Table of Contents

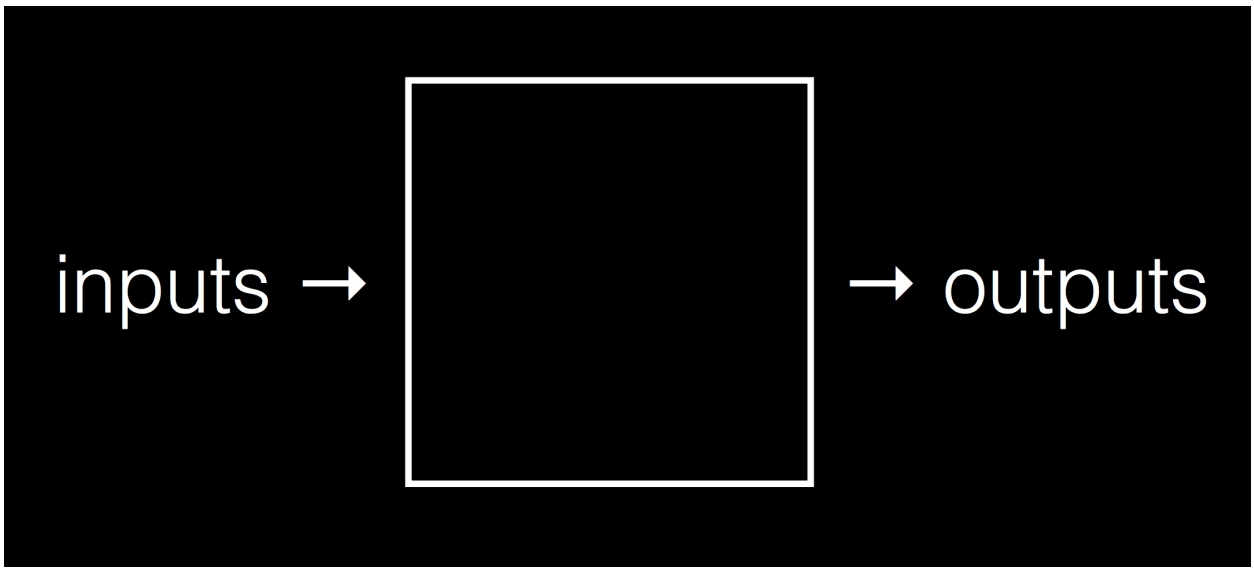
Lecture 0

Introduction

- When David was just starting his undergraduate career and considering Computer Science as a field, he too had some perceptions that it was primarily coding alone, and that everyone around him would already know more than he did. (Actually, he thought the second part about every course in the catalog.)
- So David took government, economics, and history courses instead, because that's what he liked in high school. In sophomore fall, he took CS50 (back when it was taught by someone else), but only because he was allowed to take it Pass/Fail (now called Sat/Unsat). The rest, as they say, is history, but only because David stepped a little bit outside his comfort zone!
- Today (more specifically, last year), 68% of CS50 students have never taken CS before.
- The course indeed has different tracks for its sections: Less Comfortable, Somewhere in Between, and More Comfortable. There are no formal definitions, and students place themselves into these sections.
- The grading of the course is guided by the following:
    - what ultimately matters in this course is not so much where you end up relative to your classmates but where you, in Week 11, end up relative to yourself in Week 0
- CS50's team have created project5050.org, where former students and staff have shared their stories and challenges in their own journeys in computer science.
- And the course itself, as is the field of computer science, is less about strictly programming and more about problem solving. As opposed to an end in itself, CS is a tool we can use to do more powerful and interesting things in any other field.

Binary

- We might be able to represent problem solving most simply as a black box, which take some inputs (the problem we have) and produce some outputs (the solution we want):



- We could describe this black box as an algorithm, a series of steps to solve a problem.
- But we need some standard ways to represent our inputs and outputs.
- Computers use the binary system, with two digits, 0 and 1, as opposed to humans, who typically use the decimal system, with 10 digits.
- We can count in unary with our hands, by raising one finger for each number we count. So one hand can count up to five.
- We can use the patterns of how our hands are raised to count higher. For example, just having our thumb extended could be 1, just our index finger could be 2, and both our thumb and index finger could be 3. Let's take a closer look.
- In decimal, `123` is one hundred and twenty-three, with each digit in a column:

```
    100         10          1

      1          2          3

100 x 1     10 x 2      1 x 3
```

  - The rightmost number is in the 1s place, the next one is in the 10s place, and the leftmost one above is in the 100s place.
- Binary represents numbers in the same pattern, but using powers of 2 instead of powers of 10 that decimal uses. The first row shows the value of each column, like the 100, 10, and 1 above, and the second row is our current binary number.

```
      4          2          1

      0          0          0
```

- To represent a 1, we simply place a `1` in the ones column:

```
      4          2          1

      0          0          1

            1 x 1
```

- And a 2 like so:

```
      4          2          1

      0          1          0

             2 x 1
```

- And a 3 by combining the previous two steps:

```
      4          2          1

      0          1          1

             2 x 1      1 x 1
```

- We can continue this pattern:

```
      4          2          1

      1          0          0

   4 x 1
```

```
      4          2          1

      1          0          1

   4 x 1                 1 x 1
```

```
      4          2          1

      1          1          0

   4 x 1      2 x 1
```

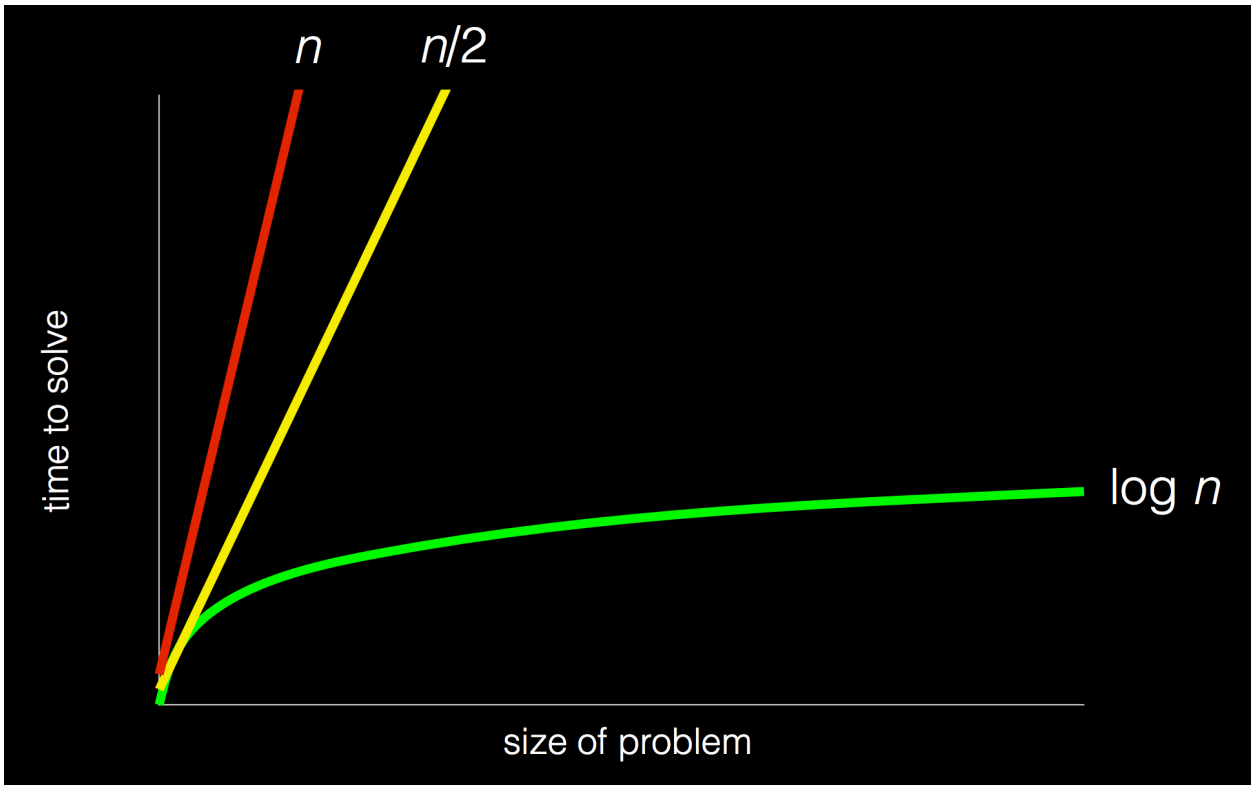
```
    4         2         1

    1         1         1

4 x 1     2 x 1     1 x 1
```

- But once we have used up all the places, we need more **bits**, or binary digit, which stores a `0` or `1`.
- It turns out that computers can conveniently represent a 0 or 1 with electricity, since something can either be turned on or off. And computers have lots of transistors, microscopic switches inside, that can be turned on and off to represent data.
- Now that we can store numbers, we need to represent words, or letters. Luckily, there is a standard mapping from numbers to letters, called **ASCII**.
- We can also similarly use certain standards to represent graphics and videos.
- A series of bits, that represent the numbers `72` `73` `33` might be the characters `H` `I` `!` in ASCII, but could also be interpreted by graphics programs as a color.
- RGB, for example, is a system where a color is represented by the amount of red, green, and blue light it is composed of. By mixing the above amounts of red, green, and blue, we get a color like a murky yellow. A picture on a screen, then, can be represented by lots and lots of these pixels, or single squares of color.
- For both ASCII and RGB, the maximum value that each character or amount of one color can be is 255, because one common standard group of bits is a **byte**, or 8 bits.
- In computer science, a common theme is **abstraction**, where we start by taking ideas to solve simple problems, and layering these solutions until we can build more and more interesting applications.

Algorithms

- Now that we know how to represent inputs and outputs, we can work on algorithms, which is just step-by-step instructions on how to solve a problem.
- **Computational thinking** is the idea of having these precise instructions.
- For example, David might want to make a peanut butter and jelly sandwich from bread, peanut butter, and jelly.
- The first step might be "open the bag of bread", and David rips the bag open.
- The next step is "remove two slices", and then "put those slices on the plate".
- Then "unscrew the jam", "grab the knife", and "stick the knife in the jam".
- We continue with these instructions that get more and more specific, until David completes his sandwich.
- In fact, when we write algorithms to solve problems, we need to think about cases when something unexpected happens. For example, the input might not be within the range of what we expect, so our computer might freeze or come up with an incorrect solution.
- We can see this in action with trying to find a name in the phone book, Mike Smith.
- One correct algorithm might be flipping through the phone book, page by page, until we find the person we are looking for.

- Another algorithm could be flipping through two pages at a time, but it's no longer correct since we might skip our friend Mike. We can fix this by adding another step, where if we notice we have passed our friend (since the phone book is alphabetized), we go back a page and check.
- We can also open the book to the middle, and find ourselves in the M section (by last name), and know that Mike Smith is in the right half of the book, and throw the left half away. We can repeat this again and again, and eventually have one page left to look at. With 1000 pages, it would only take about 10 steps of division to reach that one page.
- We can consider how fast each of these algorithms are, with a chart like this:



  - The size of the problem might be defined in this case as the number of pages in the phone book, or *n*.
  - So our first algorithm, going page by page, requires *n* steps to complete, since there are *n* pages.
  - The second algorithm, going two pages at a time, requires *n*/2 steps.
  - Our last algorithm is a different shape, with time to solve growing more and more slowly as the size of the problem increases, since we are dividing the problem in half with each step. So an increase from 1000 to 2000 pages only requires one more step to solve.
- We also need to formalize the steps we are using to solve this problem. We can write something like the following:

```
 0   pick up phone book
 1   open to middle of phone book
 2   look at names
 3   if Smith is among names
 4       call Mike
 5   else if Smith is earlier in book
 6       open to middle of left half of book
 7       go back to step 2
 8   else if "Smith" is later in book
 9       open to middle of right half of book
10       go back to step 2
11   else
12       quit
```

    - We start counting at 0 because that's the default lowest value, with all the bits off.
    - In step 3, we have the word `if`, which is a fork in the road, where the next step may not be taken, so we indent it to visually separate it from the lines that are always followed.
    - The last `else`, in step 11, happens if we're on the last page and Mike isn't in the phone book, since we can no longer divide it.
- These steps are **pseudocode**, English-like syntax that is similar in precision to code.
- Words like `pick up`, `open`, and `look` are equivalent to **functions** in code, like verbs or actions that allow us to do something.
- `if`, `else if`, and `else` are the keywords which represent forks in the road, or decisions based on answers to certain questions. These questions are called **Boolean expressions**, which have an answer of either true or false. For example, `Smith among names` is a question, as is `Smith is earlier in book` and `Smith is later in book`.
    - Notice too, that with one bit, we can represent true, with on, or 1, and false, with off, or 0.
- Finally, `go back` creates loops, or series of steps that happen over and over, until we complete our algorithm.

Introductions

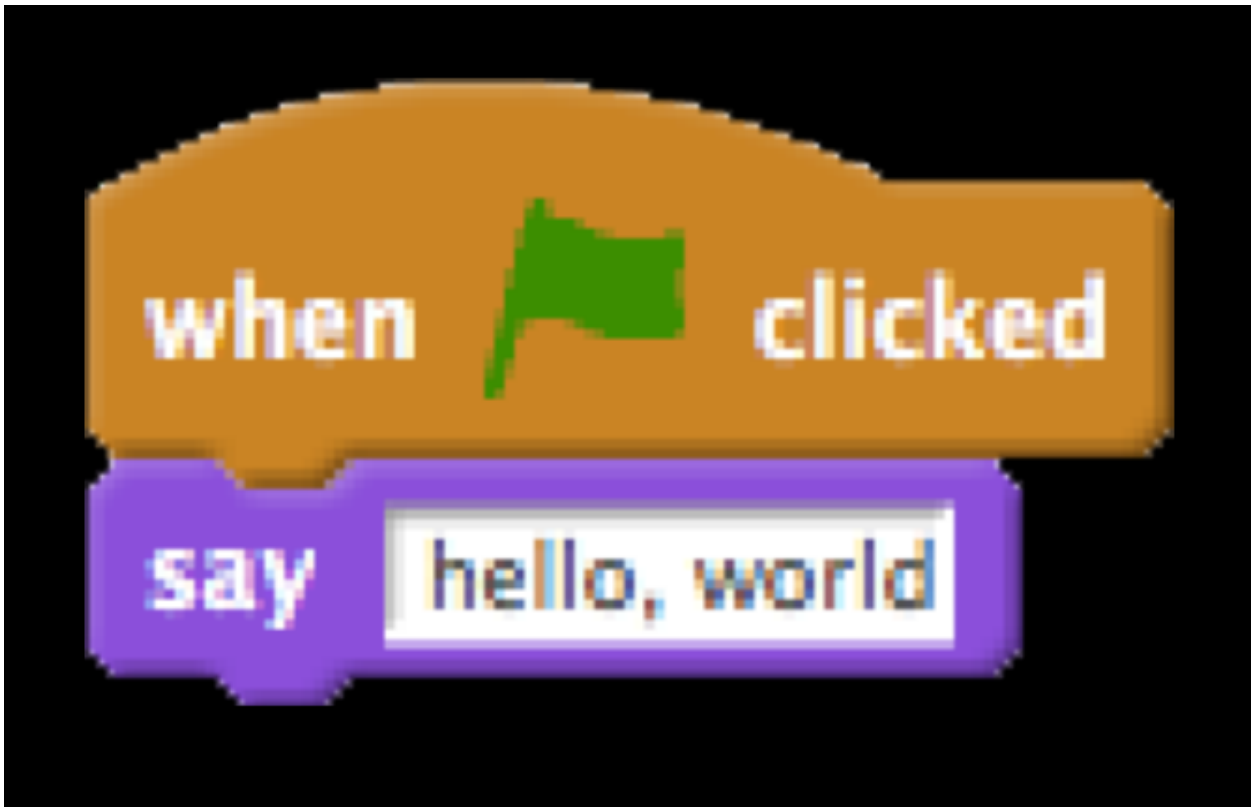
- CS50 students are supported by a team of over 100 staff members, a few of whom will say hello.
    - Doug Lloyd, who took CS50 12 years ago with no experience, has been on staff for 11 years now.
    - Maria Zlatkova also took CS50 three years ago as a freshman, and has been working with the course since.
    - Brian Yu too took CS50 his freshman year, and is the head course assistant.
    - Rob Bowden is a fourth-year PhD student, on his 8th year with the course.
- We share a short video about the community of CS50.

Scratch

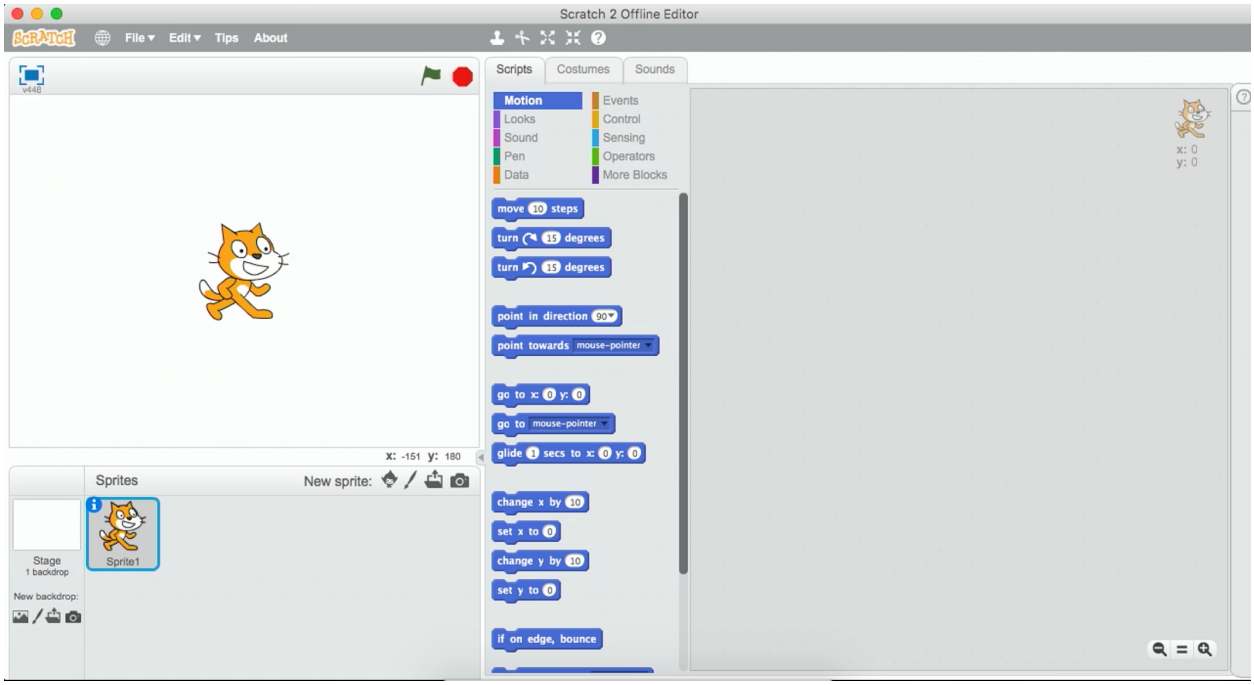
- Next week, we'll start looking at code that looks like this:

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

    - This code is in the language of C, and most of these keywords and syntax are unfamiliar to us.
    - We do see a `print` and `hello, world`, even if they are next to other unfamiliar pieces, so we might reasonably guess that this code "prints" the words "hello, world" onto the screen somehow.
- For now, we'll experiment with a simpler, graphical language, called Scratch, which allows us to drag-and-drop blocks.
- This Saturday will also be CS50's annual Puzzle Day, where teams will be solving puzzles with no computer science background needed.
- The Scratch program equivalent to the code above, for example, looks like this:

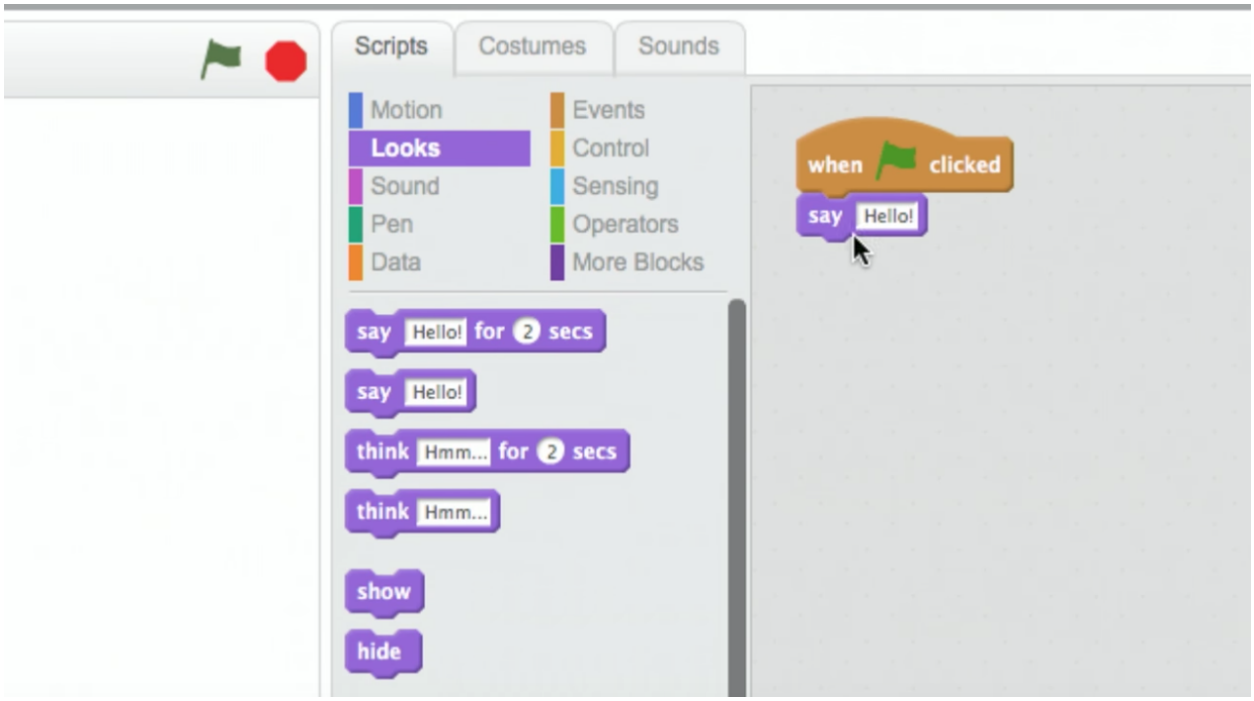


- We take a look at the Scratch editor:



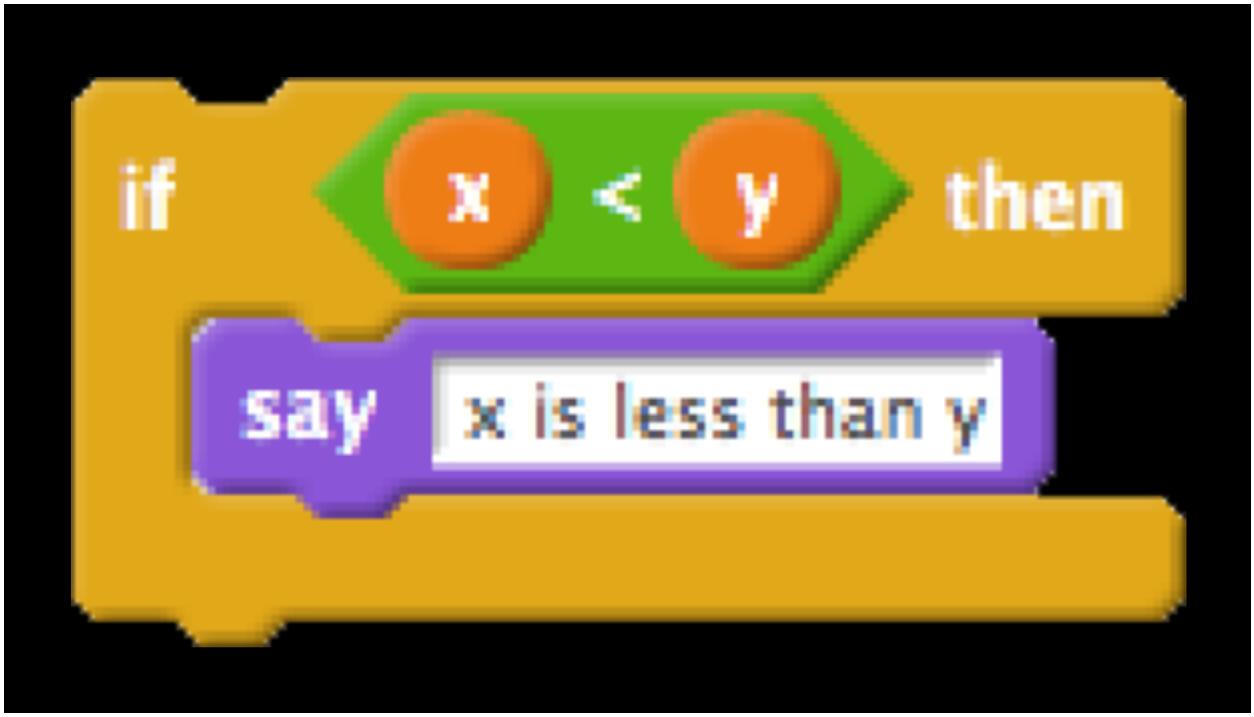
  - The box to the top left is the stage, or the area of the program we're working on, and right now it has a default character, Scratch the cat.
  - The bottom left has an area for us to add or create more characters, or sprites.
  - To the center is a toolbox of blocks we can choose from, in various categories.
  - And to the right is the script area, where we can drag and drop blocks in to do things.
- We notice that the stage has a green flag and a stop sign, so if we click around the categories of scripts, we'll notice that the Events section has a puzzle piece that looks like this:

- We can drag and drop these pieces to our script area:

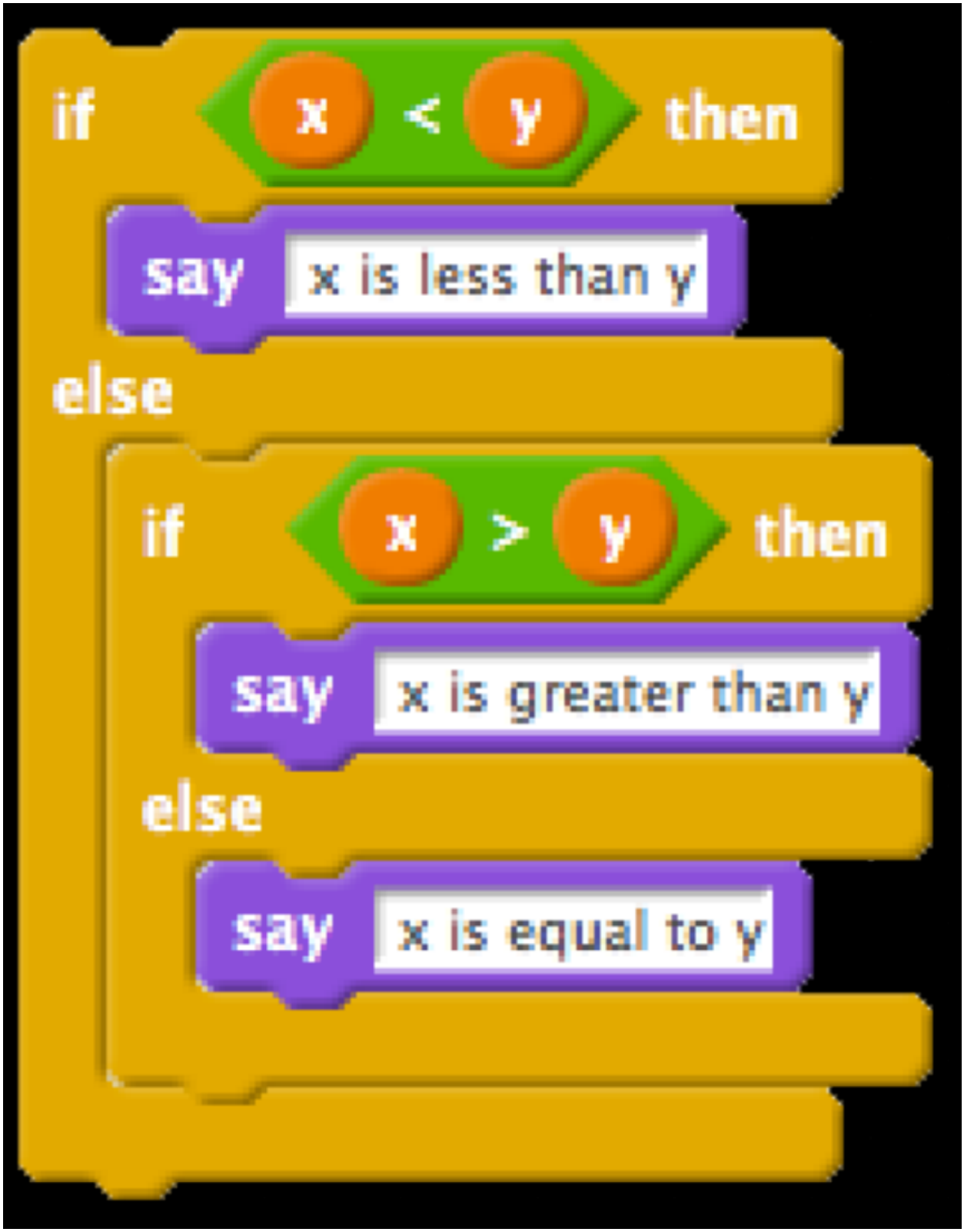


- Now if we click the green flag on the stage, we'll see the cat say our message.
- So functions like `say` in Scratch will be a purple puzzle piece:

  - These are like actions, or verbs, that do one thing.
- We also have conditions, where we have a branch that may or may not happen depending on the Boolean expression, in this case `x < y`, inside:



    - Notice that the action inside is wrapped inside the `if` block.
- We can nest more conditions inside:



    - We have a threeway fork now, where one of them will be true.
- We can compare variables to numbers in our Boolean expressions:

- We can also have blocks that repeat forever:

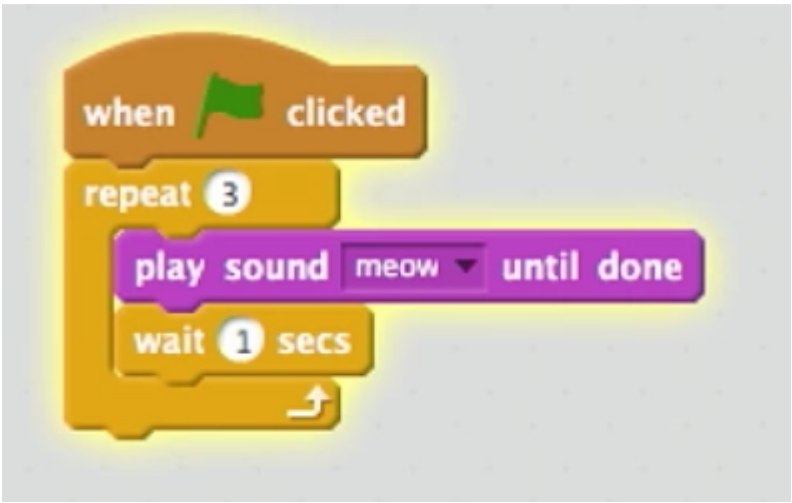
- ... or for a finite number of times:

    - Both of these are examples of loops.
- We can set variables to a certain value:

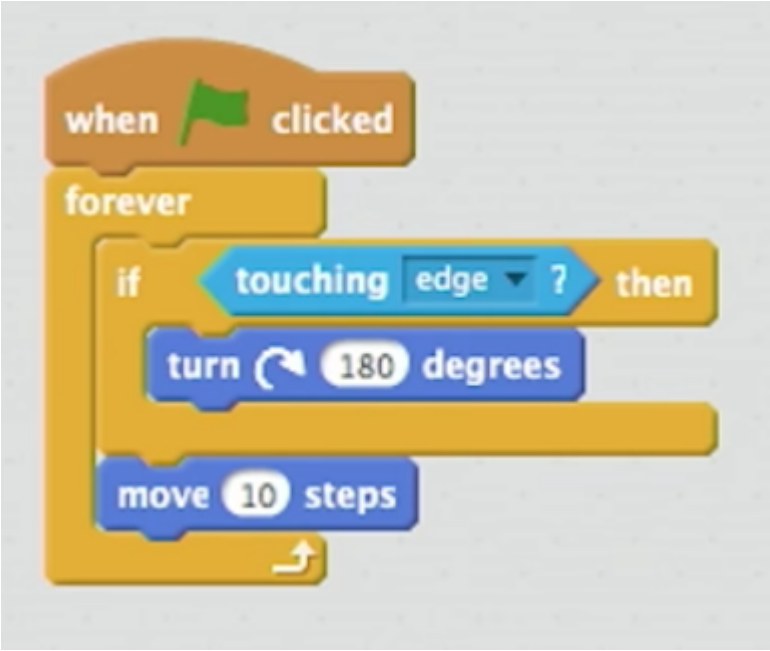
- And finally, we can have more than one of these:

    - Scratch, as well as other languages, support **multithreading**, or the ability for a computer program to do multiple things at once. Here, if we created two sets of scripts with a "when green flag clicked" block at the top of each, both will start running at the same time when we indeed click the green flag.
- We'll also see the concept of event handling, which essentially allows different piece of our program to communicate to one another:

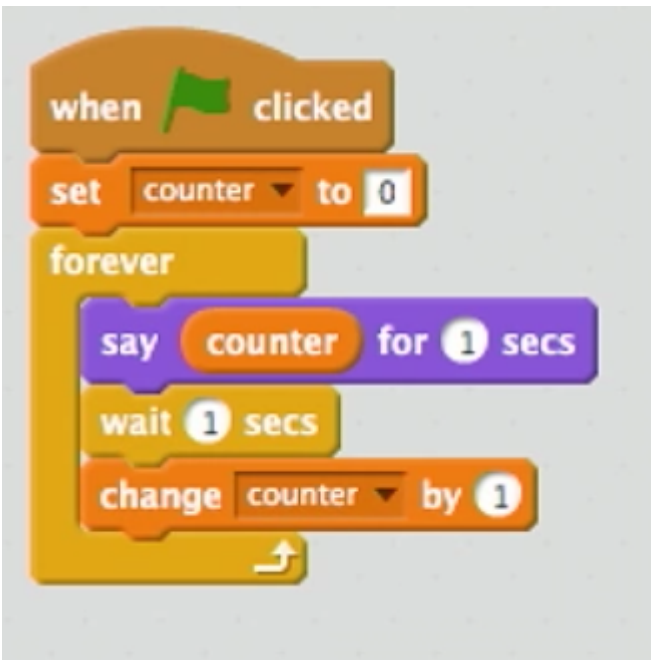
- Now let's start with some simple programs! We can drag the following blocks together, to have our cat make a "meow" sound three times:



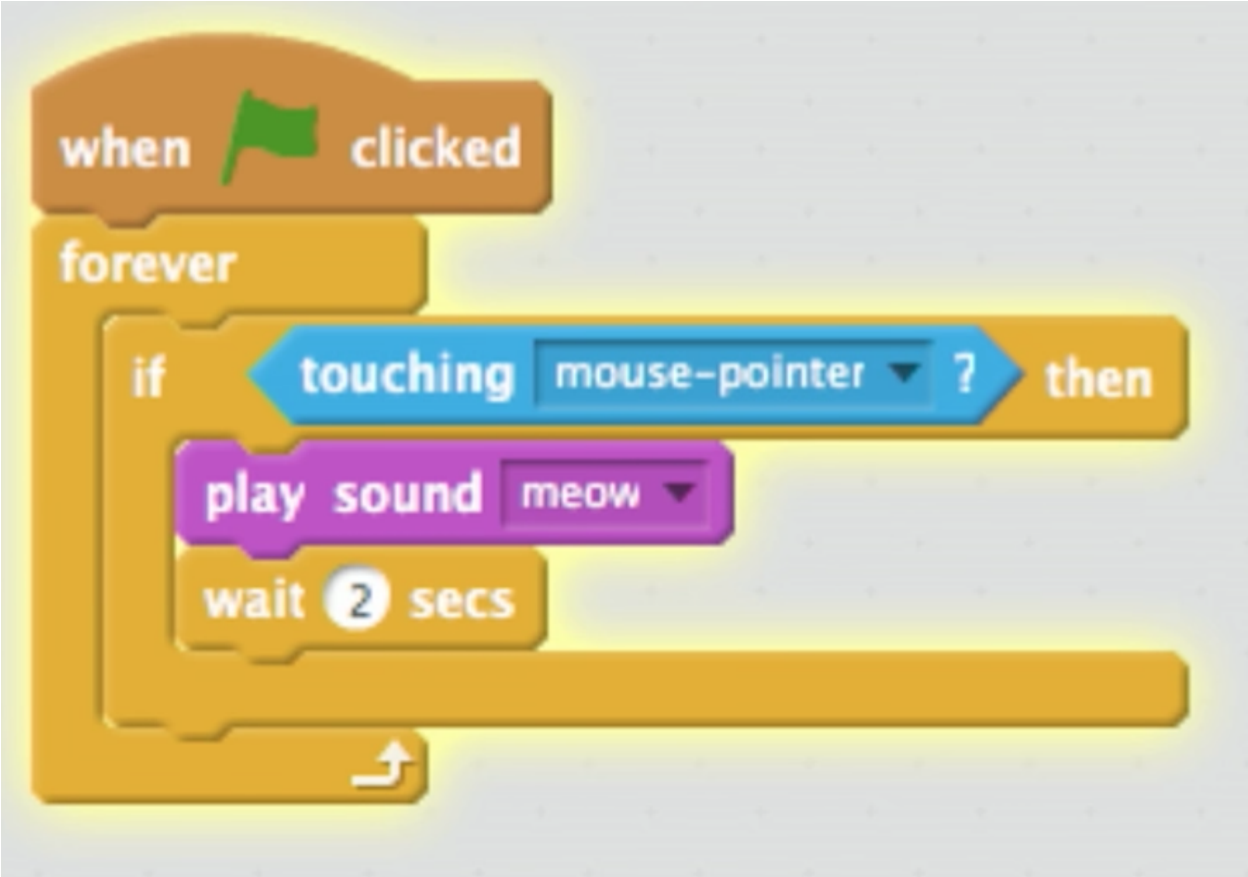
  - First, we tried the `play sound [meow]` block by itself, but we only heard "meow" once. That's because we played the sound, and immediately repeated that three times, so all three plays happened very quickly one after another. By using the `play sound [meow] until done` block, we can hear all three plays.
  - And adding `wait (1) secs` makes our cat sound a little more natural.
- We can drag the following blocks together, experimenting as we go along:



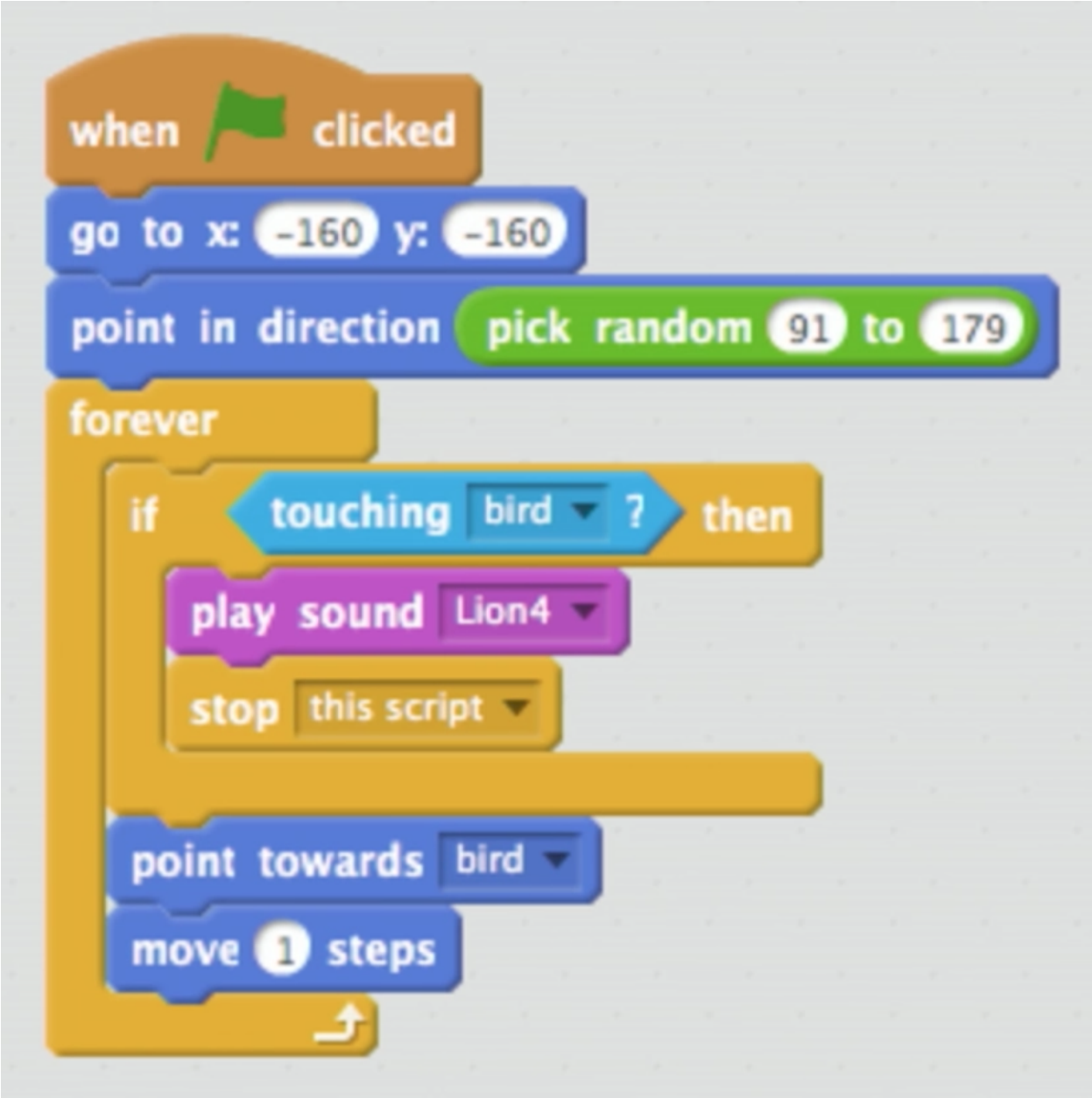
    - Now our cat will move back and forth on the stage forever!
- By putting together these small pieces, we can build more and more complicated programs.
- We demonstrate several Scratch projects, noting that for each one, perhaps a small part was implemented at a time.
- Scratch also supports setting variables to random integers, which helps us build games with more variety.
- We can look at how variables are used in this simple program, where a sheep on the stage starts counting from 0:



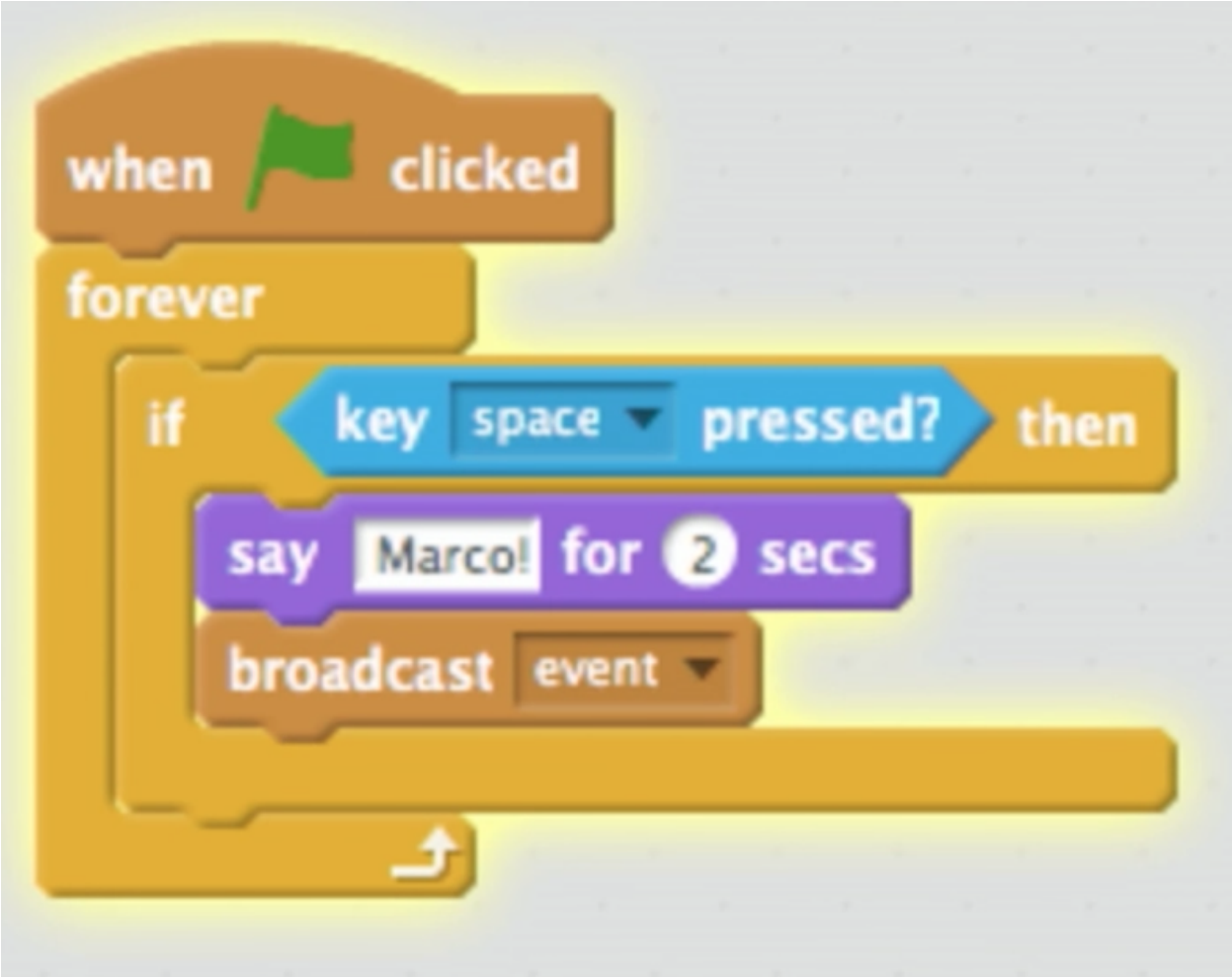
    - Here, `counter` is what we named a variable, with which we are storing the current number that the sheep is on.
- And we can use interactive Boolean expressions that automatically capture our input:



    - Now if we click the green flag and move our mouse pointer over the cat on the stage, it meows!
- We can also have interaction between two sprites, or characters, on the stage. Here we have a cat:

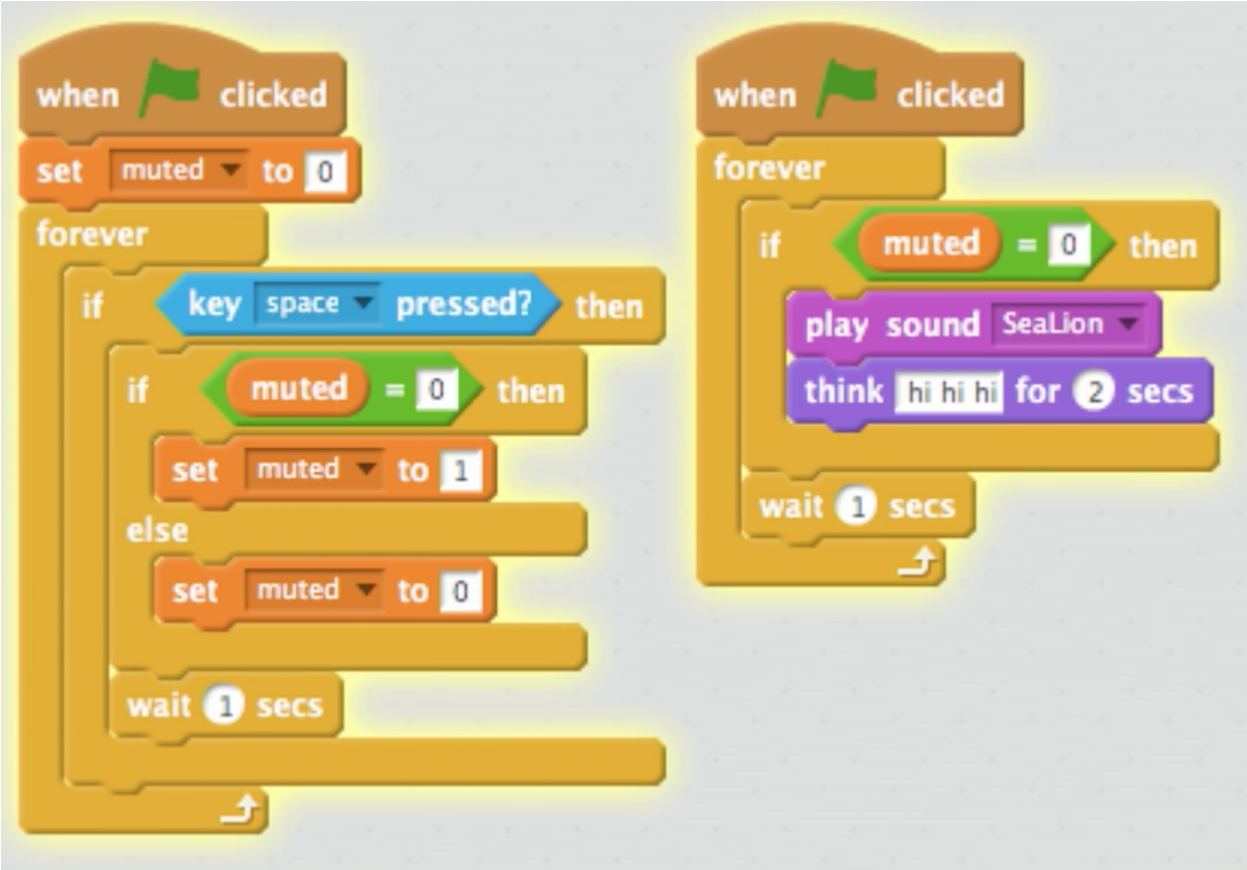


    - It starts by choosing a random direction, then turns toward the `bird` and moves toward it 1 step at a time.
    - The bird also has a similar script where it moves some number of steps at a time, bouncing from the edge as needed.
- Events, too, can be implemented with a few blocks:



- We see the `broadcast [event]` block for one of the sprites, and the "event" can be heard by other sprites like so:

- A single sprite, too, can have multiple threads that can share variables:



    - Here the `muted` variable is checked by the piece of the program that makes a sound, but it can be changed by the piece on the left that checks for whether the space key is pressed.
- We demonstrate one final example, Ivy's Hardest Game.