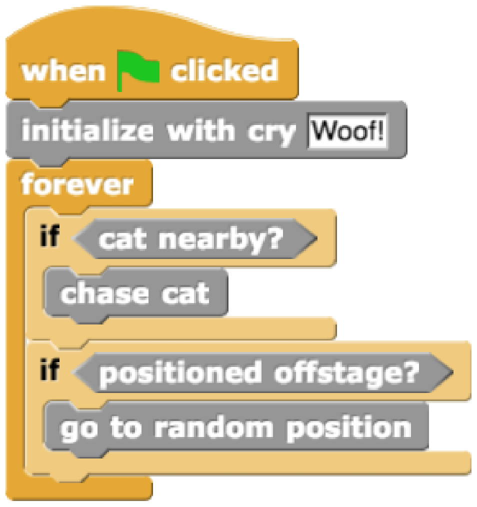
Teleport Chase: Custom Blocks Tutorial

Welcome to Unit 3: Custom Blocks. In this Unit you get to *build your own blocks*, a technique which gives you enormous power to make programs with complex behaviors, yet keep them organized and easily maintained.

In this tutorial, we'll be making a simple animation named "Teleport Chase" that involves a Mouse, a Cat, and a Dog. Here's a video of the finished product: https://www.youtube.com/watch?v=QcKvu0mB5-4 As you can see, the animals follow these simple behaviors:

- The Mouse runs in a straight line; if it runs offstage, it re-appears at a random position on stage.
- The Cat chases after the Mouse *unless* the Dog is nearby. Then it turns and flees from the Dog. The Cat also re-appears randomly if it runs offstage.
- The Dog has no interest in the Mouse, and is actually pretty lazy. It ignores the Cat and lies still unless the Cat gets too close. Then it chases the Cat. The Dog re-appears randomly if it runs offstage.
- If the Mouse gets caught or the Cat gets caught, it cries out, waits a moment, then teleports to someplace random.

The program we'll make uses 3 sprites, one for each animal. You can find a starter project with those sprites already set up at this link: Teleport Chase



```
when [flag] clicked
initialize with cry [Woof!]
forever
    if <cat nearby?>
        chase cat
    if <positioned offstage?>
        go to random position
```

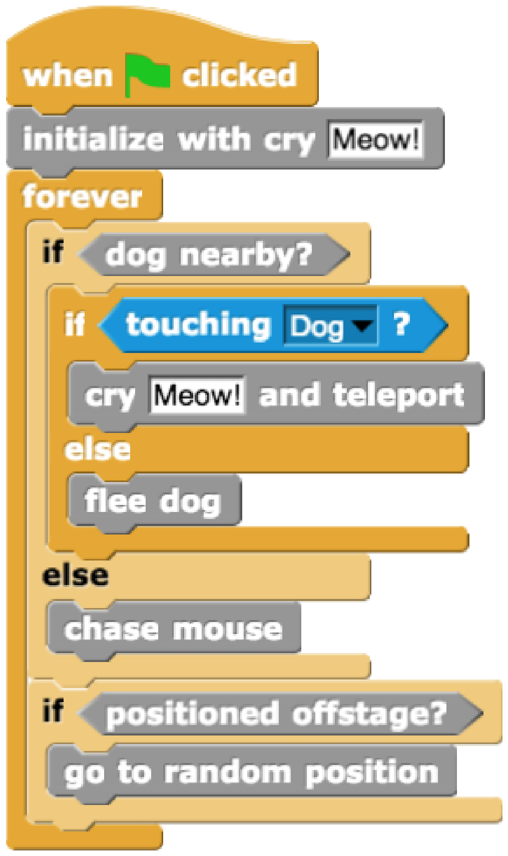
We're going to code up a single script for each animal. Here's the script we'll write for the Dog.

The single script for the Dog sprite.

As you can see, it maps pretty simply to the English statements "The dog says Woof!, then ignores the cat and lies still unless the cat gets too close. Then it chases the cat. The dog re-appears randomly if it runs offstage."

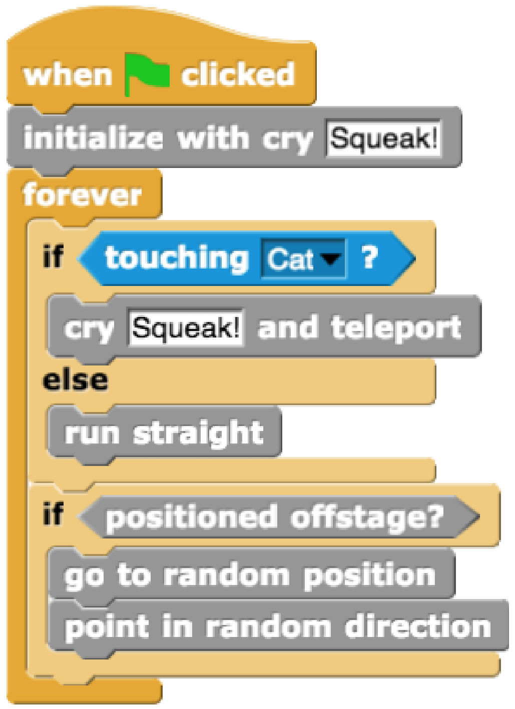
There's only one problem: which Snap! palette has the block `go to random position` ? You might look for it in the Motion palette, but this block is gray, which is a tip off that it's not one of the standard blocks. And certainly you won't find `chase cat` or `cat nearby?` in any of the standard palettes. What's going on here??

The answer is that these are *custom blocks,* blocks that you will make yourself and add to the set of blocks available. In a moment, we'll show you how to create these blocks and make them available for use. But first, here are the scripts for the Cat and Mouse sprites:



```
when 🏁 clicked
initialize with cry [Meow!]
forever
  if <dog nearby?>
    if <touching [Dog ▾] ?>
      cry [Meow!] and teleport
    else
      flee dog
  else
    chase mouse
  if <positioned offstage?>
    go to random position
```

The single script for the Cat sprite.



```
when 🏁 clicked
initialize with cry [Squeak!]
forever
  if <touching [Cat ▾] ?>
    cry [Squeak!] and teleport
  else
    run straight
  if <positioned offstage?>
    go to random position
    point in random direction
```

The single script for the Mouse sprite.

You'll be building these three as the last steps of the project. Before you can assemble them, you must create the component blocks that form them.

In total, you'll be making about a dozen custom blocks. Here's the set:

go to random position	point in random direction	point away from dog	positioned offstage?
awareness limit	cat nearby?	dog nearby?	gameSpeed
chase mouse	chase cat	flee dog	run straight
cry ▯ and teleport	initialize with cry ▯		

Let's build this project step-by-step. **Be sure to read each step fully before starting to do it!**

Step 0: Save a copy of the starter project

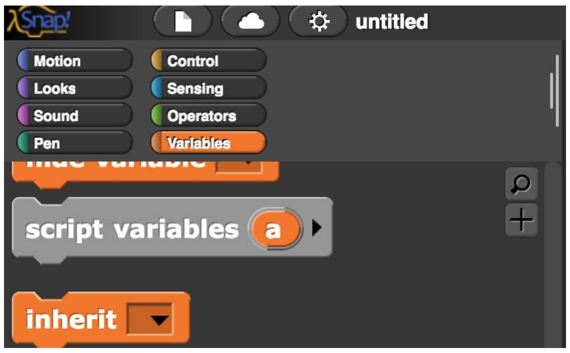
You can find a starter project with those sprites already set up at this link:Teleport Chase. Open it up, and then click "Save as…" to make your own copy to use during this tutorial. Name your copy *Block_3_1_Lastname_Firstname.*

Step 1: Make go to random position

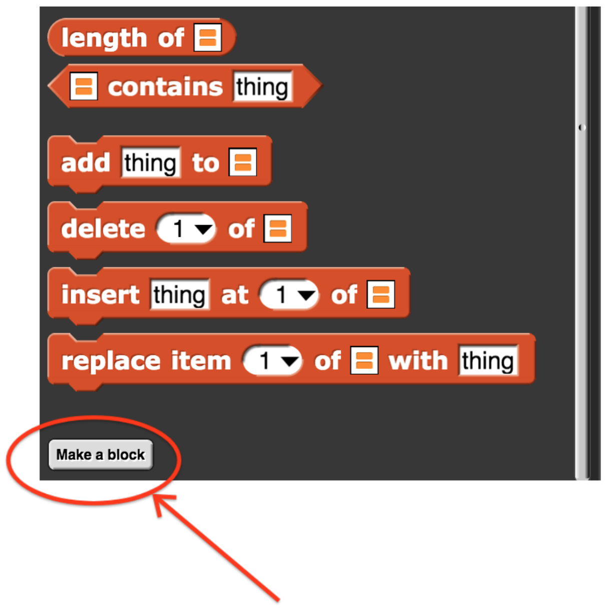
Our first block relocates a sprite to a random position. Recall that the stage runs from a left edge of x = -240 to a right edge of x = 240. Similarly, the top edge is y = 180 and the bottom edge is y = -180. So the recipe for going to a random position is simply:

```
go to x: (pick random (-240) to (240)) y: (pick random (-180) to (180))
```

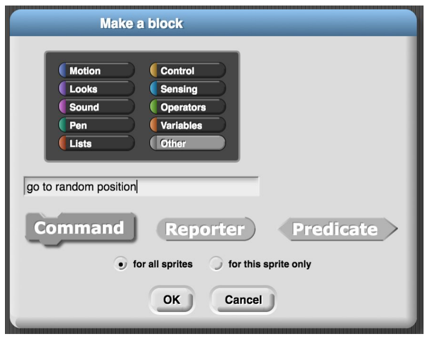
We are going to *encapsulate* this recipe into a custom block that we can use over and over again, with any sprite and in any situation where it would be helpful.
Start by selecting the "Variables" palette in left hand panel. Then scroll down to the bottom of the block set to find the button labeled "Make a block".



...

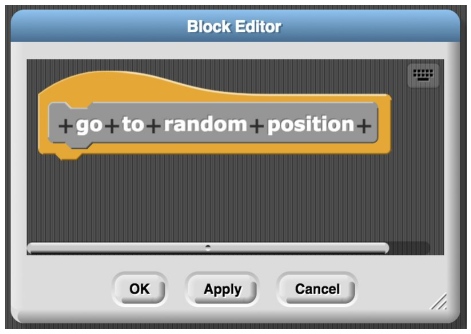


You'll get a dialog box titled "Make a block". Enter the block name "go to random position" and click OK. That will bring you to the "Block Editor" pane.

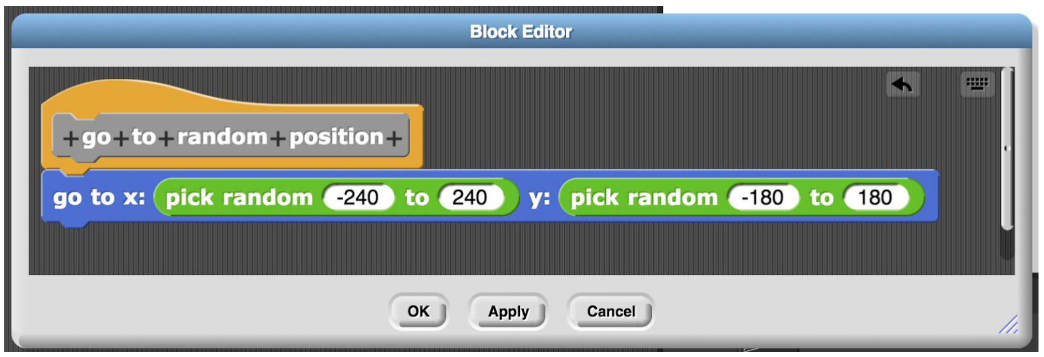


go to random position

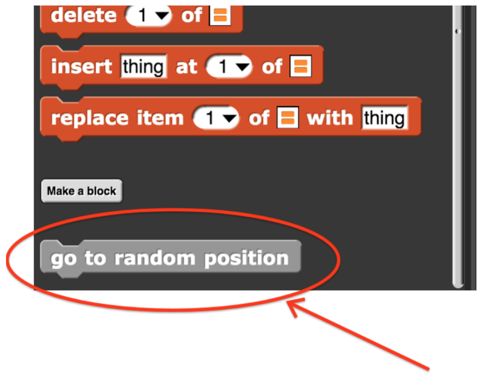
[✓ for all sprites] for this sprite only



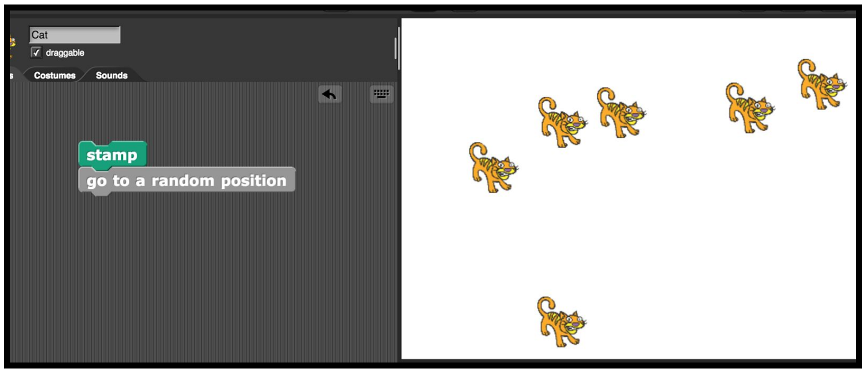
Now you can drag blocks from the various block palettes as usual, but instead of dropping them into the scripting area of an individual sprite, you drop them into the block editor pane and connect them to the hat-shaped orange block with "go to random position" in it. You can drag the lower right corner to enlarge the Block Editor window. Snap blocks together as usual to make the recipe shown then hit "OK".



After you hit "OK" to exit the Block Editor, return to the "Variables" block palette and scroll down to the bottom once more. You should find your new custom block waiting for you!



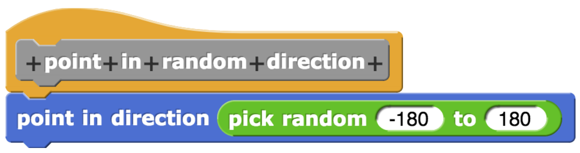
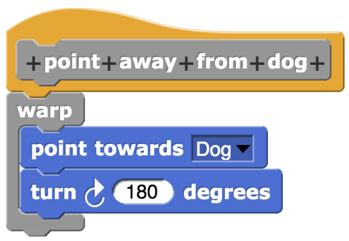
Go ahead and try it out. Drag the block into the scripting area and click it a few times. Your sprite should jump around the screen. You can combine it with the stamp block to leave an indication of where you've been:



Congratulations! You've made your first custom block!

Step 2: Make point in random direction and point away from dog

Try following the same process to create two more blocks in the block editor. Start each time with the "Make a block" button. Use these recipes:

```
+point+in+random+direction+
point in direction (pick random (-180) to (180))
```

```
+point+away+from+dog+
warp
    point towards Dog
    turn ↻ (180) degrees
```

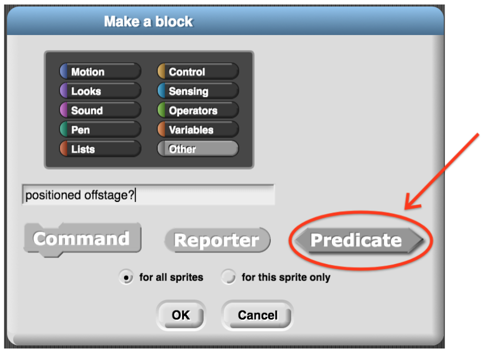
A few notes about point away from dog :

- Make sure you are working in the script area of the Cat or the Mouse when you make this block. Otherwise there won't be any Dog sprite listed in the drop-down menu for "point towards ___"
- The warp block is a standard component and can be found in the Control panel. It instructs Snap! to execute all of the blocks inside the warp before redrawing the screen again. This will make the animation smoother and avoid showing the sprite pointing toward the dog momentarily before turning away from it.

Step 3: Make positioned offstage?

So far, the three blocks you've made are Commands -- they cause a sprite to do something, and can be stacked vertically with other Commands. But you can make other kinds of custom blocks. Next, we're going to make a block that senses whether a sprite has gone off the stage. In our project, when an animal goes off the stage it "teleports" back on stage using our first custom block, go to random position . We're going to make a custom *boolean predicate* that tells when this is necessary because the sprite has gone offstage. A *predicate* is just a word for a boolean expression, something that evaluates to True or False. Most interesting predicates are statements *about something*, and that something determines if the result is True or False. For example "can drive?", "likes pizza?", and "is going trick-or-treating?" are all predicates about people -- the answers will be different for each person.

To make a custom boolean expression, we need to take one more action when creating the block. When you are in the "Make a block" dialog window, press the hexagon-shaped "Predicate" button before pressing "OK". Try it now as part of making positioned offstage? :

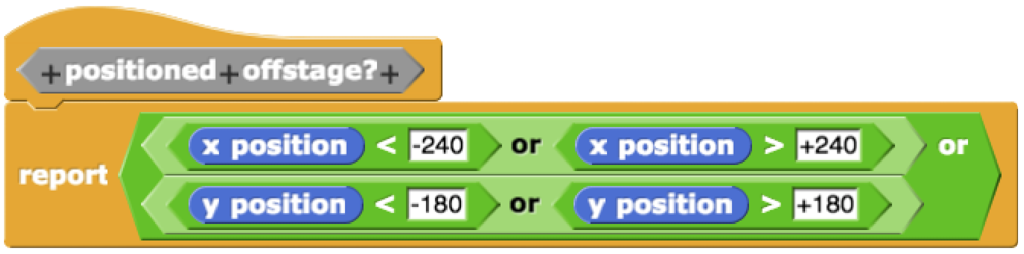


positioned offstage?

It's a little subtle because all of the block shapes on that line are already gray, so figuring out which one you've selected can be tough. Try clicking Command, Reporter, and Predicate to see how the gray deepens when you select one.

Note that it's customary to end a Predicate block with a question mark.

Once you have named your block and selected Predicate, click OK as usual. Define positioned offstage? in the block editor using this recipe (recall ±180 and ±240 are the dimensions of the stage, so if the x position is too negative the sprite is offstage left, x position too big it's offstage right, y position too negative it's offstage below the bottom, y position to positive it's offstage above the top):



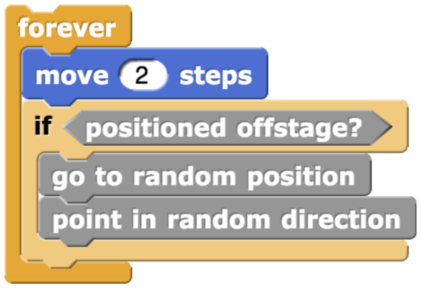
```
+positioned+offstage?+
report < < (x position) < -240 > or < (x position) > +240 > > or < < (y position) < -180 > or < (y position) > +180 > >
```

This recipe uses a new standard component we haven't seen before, the report [ ] block. It is located in the Control palette. The *report* block reports a value from a custom block to its caller.

In effect, it sets the value that the predicate has in the current call. Notice that report [ ] has no tab on its bottom -- you can't connect anything below it. The reason is that the stack of blocks within the custom block halts its execution when it reaches a report block. The value is passed back to the code that was built using the custom block.

Note that our positioned offstage? block doesn't actually *do* anything. There is no motion, no changes to variables, no Commands. This is often desirable. Predicates can be convenient ways of computing a value and encapsulating away from the rest of the code the details of how that computation is done.

Now that you've finished creating the positioned offstage? block, we can use three of the custom blocks we've made in this short script with any of the animal sprites:



```
forever
  move (2) steps
  if <positioned offstage?>
    go to random position
    point in random direction
```

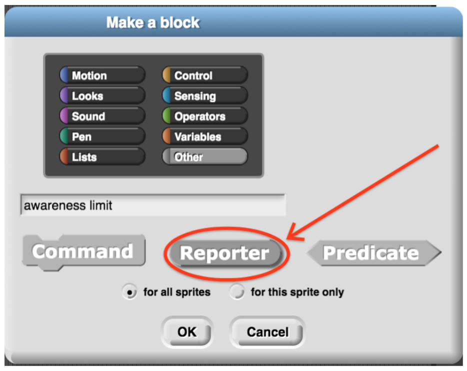
Can you predict what this stack does? Click the stack to make it run. It does just what it says on the tin! The sprite moves forward, and when it exits off stage it goes to a new random position and points in a random direction. This is the power of writing custom blocks to encapsulate the logic -- your programs become so easy to read, you can barely tell they are programs and not simply english.

Notice that positioned offstage? is evaluated each time through the loop. Each time the code in the custom block runs with the current value of x position and y position. Most of the time the boolean expression inside the custom block evaluates to False, so the random repositioning does not happen. When x or y gets too extreme, the boolean expression inside the custom block evaluates to True, and the program does execute the teleport.

Step 4: Make the Reporter awareness limit, and Predicates cat nearby? and dog nearby?.

We've seen how to make custom Commands and custom Predicates. Now we'll make a Reporter block, the last of the three kinds of custom blocks available in Snap! The block we're going to make specifies how close the dog and the cat need to get before they become aware of each other. We call this the "awareness limit".

This time when you get to the "Make a Block" dialog window, select "Reporter" as the block type:



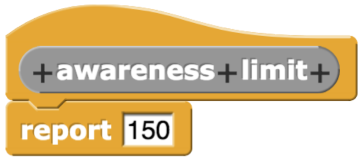
awareness limit

Command [✓ Reporter] Predicate

[✓ for all sprites] for this sprite only

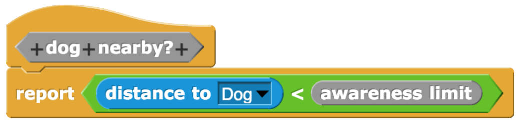
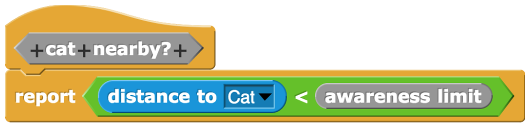
There's only one difference between a Reporter and a Predicate: they both use the report block to pass a value back to the calling code, but Reporters can pass back anything: a number, some text, a list, or a boolean. Predicates can only report out booleans -- they are just a special kind of reporter.

Create a Reporter named awareness limit with the following recipe:



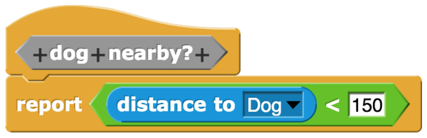
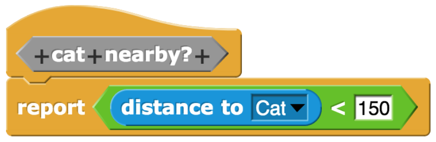
```
+awareness+limit+
report 150
```

and use it to make two new Predicates cat nearby? and dog nearby? :

 `+dog+nearby?+` `report (distance to Dog) < (awareness limit)`	 `+cat+nearby?+` `report (distance to Cat) < (awareness limit)`

Note that the distance to ▾ block is a standard component, found in the Sensing palette.

At first glance the awareness limit definition looks silly. Why have a custom block that always evaluates to 150? Wouldn't it be just as easy to write

 `+dog+nearby?+` `report (distance to Dog) < 150`	 `+cat+nearby?+` `report (distance to Cat) < 150`

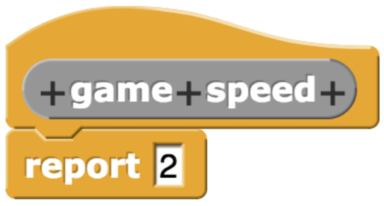
There are two reasons for using awareness limit instead of 150. The first is that it is more meaningful. When you read the code "distance to Dog < awareness limit", you know that the programmer was thinking about what the sprite knows and when it knows it. When you just see the bare number 150, you are guessing at what 150 means. Something about location? Edge? points? You have to work to figure it out. Using named Reporters helps remove the work.

The second reason for using awareness limit instead of 150 is easier maintenance. What if you write the program with 150 as the limit and later decide it's no good -- you want to make the limit 100 instead. If you have created a custom Reporter then you only need to make your change in one place -- the definition -- rather than try to remember all the places that number is used and fix each of them. You're likely to overlook one, or accidentally change a 150 to a 100 in a place where it wasn't actually the awareness limit, it was just coincidentally the same value 150 used in a different context.

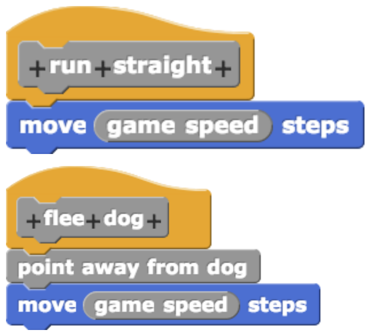
You might wonder why use a custom Reporter for this instead of defining a variable awareness limit and then using set awareness limit ▾ to 150 . This is certainly possible and brings all of the benefits described as reasons to use a Reporter. The only difference is that with the Reporter, you know the value of awareness limit won't ever change, while with the variable approach it can. Which one is the better path depends on whether you think the value ever should change during the running of the program. It's common for programmers to start with one of these approaches and later decide to switch to the other. It's usually not much work at all to make that change.

Step 5: Make the blocks gameSpeed, run straight, flee dog, chase mouse, and chase cat

Create these blocks (one Reporter, four Commands) using the Block Editor. Be sure to select the Dog sprite first so that when you choose a "touching ___" or a "point towards ___" block, the drop-down menu contains both "Mouse" and "Cat" as choices. Remember to select the correct shape of custom block in the "Make a Block" dialog window.

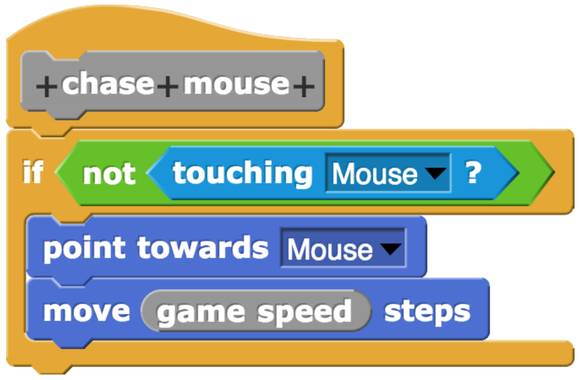


```
+game+speed+
report 2
```

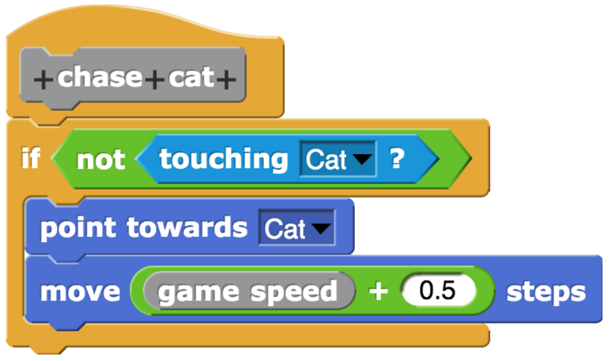


```
+run+straight+
move (game speed) steps

+flee+dog+
point away from dog
move (game speed) steps
```



```
+chase+mouse+
if <not <touching Mouse ?>>
    point towards Mouse
    move (game speed) steps
```



```
+chase+cat+
if <not <touching Cat ?>>
    point towards Cat
    move ((game speed) + 0.5) steps
```

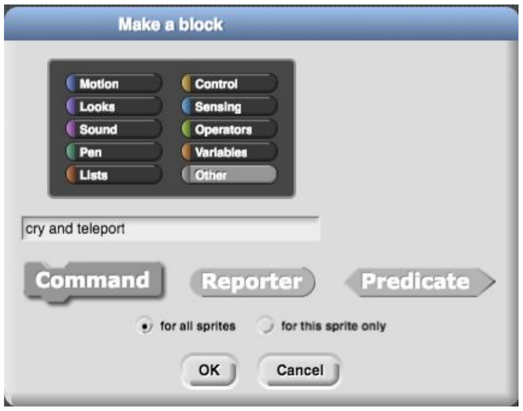
Notice that an animal chasing the Cat (presumably the Dog) runs slightly faster than an animal chasing the Mouse (presumably the Cat). This is just in place so that the Dog has a chance of catching the Cat when the Dog becomes aware of the Cat and starts chasing it.

You might wonder why bother to make a block for run straight when it's definition is just one block. The answer is that it makes the overall program more readable -- it's like a comment and a block all in one.

Step 6: Make the block cry ▯ and teleport, a Command with an input.

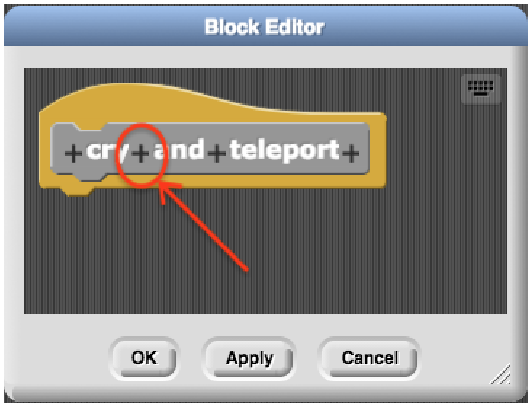
Some of the standard Snap! blocks (for example say ▯, move ● steps, or ▯ < ▯) contain one or more blank slots where a programmer can provide their own input. The inputs that the programmer provides are called *arguments*. Blocks that accept input arguments can be used flexibly, especially when other blocks are provided as the arguments. Let's see how to make a custom block that includes a blank space for an input.

Start by clicking the "Make a Block" button. In the dialog window that opens, name the block "cry and teleport", select "Command", and click "OK" to reach the Block Editor window. Now you are going to do something new: click the "+" sign just after the word "cry":

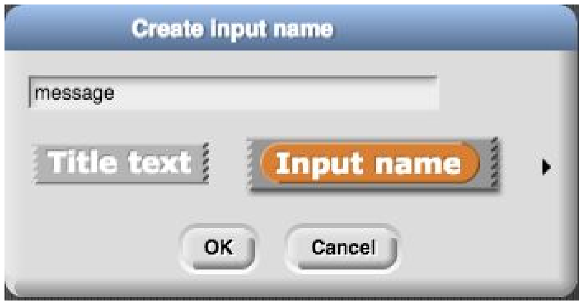


cry and teleport

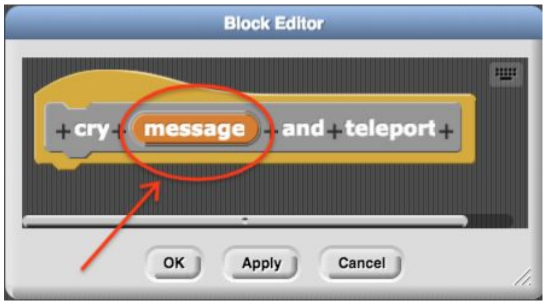
[● for all sprites] for this sprite only



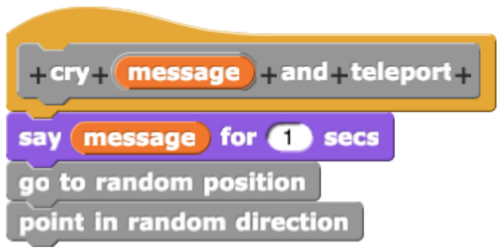
This will open a new dialog window entitled "Create Input name". Enter the name "message" and click OK. This will return you to the block editor, where you will see a variable named "message" has appeared in the name of your custom block:



message



Working in the Block Editor, you can grab the message block and drop it anywhere inside the editor, including inside another block. Use this technique to construct this definition for cry ▯ and teleport:

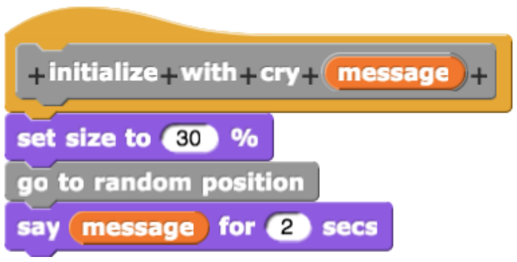


```
+cry+ (message) +and+teleport+
say (message) for (1) secs
go to random position
point in random direction
```

After clicking "OK" to finish creating the cry [ ] and teleport block for, confirm that it appears in the "Variables" palette. Drag it onto the scripting area and try it out with different values in the input: cry [Yikes!] and teleport , cry [Look out!] and teleport , cry [Help!] and teleport .
Click on each one and confirm that changing the input argument you provide does change the sprite's behavior.

Step 7: Make the initialize with cry [ ] block

Practice what you learned in Step 6 to make another command block that takes an input. One nice thing is that you can call the input argument "message" in this block too -- the variables that hold the argument values for one custom block don't interfere with the variables that hold the argument values of a different custom block. Create a new command named "initialize with cry", add an input after the cry, and give the block this definition:

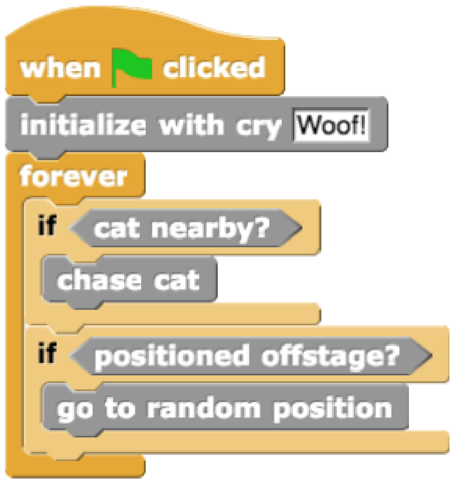


```
+initialize+with+cry+ (message) +
set size to (30) %
go to random position
say (message) for (2) secs
```

Step 8: Write top-level scripts for the sprites.

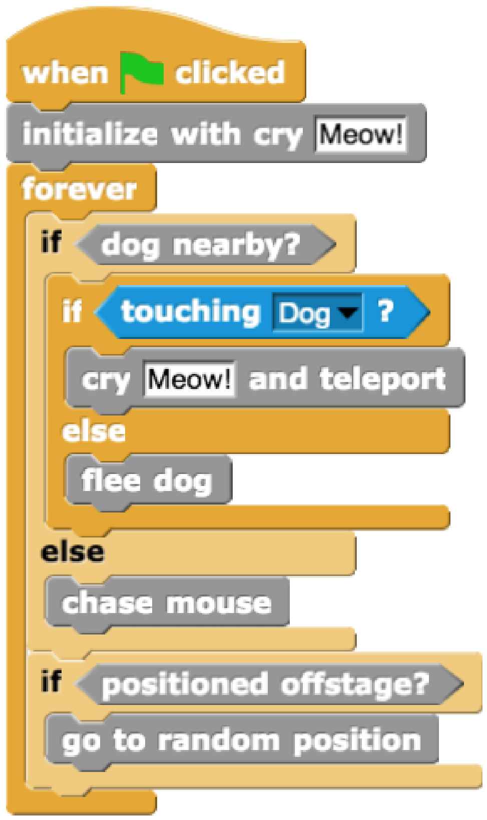
At this point, you've written a lot of code. However, your script areas are virtually empty! All the code you wrote is inside the custom blocks. You can see all your custom blocks by going to the "Variables" palette and scrolling to the bottom. Right-click / Control-click on any of the blocks and select "edit…" to review how each block was made. You can change or rename the blocks from the Block Editor at any time.

In this final step, we're going to use all our custom blocks to build the program. These are not themselves custom blocks, but ordinary scripts like the ones you've been making in Units 1 and 2 of this course. The scripts you should write already appeared at the start of this tutorial, but here they are again:



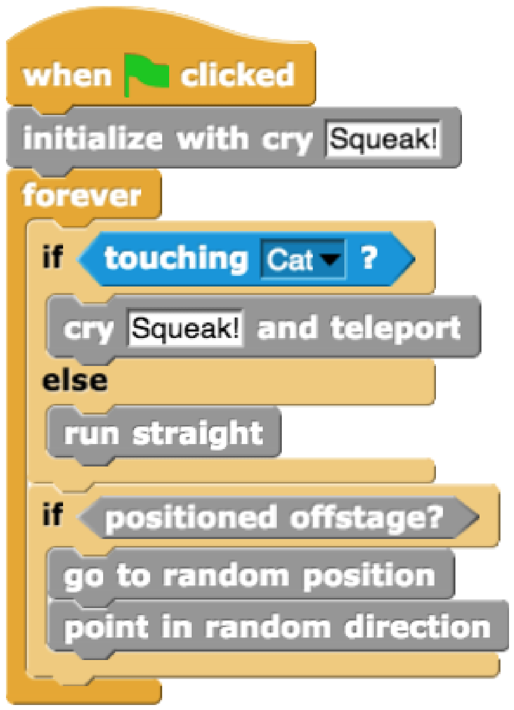
```
when [flag] clicked
initialize with cry [Woof!]
forever
  if <cat nearby?>
    chase cat
  if <positioned offstage?>
    go to random position
```

The single script for the Dog sprite.



```
when [flag] clicked
initialize with cry [Meow!]
forever
  if <dog nearby?>
    if <touching [Dog] ?>
      cry [Meow!] and teleport
    else
      flee dog
  else
    chase mouse
  if <positioned offstage?>
    go to random position
```

The single script for the Cat sprite.



```
when [flag] clicked
initialize with cry [Squeak!]
forever
  if <touching [Cat] ?>
    cry [Squeak!] and teleport
  else
    run straight
  if <positioned offstage?>
    go to random position
    point in random direction
```

The single script for the Mouse sprite.

That's it! Click on the green flag and watch the animals go. (Be sure to save your work!)