points by 10
hide
```

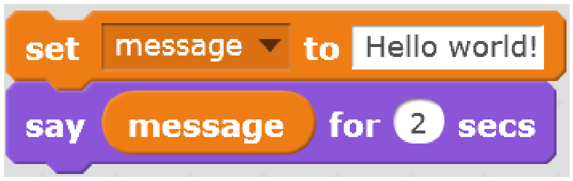
Important: Always initialize number variables to some value (such as 0) at the beginning of the program.

Variables – strings

The SET command stores a value in a variable. The value can be any number – negative or positive, whole or decimal. The value can also be a string of characters. For example, you can create a variable called “Name” and store the value “John Luke Pickard” in this variable.

Obviously the CHANGE command will not work for a string of characters.

Here is an example of how a string variable may be used. We have a variable called “Message”. In that we first store the string “Hello World”. Then, the SAY command takes that string from the variable “message” and prints it on the screen.



```
set message to Hello world!
say message for 2 secs
```

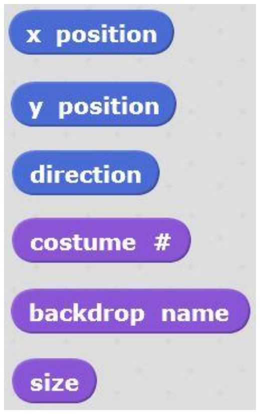
So, as you can see, you can use variables instead of actual values. When you refer to a variable in your script, the value stored inside that variable is used.

String variables can be modified by the “string operators”. For more information, see the concept “string operators”.

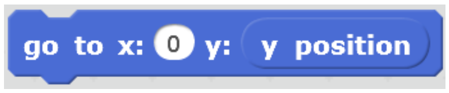
Variables - properties (built-in)

Scratch comes with many useful variables that provide some information about the sprites, the program itself, or something else. We call them *properties* to differentiate from variables that we create in the program.

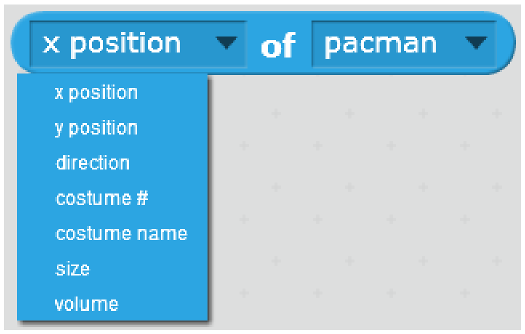
For example, every sprite in your project has several properties; some examples are shown below. Their names clearly indicate what type of information they contain. For example, "x position" contains the current X coordinate of the sprite.



Properties can be used just like any other variable. For example, the command below will make the sprite move left or right where X=0.



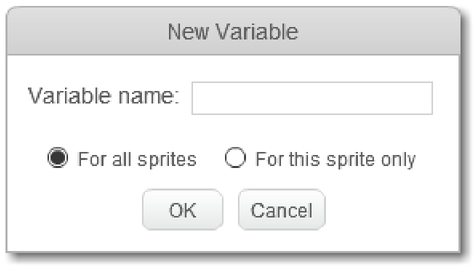
Scratch also provides a way for one sprite to access information about other sprites through the following set of properties:



One big difference between your variables and properties is that properties are read-only; their values cannot be modified directly. For example, the "x position" property will change only when the sprite moves along the X axis.

Variables - local/global scope

When you create a variable, you get the following dialog box:

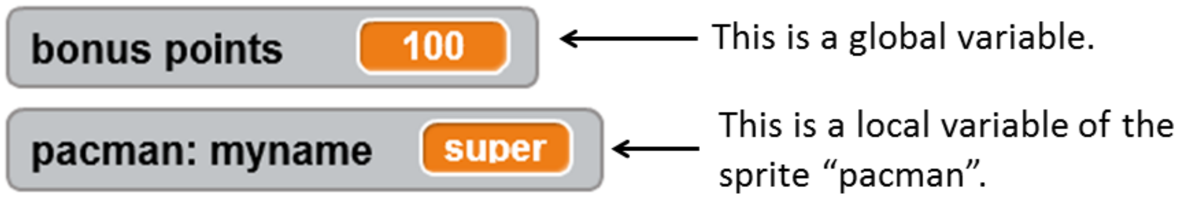


Below the name, there are two options to select from: (1) For all sprites, and (2) For this sprite only. If you select the first option, you create a "global" variable, and if you select the second option, you create a "local" variable.

A global variable is visible to all sprites in your project. Any sprite can set it, change it, or use it. On the other hand, a local variable is visible only to this sprite (one in which you created it).

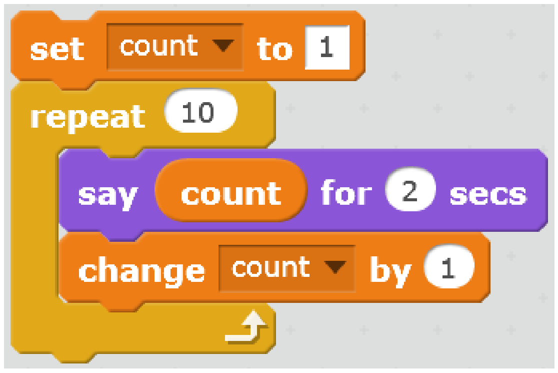
It is a good practice to decide the scope of each variable carefully, and make it "global" only if it is clearly going to be used by multiple sprites. For example, the "score" variable in a "pacman" game would be needed by multiple sprites (prizes, pacman, obstacles, etc.) so it should be "global".

A local variable, if displayed, uses a different notation. See below:


bonus points
100
This is a global variable.
pacman: myname
super
This is a local variable of the sprite "pacman".

Variables - as counters

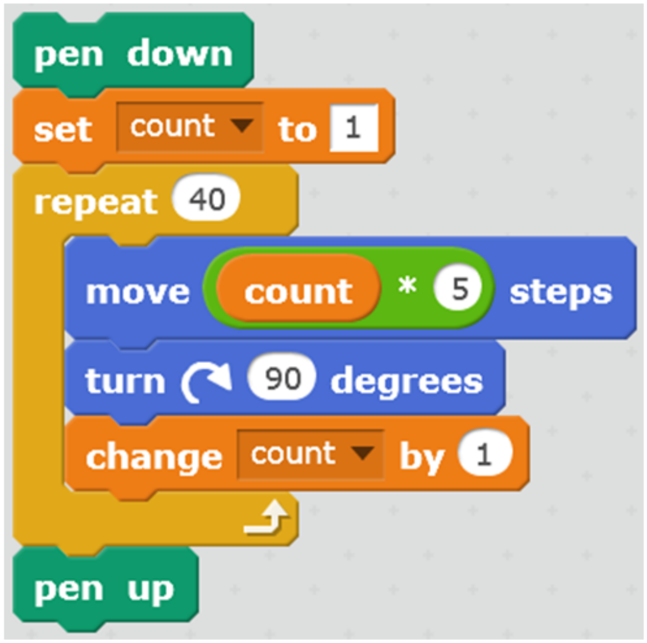
In the script below, the variable “count” counts the number of repetitions:



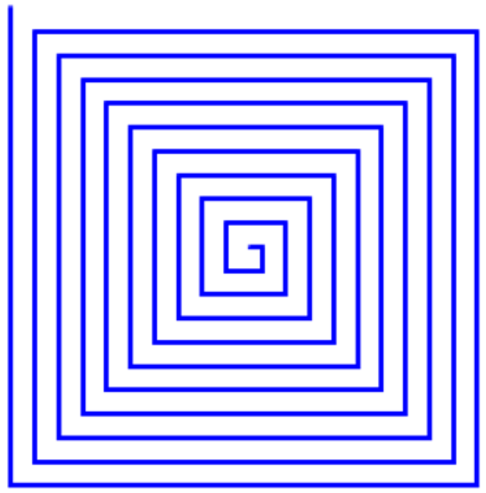
```
set count to 1
repeat 10
    say count for 2 secs
    change count by 1
```

It essentially counts from 1 to 10. This is called a *counter*.

Counters can be used for a variety of applications. The following script, for example, draws a squiral (a square spiral) using a counter:



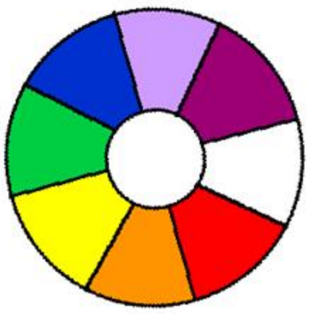
```
pen down
set count to 1
repeat 40
    move count * 5 steps
    turn ↻ 90 degrees
    change count by 1
pen up
```

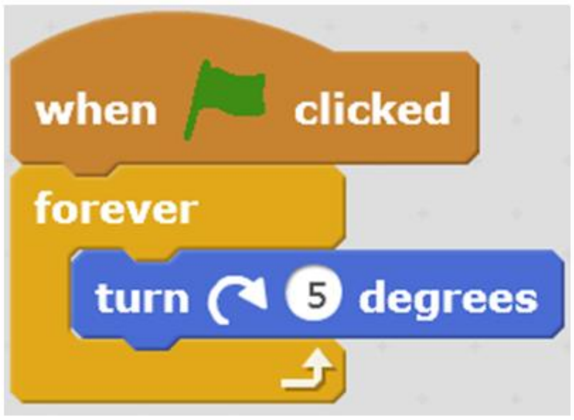


Variables - as remote control

This concept is best explained through an example.

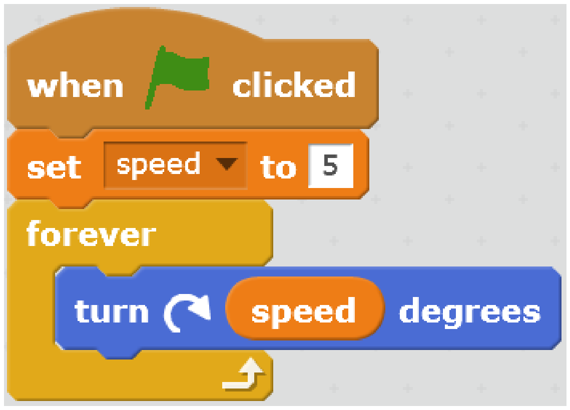
Let's say we have a spinning wheel as shown below:





```
when [flag] clicked
forever
    turn ↻ (5) degrees
```

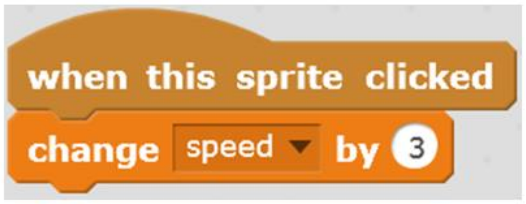
If we wanted to change the speed of this wheel, we will need to change the input of the "turn" command. Instead of manually changing the number, we could use a variable in place of this input:



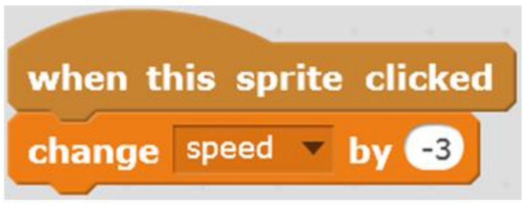
```
when [flag] clicked
set [speed ▾] to [5]
forever
    turn ↻ (speed) degrees
```

Now, whenever the variable "speed" changes, the spinning speed will also change immediately. So, the variable has become a "remote control" of the spinning wheel.

If you create button sprites labelled "Faster" and "Slower" with scripts as shown below, you will see how the remote control works:



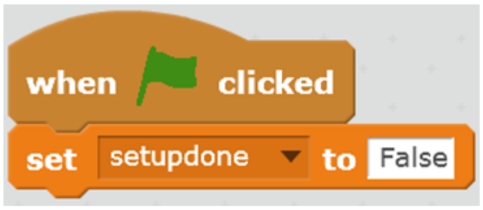
```
when this sprite clicked
change speed by 3
```



```
when this sprite clicked
change speed by -3
```

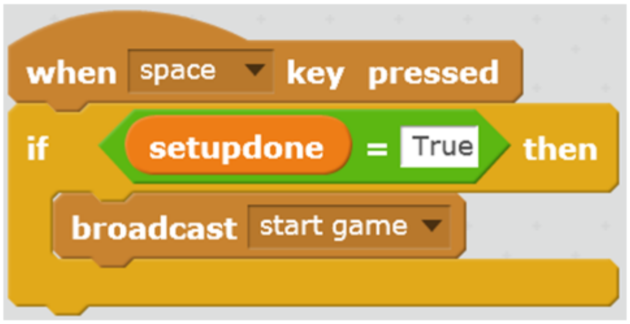
Variables - as gates

There are occasions when the user must not be allowed to start playing the game until setup is complete. For example, let's say your game starts when the SPACE key pressed. But, before pressing the SPACE key the user must set a few things, for example, slider variables. We can enforce this by using the concept of "using variables as gates". In the example below, we will use a variable called "setupdone" which will be False initially and True after setup is done.



```
when clicked
set setupdone to False
```

Set the variable to False initially.

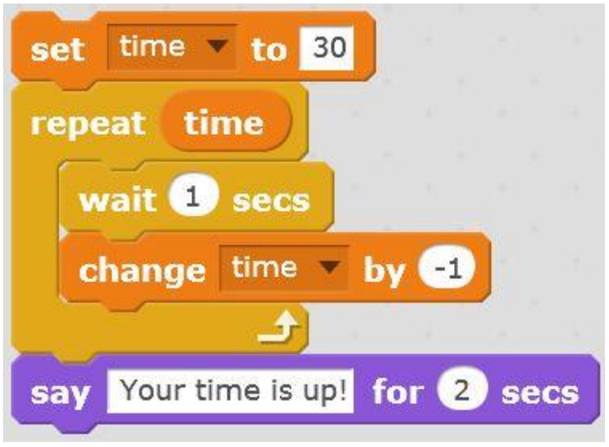


```
when space key pressed
if setupdone = True then
    broadcast start game
```

Start the game (by sending broadcast) only when set up is done. The variable will be set to True by whoever is in charge of setup.

Variables - as timer

The combination of variables and the WAIT command can be used to implement a timer. See the script below:

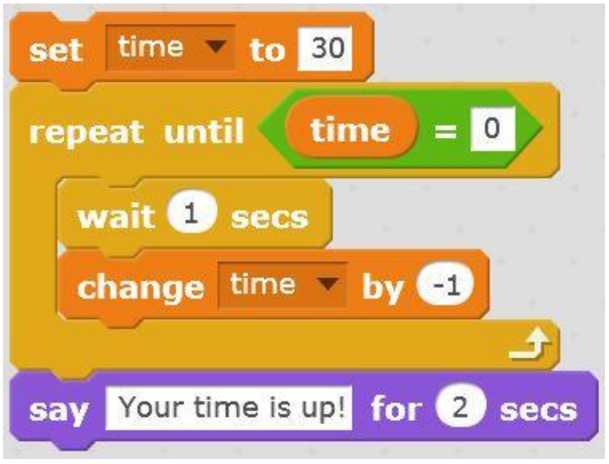


```
set time to 30
repeat time
    wait 1 secs
    change time by -1
say Your time is up! for 2 secs
```

The "time" variable is set in the beginning to whatever value you want to set for your game. The repeat loop then decrements it a second at a time until it becomes 0.

It may be confusing to see the "time" variable getting decremented and also being used in the REPEAT command. The REPEAT command uses its initial value (in this case 30) to decide the number of repetitions.

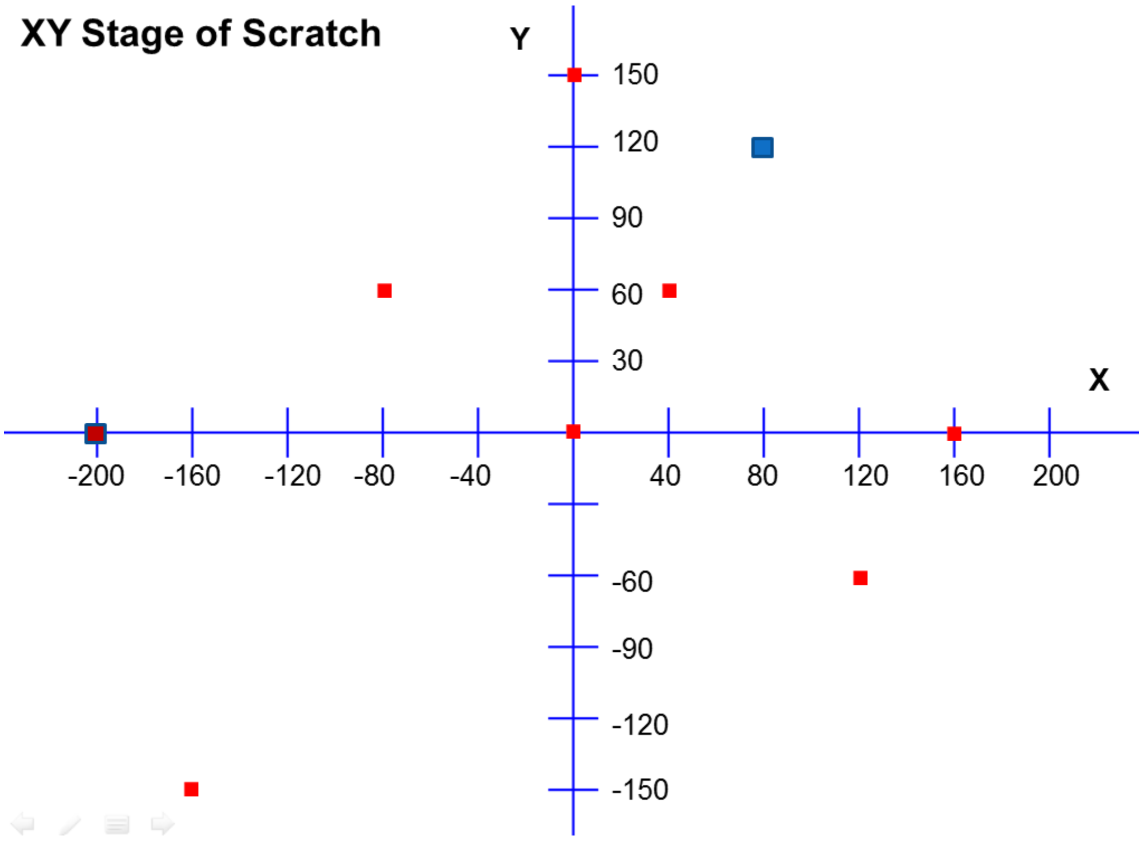
A more intuitive (and less confusing) way to do this might be:



```
set time to 30
repeat until time = 0
    wait 1 secs
    change time by -1
say Your time is up! for 2 secs
```

XY Geometry

We use the screen in Scratch as a geometric plane. Every point on the screen has X and Y coordinates. You might have heard about this in your school Mathematics.



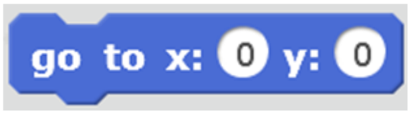
Every point on the screen has two coordinates: x and y. Imagine two number lines laid out on the screen: one horizontally and the other vertically, both intersecting at 0. The horizontal line is called the X axis, and the vertical line is called the Y axis. The distance of a point from the Y axis is called the X coordinate, and the distance from the X axis is called the Y coordinate.

For example, for the blue point here its distance from the Y axis is 80, so its X coordinate is 80. And its distance from the X axis is 120, so its Y coordinate is 120.

For the dark red point its distance from the Y axis is -200, so its X coordinate is -200. And its distance from the X axis is 0, so its Y coordinate is 0.

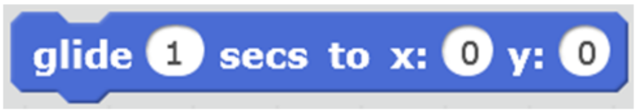
Every sprite in Scratch has X Y coordinates. Each sprite has a center point which you can set in the Paint editor. The X Y coordinates of this center point are assumed to be the X Y coordinates of the sprite. When the sprite moves these values also change automatically.

Now that we understand what X Y coordinates are let us look at some Scratch commands that make use of these coordinates.



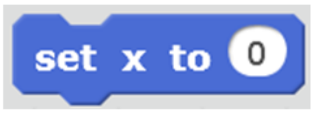
```
go to x: (0) y: (0)
```

The Go To command makes the sprite jump to a specific point on the screen. The jump is instant, not a smooth one.

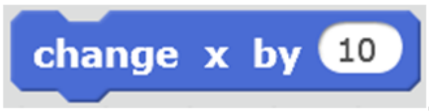


```
glide (1) secs to x: (0) y: (0)
```

The Glide command makes the sprite move smoothly to another point. The sprite's speed depends on the number of seconds.

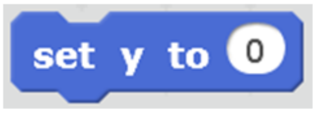


```
set x to (0)
```

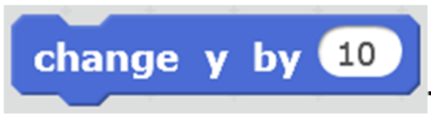


```
change x by (10)
```

These commands will move the sprite only left and right because its Y coordinate will remain the same.



```
set y to (0)
```



```
change y by (10)
```

These commands will move the sprite only up and down because its X coordinate will remain the same. So you could use these commands to make a sprite jump up or down.

The SET commands are interesting because they move the sprite to a specific point no matter where the sprite is right now. Whereas, the CHANGE command depends on the sprite's current location.

For example, if you say CHANGE X by 10 the sprite will move to its right by 10 pixels. If you run CHANGE X by 10 once more, again the sprite will move to its right by 10 pixels.

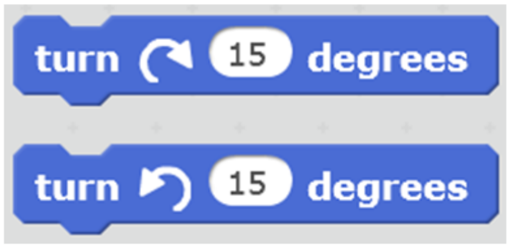
But, if you say SET X to 100, the sprite will jump to where X is 100. And if you run SET X to 100 once more, nothing will happen because the sprite is already at X = 100.

There is one important thing to note about all these commands shown here. The orientation, or, the direction in which the sprite is facing, does not change when you use any of these commands.

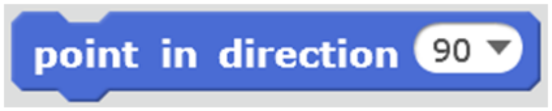
Ok, so far we have seen how to change the position of sprites, that is, how to move them from where they are. All these commands do not affect the orientation of the sprites.

Orientation of a sprite is basically the direction in which it is facing. It is shown in terms of the angle made with the North direction. So if a sprite is facing North its direction is 0.

Initially all sprites are facing east, which means their direction is 90. You can change a sprite's orientation using one of the commands shown below.



The TURN commands turn the sprite around its center point either clockwise or anti-clockwise.



This command makes the sprite face in any direction you want. You can pick one from the drop-down list, or you can enter a value of the angle.