SCRATCH for Budding Computer Scientists

by David J. Malan <malan@post.harvard.edu>

Table of Contents

Introduction

Most programming languages, on first glance, "look like Greek" to the untrained eye, an amalgam of English and unusual syntax. Consider, for instance, the program below, written in a language called Java.

```
class Hello
{
    public static void main(String [] args)
    {
        System.out.println("hello, world!");
    }
}
```

All the program above does, when executed, is display "hello, world!" on the user's screen. You might have guessed as much just by looking over the code—and ignoring anything that didn't make sense! But what's with all the curly braces (`{` and `}`)? What's `System.out`? What does `class Hello` mean? And `public static void main(String [] args)`? Let's not even go there.

Suffice it to say that, when it comes to learning to program, there's quite a learning curve with languages like Java. Before you can begin to solve problems, you must first learn to read and write a new language, even if the task at hand is relatively simple (*e.g.*, "hello, world!"). And whereas you might still understand a foreigner who mispronounces some English word, computers aren't so forgiving when it comes to mistakes. Leave out a semicolon, and the program above won't even work!

Learning to program is ultimately about learning to think logically and to approach problems methodically. The building blocks out of which a programmer constructs solutions, meanwhile, are relatively simple. Common in programming, for instance, are "loops" (whereby a program does something multiple times) and "conditions" (whereby a program only does something under certain circumstances. Also common are "variables" (so that a program, like a mathematician, can remember certain values).

For many students, the seemingly cryptic syntax of languages like Java tends to get in the way of mastery of such relatively simple constructs as these. Before we tackle a language like Java, then, with its curly braces and semicolons, we turn our attention to Scratch, a "new programming language that lets you create your own animations, games, and interactive art." Although originally developed for kids by the Lifelong Kindergarten research group at the MIT Media Lab, Scratch is just as useful (and fun) for budding computer scientists. By representing programs' building blocks with color-coded blocks (*i.e.*, puzzle pieces), Scratch "lowers the bar" to programming, allowing budding computer scientists to focus on problems rather than syntax, to master programmatic constructs rather than syntax. Syntax, of course, will come later. But, for now, we focus on programming itself. It just so happens that programming, for now, will be more like putting together a puzzle than writing Greek.

This tutorial introduces budding computer scientists to the building blocks of programming by way of Scratch. It assumes that you are already familiar with Scratch's usage and, accordingly, have a general sense of how to program with Scratch. This tutorial aspires to formalize your understanding of programming, framing some basic programming constructs in the language of Scratch.

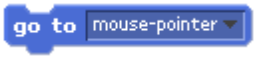
We turn our attention first to statements.

Statements

In programming, a **statement** is simply a directive that tells the computer to do something. Think of it as a command or an instruction. In Scratch, any block whose label reads like a command is a statement.

One such block instructs a sprite to say something:

Another such block instructs a sprite to go to some location:



Sometimes, you only want a statement to be executed under certain conditions. Such conditions are defined in terms of Boolean expressions, to which we turn our attention next.

Boolean Expressions

In programming, a **Boolean expression** is an expression that is either true or false. In Scratch, any block shaped like an elongated diamond is a Boolean expression.

One such block is:

After all, it is either true that the mouse button is down or it is false.

Another such block is:

After all, it is either true that some number is less than another number or it is false.

With Boolean expressions can we construct conditions, to which we turn our attention next.

Conditions

In programming, a **condition** is something that must be true in order for something to happen. A condition is thus said to "evaluate to true" or "evaluate to false." In Scratch, any block whose label says "if," "when," or "until" is a sort of conditional construct.

One such block is:

The construct above is generally known as an "if construct." With it can we instruct a sprite to say hello only if, say, the user has depressed the mouse button:

A related construct is the "if-else construct":

With the above construct can we instruct a sprite to say hello or goodbye, depending on whether the user has depressed the mouse button:

Realize that these constructs can be nested to allow, for example, for three different conditions:

The above construct could be called an "if-else if-else construct".

Another conditional block is:

Yet another such block is:

Sometimes, you want one or more statements to be executed multiple times in a row. To implement this

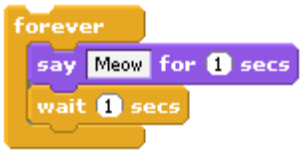
behavior, we turn our attention to loops.

Loops

In programming, a **loop** can induce multiple executions of statements. In Scratch, any block whose label begins with "forever" or "repeat" is a looping construct.

One such block is:

This construct allows us, for instance, to instruct a sprite to meow every other second:



Another block allows you to loop a specific number of times:

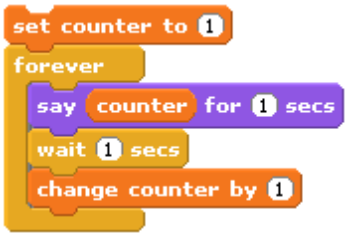
And another block allows you to loop until some condition is true:

Sometimes, you want execute some statement multiple times, each time varying your behavior ever so slightly. We thus turn our attention to variables.

Variables

In programming, a **variable** is a placeholder for some value, much like *x* and *y* are popular variables in algebra. In Scratch, variables are represented with blocks shaped like elongated circles, uniquely labeled by you. Variables, generally speaking, can be **local** or **global**. In Scratch, a local variable can be used by just one sprite; a global variable can be used by all of your sprites.

Variables allow us, for instance, to instruct a sprite to count up from 1:



A variable that only takes on a value of true (*i.e.*, 1) or false (*i.e.*, 0), incidentally, is called a **Boolean variable**.

With statements, Boolean expressions, conditions, loops, and variables now under your belt as building blocks, we can now explore two higher-level programming constructs, starting with threads.

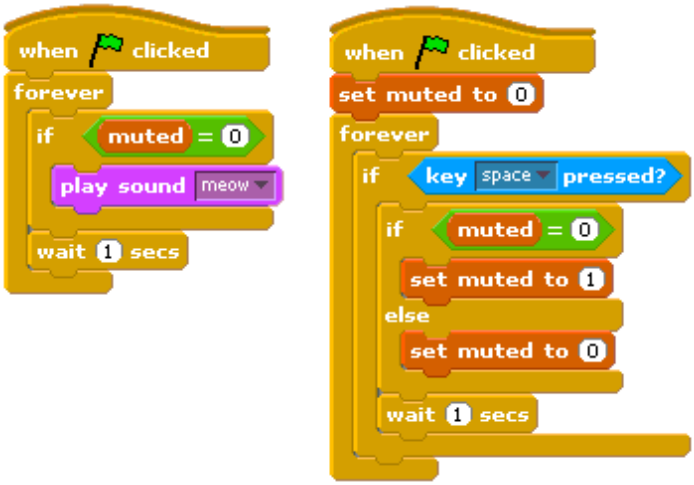
Threads

In programming, a **thread** is like a mini-program within a program that can execute at the same time as other threads. A program with multiple threads, then, can do multiple things at once. In Scratch, any block whose label begins with "when" essentially demarks the start of a thread; think of what Scratch calls a "script" as a thread. (Technically, scripts run *in* threads, but never mind that.)

One such block is:

As the above block's label suggests, this thread begins to execute when the user click's Scratch's green flag. A program with two such blocks thus has two "threads of execution," both of which start simultaneously when the user clicks Scratch's green flag.

It's often helpful to use separate threads for conceptually distinct tasks. For instance, you might want to keep track of whether the user ever presses some key during a program's execution in order to, say, toggle sound on and off:



Notice how, in the above, the left-hand thread handles meowing, if appropriate, whereas the right-hand thread constantly checks and remembers whether the user has muted or unmuted sound by pressing the space bar.

Related to threads are events, to which we turn our attention next.

Events

In programming, multiple threads can communicate with each other by signaling events and handling events. An event, then, is like a message from one thread to another. In Scratch, blocks whose labels begin with "broadcast" signal events whereas blocks whose labels begin with "when" handle events, the latter of which, recall, effectively represent threads themselves.

A block that signals an event is:

A block that handles an event is:

Not only can events be signaled by blocks, they can also be signaled by a user's actions. Clicking Scratch's green flag, for instance, effectively signals an event that is handled by:

In Scratch, not only do events enable threads to communicate, they also allow sprites to communicate with

each other. For instance, two sprites might want to play Marco Polo with each other, with one sprite's behavior defined by the leftmost thread below and the other sprite's behavior defined by the rightmost thread below:

Out of statements, Boolean expressions, conditions, loops, variables, threads, and events can you construct interesting (and fun) programs. In fact, let's explore the inner workings of what, on first glance, appears to be a very complex program but, ultimately, is just an application of these building blocks.

Let's turn our attention to Oscartime!

Oscartime

Okay, it's Oscartime! Oscartime is a game, written in Scratch, that challenges a player to drag as much falling trash to Oscar's trash can as possible before Oscar finishes singing a classic song. Here's a screenshot:

Go ahead, if you haven't already, and download Oscartime.sb to, say, your desktop. Then, go ahead and open it within Scratch. Click Scratch's green flag, read the game's instructions, and have yourself just over two minutes of fun! (True fans can even bookmark the Web-based version.)

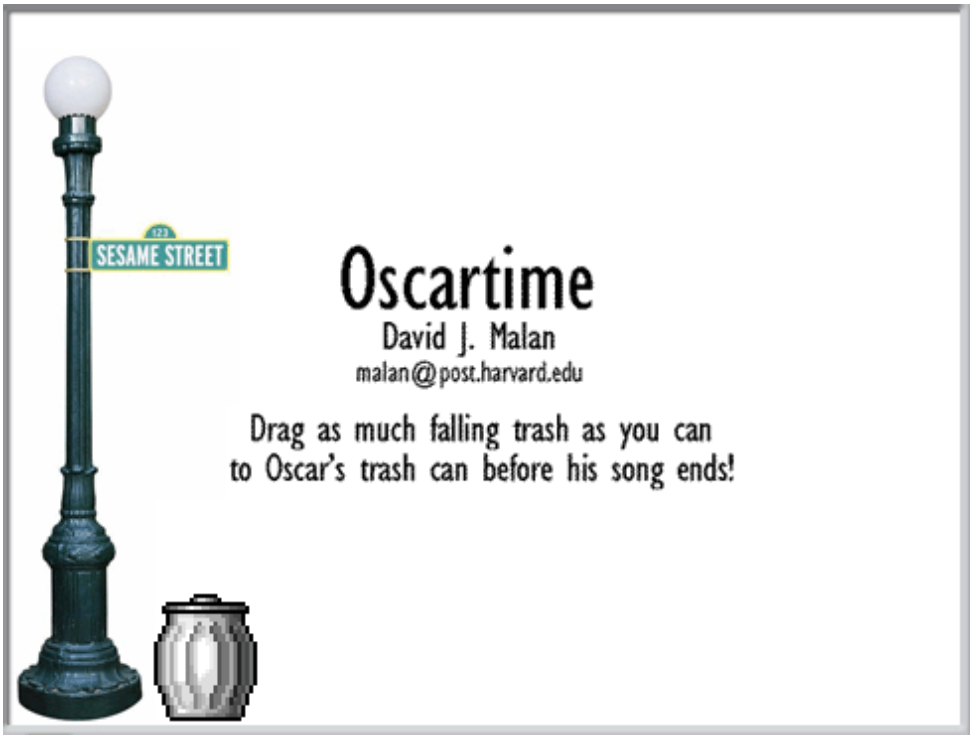
Oscartime is implemented with nine sprites, each of which uses between one and three threads. Let's explore

this tutorial's programmatic constructs in the context of Oscartime so that you understand not only how to play the game but how to implement such a game yourself!

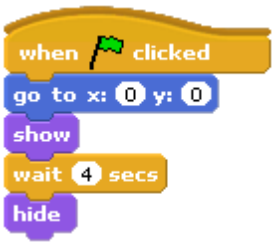
Let's first examine Oscartime's Instructions sprite.

Oscartime's Instructions Sprite

Oscartime's Instructions sprite is responsible for the brief display of the game's instructions at startup:



This sprite uses just one thread to display the game's instructions, which are implemented as a costume, centered on the screen for four seconds:



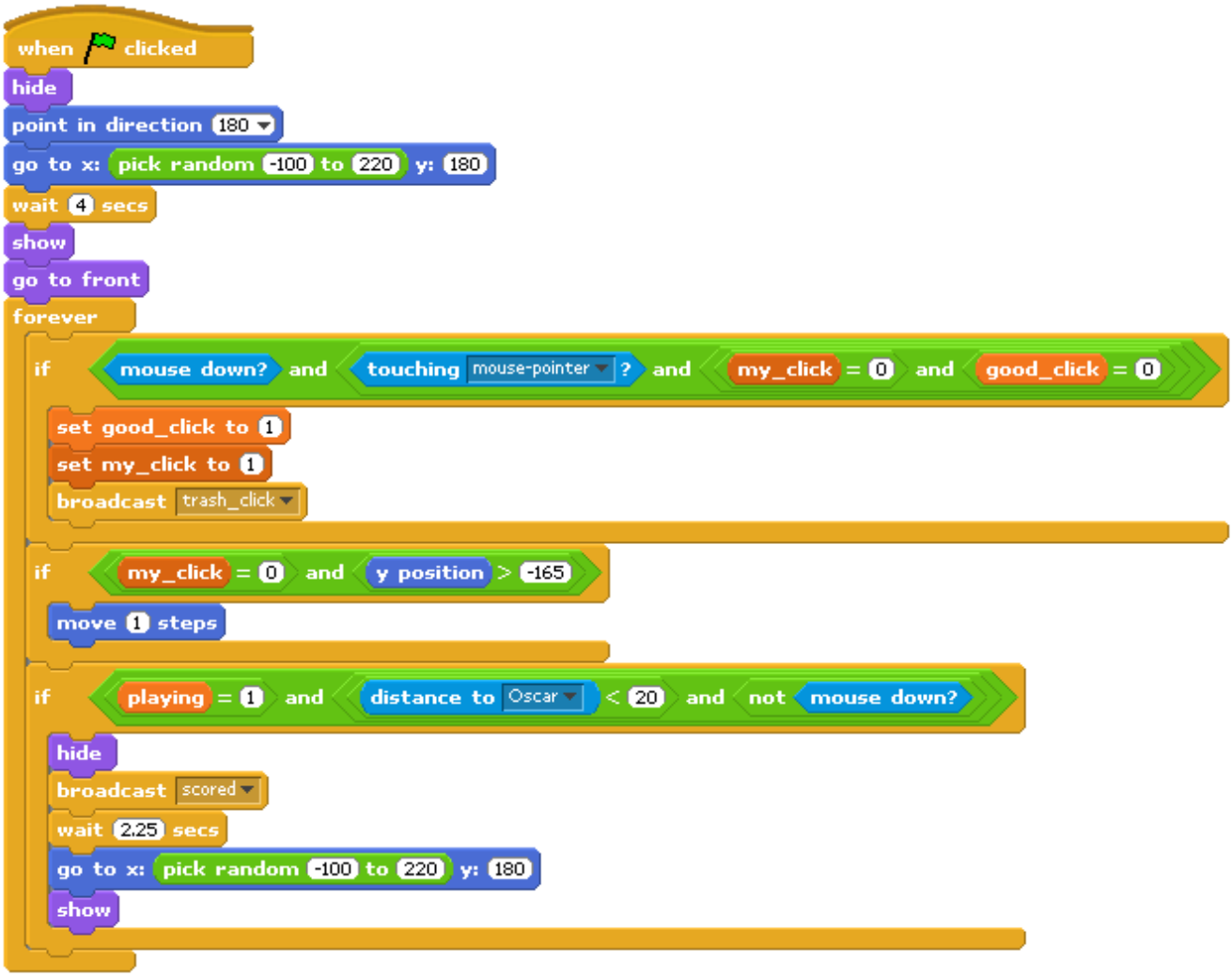
Pretty simple stuff. Let's now examine the first of the falling sprites, Oscartime's Trash sprite.

Oscartime's Trash Sprite

Oscartime's Trash sprite is designed to fall from a random location in the sky to the ground:

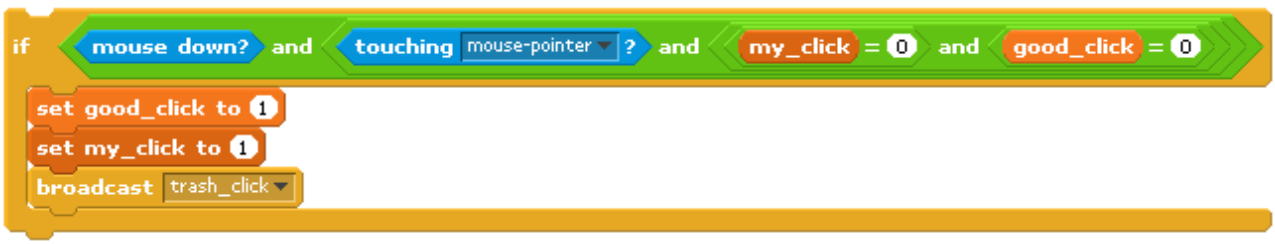
Once picked up by the player and dragged to Oscar's trash can, the sprite falls from a new location, taunting the player. The process repeats forever. Of course, each disposal of the sprite is worth a point!

Let's examine the first of this sprite's threads:



This thread is pretty complicated, so let's consider it parts. Needless to say, this thread begins executing when the user clicks Scratch's green flag. The thread's first statements essentially aim the sprite downward, place it (invisibly) at some random location atop the screen, and, after four seconds, reveal the sprite:

Let's now consider the first conditional construct in the thread's loop:

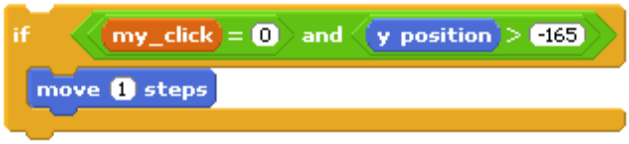


Essentially, the construct above asks:

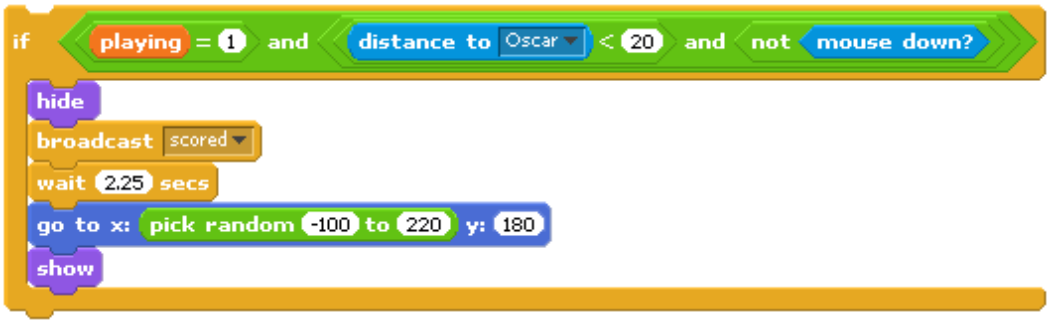
1. Is the player depressing the mouse button? (If so, the player is presumably clicking on something.)
2. Is the mouse touching this sprite? (If so, it is presumably on this sprite that the player is clicking.)
3. Has the player *just* clicked on this sprite? (If so, we need to enter drag mode. If not, the player's already in drag mode!)
4. Is the player definitely not clicking on some other sprite already? (We want to make sure the player can only pick up one sprite at once, even if they're overlapping on the screen.)

If all four conditions are met, the sprite sets a global, Boolean variable (named good_click) to true, so that the program remembers that a sprite (and not some random location on the screen) has, in fact, been clicked on for dragging; it also sets a local variable (named my_click) to true, so that the sprite knows that it is it that is being dragged); and it broadcasts an event (named trash_click) to itself so that a separate thread can handle the actual dragging.

If not all four conditions are met and the sprite is not already being dragged, the loop's second conditional construct induces the sprite to "fall" downward by one step unless it already appears to be lying on the ground:



The loop's final conditional construct determines whether the sprite has been dropped close enough to Oscar's trash can to be considered deposited (and thus worthy of a point):



Essentially, the construct above asks:

1. Is the game still in progress? (Once Oscar finishes his song, after all, we don't want to accept any more trash.)
2. Is the sprite within 20 pixels of Oscar's trash can? (If so, that's close enough to be considered deposited.)
3. Is the player not depressing the mouse button? (Trash ain't deposited until you let go!)

If all three conditions are met, the sprite hides itself (as though it's been deposited), broadcasts an event (named scored) to one of the Oscar sprite's threads, waits a couple of seconds, moves to a new location in the sky, and reveals so as to begin a new descent.

Because all three of these conditional constructs are nested within a block labeled "forever," the sprite behaves as it should until game's end.

The aesthetics of dragging, meanwhile, are handled by a second thread:

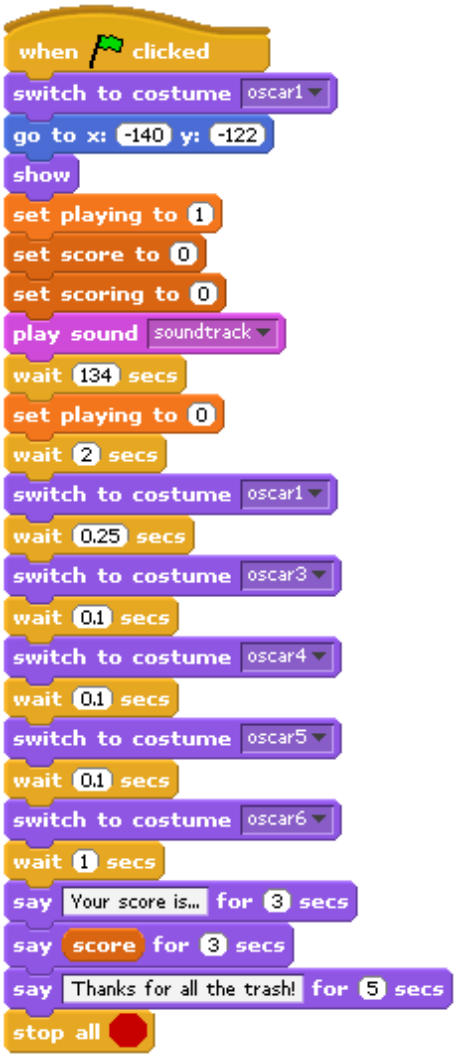
Essentially, so long as, upon clicking the sprite, the player keeps the mouse button depressed, the sprite will follow the mouse's movements, thus creating the appearance of being dragged. As soon as the player releases the mouse button, the thread notes that neither this sprite nor any other is, for the moment, being clicked on anymore. The thread then "dies," to be re-"spawned" only when the trash is picked up again.

Oscartime's Sneaker, Newspaper, Clock, Telephone, Umbrella, and Trombone sprites essentially behave just like this Trash sprite, the only differences being the times at which they first appear. (Their appearance is synchronized with Oscar's first mention of them in his song.)

So let's conclude by looking at the last of Oscartime's sprites, Oscartime's Oscar sprite.

Oscartime's Oscar Sprite

Oscartime's Oscar sprite uses three threads to keeps track of a player's score and the announcement thereof. The first of these threads provides the game's overall framework:

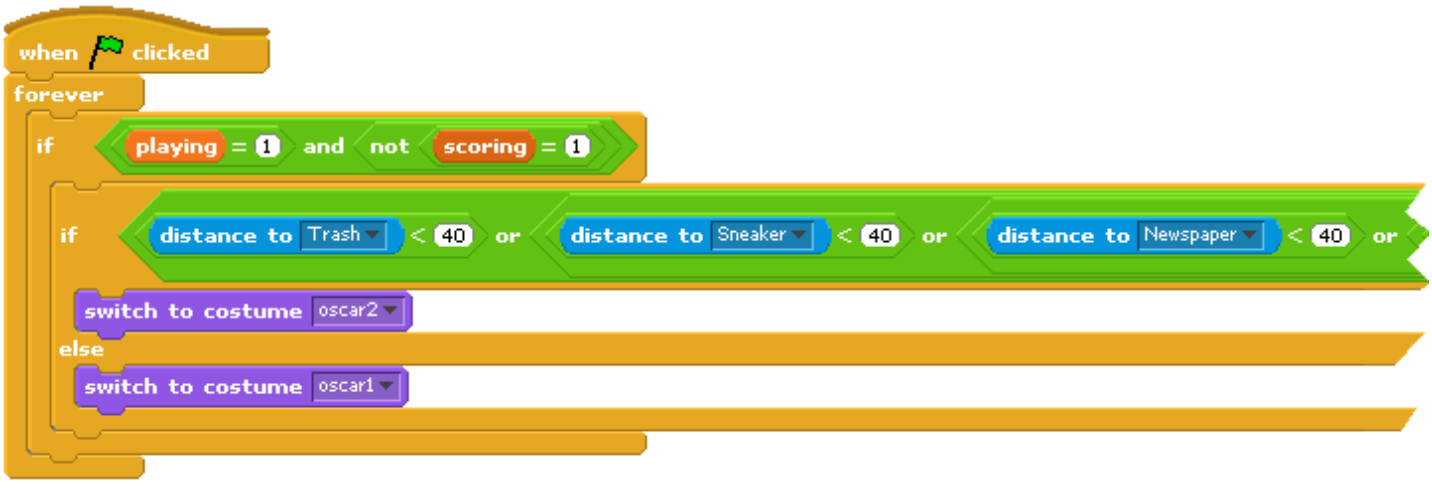


Essentially, this thread dresses the sprite in its default costume (that of a closed trash can); moves the sprite to its permanent location; sets the global, Boolean variable (named playing) to true, so that the other sprites know a game's in progress; and then plays Oscar's song whilst the player plays the game. Once that song ends (after 134 seconds), the sprite sets the global, Boolean variable (named playing) to false, so that the other sprites know the game's over; announces the player's score, then kills all threads.

The sprite's second thread keeps track, moment by moment, of a player's score:

Essentially, every time some other sprite signals an event (named scored), the above thread handles that event by setting a local, Boolean variable (named scoring) to true, so that the sprite's other thread knows not to change Oscar's costume while this thread is doing so, thereafter popping Oscar out of his trash can to announce the player's current score.

The sprite's third thread induces the lid of Oscar's trash can to rise anytime trash is dragged near it:



Essentially, the lid rises so long as a game is in progress, the sprite isn't already changing Oscar's costume in order to announce the player's score, and some sprite is within 40 pixels of Oscar's trash can.

That's pretty much how Oscartime works. The game's implementation involves lots of blocks, to be sure. But, ultimately, the game is just the result of piecing together building blocks of programming.

Now, though Scratch does support programmatic constructs common to many programming languages, it doesn't do everything. In this tutorial's conclusion, let's survey some programmatic constructs that Scratch doesn't have.

Conclusion

Though Scratch does support many programmatic constructs, it doesn't support 'em all. Common to many programming languages but missing from Scratch are:

- methods, which allow you pass control of execution from one sequence of blocks to another;
- parameters, which allow you to influence the behavior of methods;
- return values, which allow one sequence of blocks to "return" information to another;
- inheritance and polymorphism, which allow relationships to exist among data structures.

However, to a budding computer scientist, we daresay Scratch is valuable *because* it omits support for such features as these. What Scratch very much does offer is an intuitive, fun environment in which basic tenets of programming can be explored and deployed without complication by syntax.

May that Scratch ultimately help you, the budding computer scientist, focus less on semicolons and more on the solving of problems.
