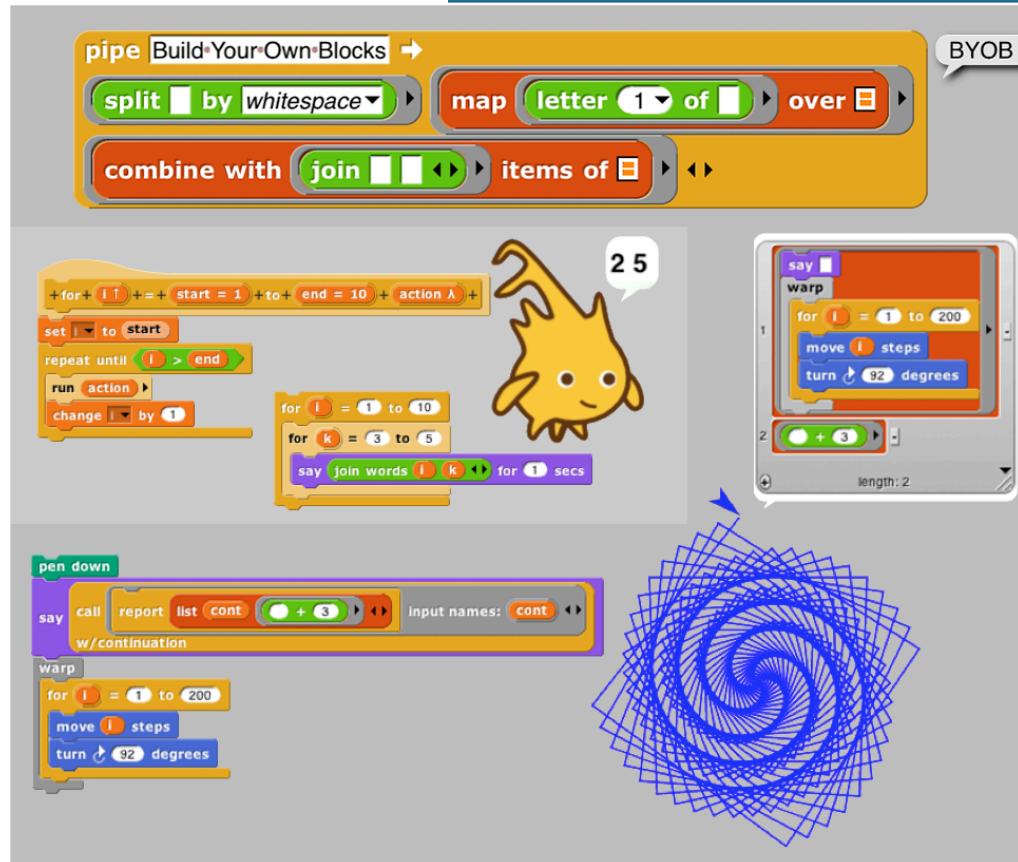




Build Your Own Blocks

5.0

SNAP! Reference Manual



Brian Harvey

Jens Mönig

Table of Contents

I. Blocks, Scripts, and Sprites	4
Hat Blocks and Command Blocks	5
A. <i>Sprites and Parallelism</i>	7
Costumes and Sounds	7
Inter-Sprite Communication with Broadcast	8
B. <i>Nesting Sprites: Anchors and Parts</i>	9
C. <i>Reporter Blocks and Expressions</i>	9
D. <i>Predicates and Conditional Evaluation</i>	11
E. <i>Variables</i>	12
Global Variables	13
Script Variables	14
Renaming variables	14
Transient variables	15
F. <i>Debugging</i>	16
The pause button	16
Breakpoints: the <code>pause all</code> block	16
Visible stepping	17
G. <i>Etcetera</i>	18
II. Saving and Loading Projects and Media.....	22
A. <i>Local Storage</i>	22
B. <i>Creating a Cloud Account</i>	22
C. <i>Saving to the Cloud</i>	23
D. <i>Loading Saved Projects</i>	23
E. <i>Private and Public Projects</i>	24
III. Building a Block.....	25
A. <i>Simple Blocks</i>	25
Custom Blocks with Inputs	27
Editing Block Properties	28
B. <i>Recursion</i>	28
C. <i>Block Libraries</i>	29
IV. First Class Lists	31
A. <i>The <code>list</code> Block</i>	31
B. <i>Lists of Lists</i>	32
C. <i>Functional and Imperative List Programming</i>	33
D. <i>Higher Order List Operations and Rings</i>	34
E. <i>Table View vs. List View</i>	36
Comma-Separated Values	39
Multi-dimensional lists and JSON	39
V. Typed Inputs	40
A. <i>Scratch's Type Notation</i>	40
B. <i>The Snap! Input Type Dialog</i>	40
Procedure Types	41
Pulldown inputs	42
Input variants	44
Prototype Hints	45
Title Text and Symbols	45
VI. Procedures as Data.....	46
A. <i>Call and Run</i>	46
<code>Call/Run</code> with inputs	46
Variables in Ring Slots	47
B. <i>Writing Higher Order Procedures</i>	47
Recursive Calls to Multiple-Input Blocks	49
C. <i>Formal Parameters</i>	50
D. <i>Procedures as Data</i>	51
E. <i>Special Forms</i>	52
Special Forms in Scratch	53
VII. Object Oriented Programming with Sprites	54
A. <i>First Class Sprites</i>	54
B. <i>Permanent and Temporary Clones</i>	55
C. <i>Sending Messages to Sprites</i>	55
Polymorphism	56
D. <i>Local State in Sprites: Variables and Attributes</i>	57
E. <i>Prototyping: Parents and Children</i>	57
F. <i>Inheritance by Delegation</i>	58
G. <i>List of attributes</i>	59
H. <i>First Class Costumes and Sounds</i>	60
Media Computation with Costumes	60
Media Computation with Sounds	63
VIII. OOP with Procedures	66
A. <i>Local State with Script Variables</i>	66
B. <i>Messages and Dispatch Procedures</i>	67
C. <i>Inheritance via Delegation</i>	68
D. <i>An Implementation of Prototyping OOP</i>	69
IX. The Outside World	72
A. <i>The World Wide Web</i>	72
B. <i>Hardware Devices</i>	73
C. <i>Date and Time</i>	73
X. Continuations.....	74
A. <i>Continuation Passing Style</i>	75
B. <i>Call/Run w/Continuation</i>	78
Nonlocal exit	80
XI. User Interface Elements	82
A. <i>Tool Bar Features</i>	82
The Snap!Logo Menu	82
The File Menu	83
The Cloud Menu	96
The Settings Menu	97
Visible Stepping Controls	100
Stage Resizing Buttons	100
Project Control Buttons	100
B. <i>The Palette Area</i>	101
Buttons in the Palette	101
Context Menus for Palette Blocks	101
Context Menu for the Palette Background	102
Palette Resizing	102
C. <i>The Scripting Area</i>	103
Sprite Appearance and Behavior Controls	103
Scripting Area Tabs	103
Scripts and Blocks Within Scripts	103
Scripting Area Background Context Menu	105
Controls in the Costumes Tab	106
The Paint Editor	108
Controls in the Sounds Tab	109
D. <i>Keyboard Editing</i>	110
Starting and stopping the keyboard editor	110
Navigating in the keyboard editor	110
Editing a script	111
Running the selected script	112
E. <i>Controls on the Stage</i>	112
F. <i>The Sprite Corral and Sprite Creation Buttons</i>	114
G. <i>Preloading a Project when Starting Snap!</i>	114
H. <i>Mirror Sites</i>	115
Index	115

Acknowledgements

We have been extremely lucky in our mentors. Jens cut his teeth in the company of the Smalltalk pioneers: Alan Kay, Dan Ingalls, and the rest of the gang who invented personal computing and object oriented programming in the great days of Xerox PARC. He worked with John Maloney, of the MIT Scratch Team, who developed the Morphic graphics framework that's still at the heart of **Snap!**.

The brilliant design of Scratch, from the Lifelong Kindergarten Group at the MIT Media Lab, is crucial to Snap!. Our earlier version, BYOB, was a direct modification of the Scratch source code. Snap! is a complete rewrite, but its code structure and its user interface remain deeply indebted to Scratch. And the Scratch Team, who could have seen us as rivals, have been entirely supportive and welcoming to us.

Brian grew up at the MIT and Stanford Artificial Intelligence Labs, learning from Lisp inventor John McCarthy, Scheme inventors Gerald J. Sussman and Guy Steele, and the authors of the world's best computer science book, *Structure and Interpretation of Computer Programs*, Hal Abelson and Gerald J. Sussman with Julie Sussman, among many other heroes of computer science.

In the glory days of the MIT Logo Lab, we used to say, “Logo is Lisp disguised as BASIC.” Now, with its first class procedures, lexical scope, and first class continuations, Snap! is Scheme disguised as Scratch.

Two people have made such massive contributions to the implementation of **Snap!** that we have officially declared them members of the team: Michael Ball and Bernat Romagosa. Other close collaborators include Joan Guillén i Pelegay, who has contributed very careful and wise analysis of outstanding issues, including help in taming the management of translations to non-English languages, and Jadga Huegle, a colleague of Jens at SAP, who has energetically contributed to online mini-courses about **Snap!** and leading workshops for kids and for adults.

We have been fortunate to get to know an amazing group of brilliant middle school(!) and high school students through the Scratch Advanced Topics forum, several of whom have contributed code to **Snap!**: Kartik Chandra, Nathan Dinsmore, Connor Hudson, Ian Reynolds, and Dylan Servilla. Many more have contributed ideas and alpha-testing bug reports. UC Berkeley students who've contributed code include Michael Ball, Achal Dave, Kyle Hotchkiss, Ivan Motyashov, and Yuan Yuan. Contributors of translations are too numerous to list here, but they're in the "About..." box in **Snap!** itself.

This material is based upon work supported in part by the National Science Foundation under Grants No. 1138596, 1143566, and 1441075; and in part by MioSoft, Arduino.org, SAP, and YC Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funders.

Snap! Reference Manual

Version 5.0

Snap! (formerly BYOB) is an extended reimplementation of Scratch (<http://scratch.mit.edu>) that allows you to Build Your Own Blocks. It also features first class lists, first class procedures, first class sprites, first class costumes, first class sounds, and first class continuations. These added capabilities make it suitable for a serious introduction to computer science for high school or college students.

In this manual we sometimes make reference to Scratch, e.g., to explain how some **Snap!** feature extends something familiar in Scratch. It's very helpful to have some experience with Scratch before reading this manual, but not essential.

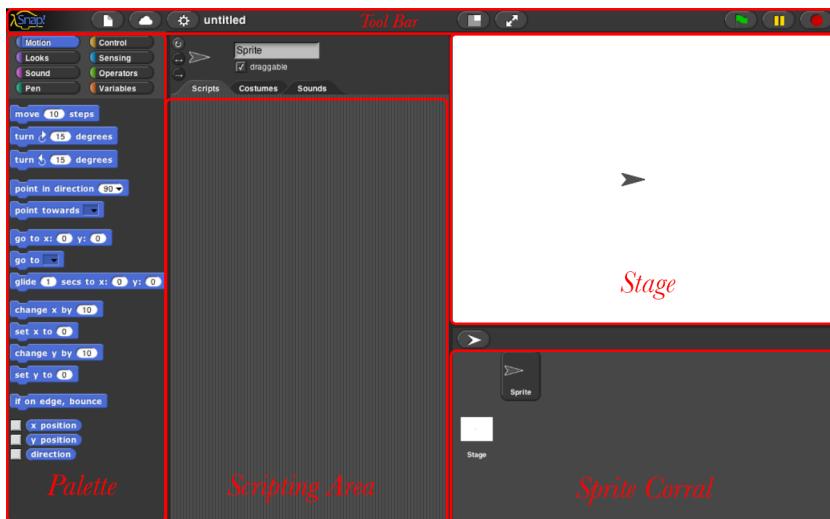
To run **Snap!**, open a browser window and connect to <http://snap.berkeley.edu/run>. The **Snap!** community web site at <http://snap.berkeley.edu> is not part of this manual's scope.

I. Blocks, Scripts, and Sprites

This section describes the **Snap!** features inherited from Scratch; experienced Scratch users can skip to subsection B.

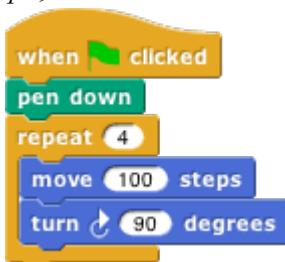
Snap! is a programming language — a notation in which you can tell a computer what you want it to do. Unlike most programming languages, though, **Snap!** is a *visual* language; instead of writing a program using the keyboard, the **Snap!** programmer uses the same drag-and-drop interface familiar to computer users.

Start **Snap!**. You should see the following arrangement of regions in the window:



(The proportions of these areas may be different, depending on the size and shape of your browser window.)

A **Snap!** program consists of one or more *scripts*, each of which is made of *blocks*. Here's a typical script:



The five blocks that make up this script have three different colors, corresponding to three of the eight *palettes* in which blocks can be found. The palette area at the left edge of the window shows one palette at a time, chosen with the eight buttons just above the palette area. In this script, the gold blocks are from the Control palette; the green block is from the Pen palette; and the blue blocks are from the Motion palette. A script is assembled by dragging blocks from a palette into the *scripting area* in the middle part of the window. Blocks snap together (hence the name **Snap!** for the language) when you drag a block so that its indentation is near the tab of the one above it:



The white horizontal line is a signal that if you let go of the green block it will snap into the tab of the gold one.

Hat Blocks and Command Blocks

At the top of the script is a *hat* block, which indicates when the script should be carried out. Hat block names typically start with the word “**when**”; in the square-drawing example on page 4, the script should be run when the green flag near the right end of the **Snap!** tool bar is clicked. (The **Snap!** tool bar is part of the **Snap!** window, not the same as the browser’s or operating system’s menu bar.) A script isn’t required to have a hat block, but if not, then the script will be run only if the user clicks on the script itself. A script can’t have more than one hat block, and the hat block can be used only at the top of the script; its distinctive shape is meant to remind you of that.¹

The other blocks in our example script are *command* blocks. Each command block corresponds to an action that **Snap!** already knows how to carry out. For example, the block **move [10] steps** tells the sprite (the arrowhead shape on the *stage* at the right end of the window) to move ten steps (a step is a very small unit of distance) in the direction in which the arrowhead is pointing. We’ll see shortly that there can be more than one sprite, and that each sprite has its own scripts. Also, a sprite doesn’t have to look like an arrowhead, but can have any picture as a *costume*. The shape of the **move** block is meant to remind you of a Lego™ brick; a script is a stack of blocks. (The word “block” denotes both the graphical shape on the screen and the procedure, the action, that the block carries out.)

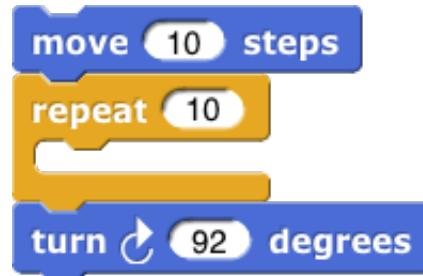
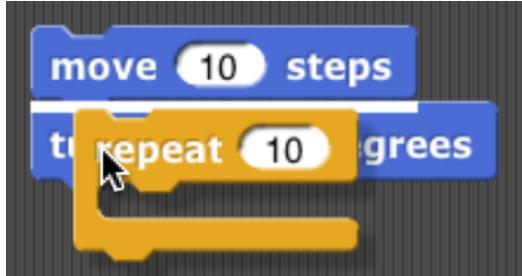
The number 10 in the **move** block above is called an *input* to the block. By clicking on the white oval, you can type any number in place of the 10. The sample script on the previous page uses 100 as the input value. We’ll see later that inputs can have non-oval shapes that accept values other than numbers. We’ll also see that you can compute input values, instead of typing a particular value into the oval. A block can have more than one input slot. For example, the **glide** block located about halfway down the Motion palette has three inputs.

¹ One of the hat blocks, the generic “when anything” block **when []**, is subtly different from the others. When the stop sign is clicked, this block no longer tests whether the condition in its hexagonal input slot is true, so the script beneath it will not run, until some *other* script in the project runs (because, for example, you click the green flag).

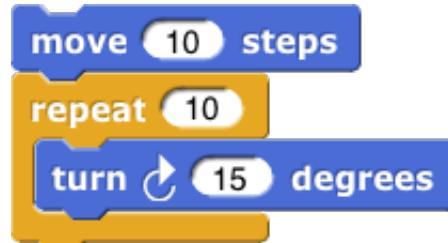
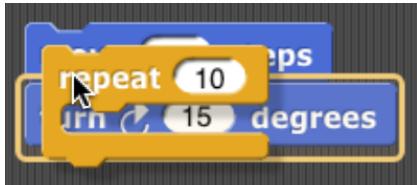
Most command blocks have that brick shape, but some, like the **repeat** block in the sample script, are *C-shaped*. Most C-shaped blocks are found in the Control palette. The slot inside the C shape is a special kind of input slot that accepts a *script* as the input. In the sample script, the **repeat** block has two inputs: the number 4 and the script



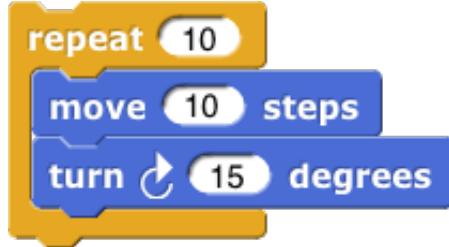
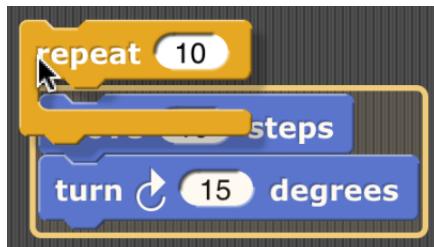
C-shaped blocks can be put in a script in two ways. If you see a white line and let go, the block will be inserted into the script like any command block:



But if you see an orange halo and let go, the block will *wrap* around the haloed blocks:

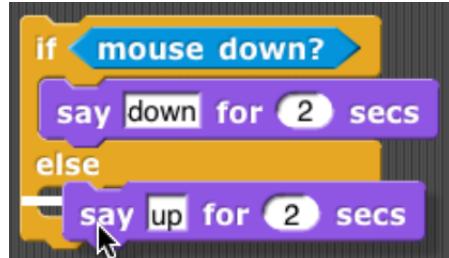


The halo will always extend from the cursor position to the bottom of the script:



If you want only some of those blocks, after wrapping you can grab the first block you don't want wrapped, pull it down, and snap it under the C-shaped block.

For “E-shaped” blocks with more than one C-shaped slot, only the first slot will wrap around existing blocks in a script, and only if that C-shaped slot is empty before wrapping. (You can fill the other slots by dragging blocks into the desired slot.)



A. Sprites and Parallelism

Just below the stage is the “new sprite” button ➤. Click the button to add a new sprite to the stage. The new sprite will appear in a random position on the stage, with a random color, but always facing to the right.

Each sprite has its own scripts. To see the scripts for a particular sprite in the scripting area, click on the picture of that sprite in the *sprite corral* in the bottom right corner of the window. Try putting one of the following scripts in each sprite’s scripting area:



When you click the green flag, you should see one sprite rotate while the other moves back and forth. This experiment illustrates the way different scripts can run in parallel. The turning and the moving happen together. Parallelism can be seen with multiple scripts of a single sprite also. Try this example:



When you press the space key, the sprite should move forever in a circle, because the **move** and **turn** blocks are run in parallel. (To stop the program, click the red stop sign at the right end of the tool bar.)

Costumes and Sounds

To change the appearance of a sprite, paint or import a new *costume* for it. To paint a costume, click on the **Costumes** tab above the scripting area, and click the paint button 🖌. The *Paint Editor* that appears is explained on page 108. There are three ways to import a costume. First select the desired sprite in the sprite corral. Then, one way is to click on the file icon in the tool bar, then choose the “**Costumes...**” menu item. You will see a list of costumes from the public media library, and can choose one. The second way, for a costume stored on your own computer, is to click on the file icon and choose the “**Import...**” menu item. You can then select a file in any picture format (PNG, JPEG, etc.) supported by your browser. The third way is quicker if the file you want is visible on the desktop: Just drag the file onto the **Snap!** window. In any of these cases, the scripting area will be replaced by something like this:



Just above this part of the window is a set of three tabs: Scripts, Costumes, and Sounds. You'll see that the Costumes tab is now selected. In this view, the sprite's *wardrobe*, you can choose whether the sprite should wear its Turtle costume or its Alonzo costume. (Alonzo, the **Snap!** mascot, is named after Alonzo Church, a mathematician who invented the idea of procedures as data, the most important way in which **Snap!** is different from Scratch.) You can give a sprite as many costumes as you like, and then choose which it will wear either by clicking in its wardrobe or by using the **switch to costume** [Turtle] or **next costume** block in a script. (Every costume has a number as well as a name. The **next costume** block selects the next costume by number; after the highest-numbered costume it switches to costume 1. The Turtle, costume 0, is never chosen by **next costume**.) The Turtle costume is the only one that changes color to match a change in the sprite's pen color. Protip: **switch to costume** [- 1] switches to the *previous* costume, wrapping like **next costume**.

In addition to its costumes, a sprite can have *sounds*; the equivalent for sounds of the sprite's wardrobe is called its *jukebox*. Sound files can be imported in any format (WAV, OGG, MP3, etc.) supported by your browser. Two blocks accomplish the task of playing sounds. If you would like a script to continue running while the sound is playing, use the block **play sound** [Help!]. In contrast, you can use the **play sound** [Help!] until done block to wait for the sound's completion before continuing the rest of the script.

Inter-Sprite Communication with Broadcast

Earlier we saw an example of two sprites moving at the same time. In a more interesting program, though, the sprites on stage will *interact* to tell a story, play a game, etc. Often one sprite will have to tell another sprite to run a script. Here's a simple example:



In the **broadcast bark and wait** block, the word "bark" is just an arbitrary name I made up. When you click on the downward arrowhead in that input slot, one of the choices (the only choice, the first time) is "**new**," which then prompts you to enter a name for the new broadcast. When this block is run, the chosen message is sent to *every* sprite, which is why the block is called "broadcast." In this program, though, only one sprite has a script to run when that broadcast is sent, namely the dog. Because the boy's script uses **broadcast and wait** rather than just **broadcast**, the boy doesn't go on to his next **say** block until the dog's script finishes. That's why the two sprites take turns talking, instead of both talking at once.

Notice, by the way, that the **say** block's first input slot is rectangular rather than oval. This means the input can be any text string, not only a number. In the text input slots, a space character is shown as a brown dot, so that you can count the number of spaces between words, and in particular you can tell the difference between an empty slot and one containing spaces. The brown dots are *not* shown on the stage when the block is run.

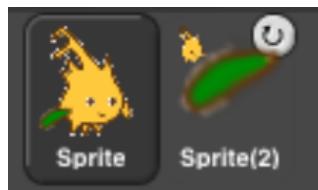
The stage has its own scripting area. It can be selected by clicking on the Stage icon at the left of the sprite corral. Unlike a sprite, though, the stage can't move. Instead of costumes, it has *backgrounds*: pictures that fill the entire stage area. The sprites appear in front of the current background. In a complicated project, it's often convenient to use a script in the stage's scripting area as the overall director of the action.

In Section VII, “Object-Oriented Programming with Sprites,” you’ll see how to send a message to a specific sprite using the **tell** and **ask** blocks, rather than to all sprites as **broadcast** does.

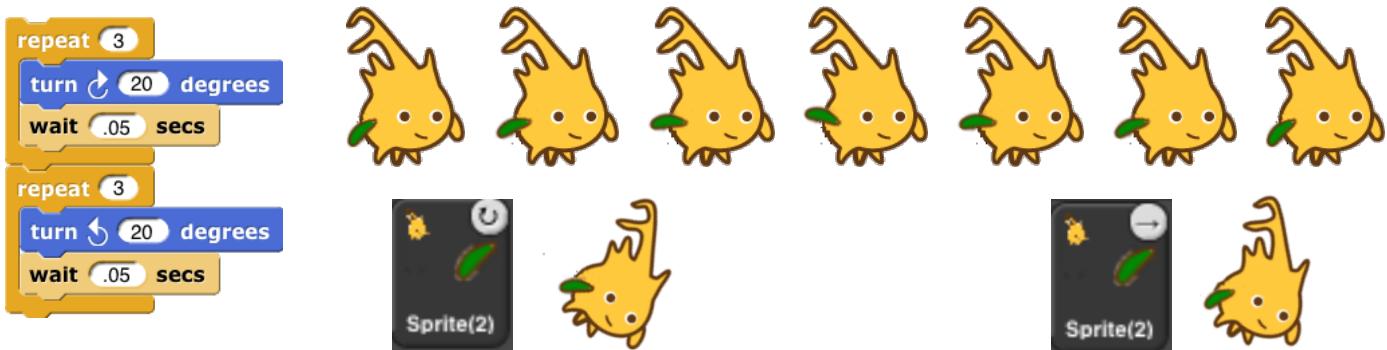
B. Nesting Sprites: Anchors and Parts

Sometimes it’s desirable to make a sort of “super-sprite” composed of pieces that can move together but can also be separately articulated. The classic example is a person’s body made up of a torso, limbs, and a head. **Snap!** allows one sprite to be designated as the *anchor* of the combined shape, with other sprites as its *parts*. To set up sprite nesting, drag the sprite corral icon of a *part* sprite onto the stage display (not the sprite corral icon!) of the desired *anchor* sprite.

Sprite nesting is shown in the sprite corral icons of both anchors and parts:



In this illustration, it is desired to animate Alonzo’s arm. (The arm has been colored green in this picture to make the relationship of the two sprites clearer, but in a real project they’d be the same color, probably.) Sprite, representing Alonzo’s body, is the anchor; Sprite(2) is the arm. The icon for the anchor shows small images of up to three attached parts at the bottom. The icon for each part shows a small image of the anchor in its top left corner, and a *synchronous/dangling rotation flag* in the top right corner. In its initial, synchronous setting, as shown above, it means that when the anchor sprite rotates, the part sprite also rotates as well as revolving around the anchor. When clicked, it changes from a circular arrow to a straight arrow, and indicates that when the anchor sprite rotates, the part sprite revolves around it, but does not rotate, keeping its original orientation. (The part can also be rotated separately, using its **turn** blocks.) Any change in the position or size of the anchor is always extended to its parts.



Top: turning the part: the green arm. Bottom: turning the anchor, with the arm synchronous (left) and dangling (right).

C. Reporter Blocks and Expressions

So far, we’ve used two kinds of blocks: hat blocks and command blocks. Another kind is the *reporter* block, which has an oval shape: **x position**. It’s called a “reporter” because when it’s run, instead of carrying out an action, it reports a value that can be used as an input to another block. If you drag a reporter into the scripting area by itself and click on it, the value it reports will appear in a speech balloon next to the block:



When you drag a reporter block over another block's input slot, a white "halo" appears around that input slot, analogous to the white line that appears when snapping command blocks together:



Here's a simple script that uses a reporter block:

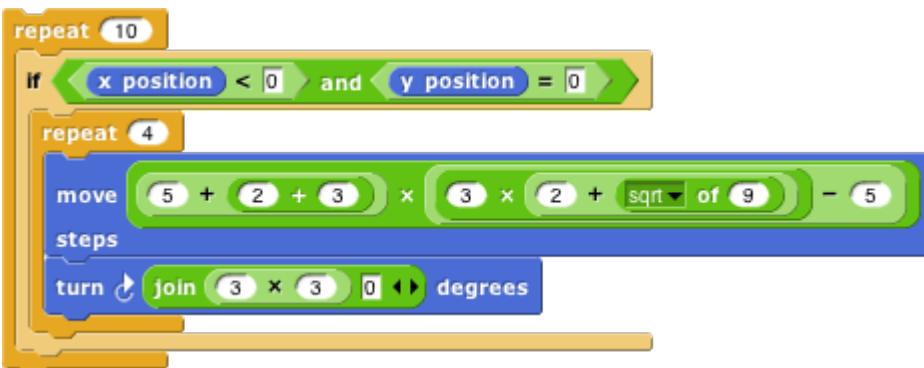


Here the **x position** reporter provides the first input to the **say** block. (The sprite's X position is its horizontal position, how far left (negative values) or right (positive values) it is compared to the center of the stage. Similarly, the Y position is measured vertically, in steps above (positive) or below (negative) the center.)

You can do arithmetic using reporters in the Operators palette:

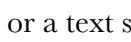


The **round** block rounds 35.3905... to 35, and the **+** block adds 100 to that. (By the way, the **round** block is in the Operators palette, just like **+**, but in this script it's a lighter color with black lettering because Snap! alternates light and dark versions of the palette colors when a block is nested inside another block from the same palette:



This aid to readability is called *zebra coloring*.) A reporter block with its inputs, maybe including other reporter blocks, such as **round x position + 100**, is called an *expression*.

D. Predicates and Conditional Evaluation

Most reporters report either a number, like  + , or a text string, like   world  . A *predicate* is a special kind of reporter that always reports **true** or **false**. Predicates have a hexagonal shape:

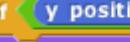


The special shape is a reminder that predicates don't generally make sense in an input slot of blocks that are expecting a number or text. You wouldn't say   steps, although (as you can see from the picture) **Snap!** lets you do it if you really want. Instead, you normally use predicates in special hexagonal input slots like this one:



The C-shaped **if** block runs its input script if (and only if) the expression in its hexagonal input reports **true**.



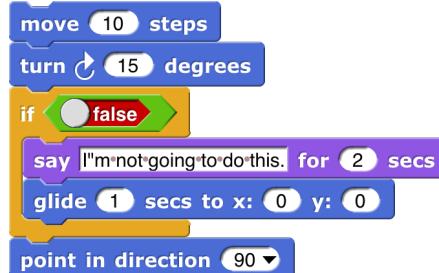
If  < 0
say Help! I'm under water!

A really useful block in animations runs its input script *repeatedly* until a predicate is satisfied:



point towards Sprite2
repeat until touching Sprite2 ?
move 3 steps

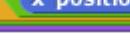
If, while working on a project, you want to omit temporarily some commands in a script, but you don't want to forget where they belong, you can say



move 10 steps
turn 15 degrees
if false
say I'm not going to do this. for 2 secs
glide 1 secs to x: 0 y: 0
point in direction 90

Sometimes you want to take the same action whether some condition is true or false, but with a different input value. For this purpose you can use the *reporter if* block:



say join I'm on the  if  < 0 then left else right

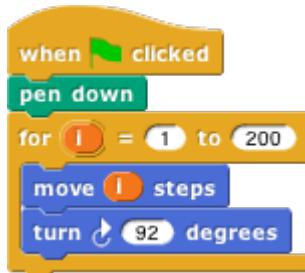
The technical term for a **true** or **false** value is a “Boolean” value; it has a capital B because it's named after a person, George Boole, who developed the mathematical theory of Boolean values. Don't get confused; a hexagonal block is a *predicate*, but the value it reports is a *Boolean*.

Another quibble about vocabulary: Many programming languages reserve the name “procedure” for Commands (that carry out an action) and use the name “function” for Reporters and Predicates. In this manual, a *procedure* is any computational capability, including those that report values and those that don't. Commands, Reporters, and Predicates are all procedures. The words “a Procedure type” are shorthand for “Command type, Reporter type, or Predicate type.”

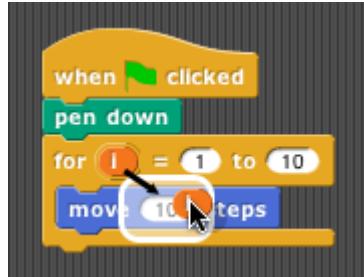
If you want to put a *constant* Boolean value in a hexagonal slot instead of a predicate-based expression, hover the mouse over the block and click on the control that appears:  

E. Variables

Try this script:



The input to the **move** block is an orange oval. To get it there, drag the orange oval that's part of the **for** block:



The orange oval is a *variable*: a symbol that represents a value. (I took this screenshot before changing the second number input to the **for** block from the default 10 to 200, and before dragging in a **turn** block.) **For** runs its script input repeatedly, just like **repeat**, but before each repetition it sets the variable **i** to a number starting with its first numeric input, adding 1 for each repetition, until it reaches the second numeric input. In this case, there will be 200 repetitions, first with **i**=1, then with **i**=2, then 3, and so on until **i**=200 for the final repetition. The result is that each **move** draws a longer and longer line segment, and that's why the picture you see is a kind of spiral. (If you try again with a turn of 90 degrees instead of 92, you'll see why this picture is called a "squiral.")

The variable **i** is created by the **for** block, and it can only be used in the script inside the block's C-slot. (By the way, if you don't like the name **i**, you can change it by clicking on the orange oval without dragging it, which will pop up a dialog window in which you can enter a different name:



"**i**" isn't a very descriptive name; you might prefer "**length**" to indicate its purpose in the script. "**i**" is traditional because mathematicians tend to use letters between **i** and **n** to represent integer values, but in programming languages we don't have to restrict ourselves to single-letter variable names.)

Global Variables

You can create variables “by hand” that aren’t limited to being used within a single block. At the top of the Variables palette, click the “**Make a variable**” button:



This will bring up a dialog window in which you can give your variable a name:



The dialog also gives you a choice to make the variable available to all sprites (which is almost always what you want) or to make it visible only in the current sprite. You’d do that if you’re going to give several sprites individual variables *with the same name*, so that you can share a script between sprites (by dragging it from the current sprite’s scripting area to the picture of another sprite in the sprite corral), and the different sprites will do slightly different things when running that script because each has a different value for that variable name.

If you give your variable the name “**name**” then the Variables palette will look like this:



There’s now a “**Delete a variable**” button, and there’s an orange oval with the variable name in it, just like the orange oval in the **for** block. You can drag the variable into any script in the scripting area. Next to the oval is a checkbox, initially checked. When it’s checked, you’ll also see a *variable watcher* on the stage:



When you give the variable a value, the orange box in its watcher will display the value.

How *do* you give it a value? You use the **set** block:



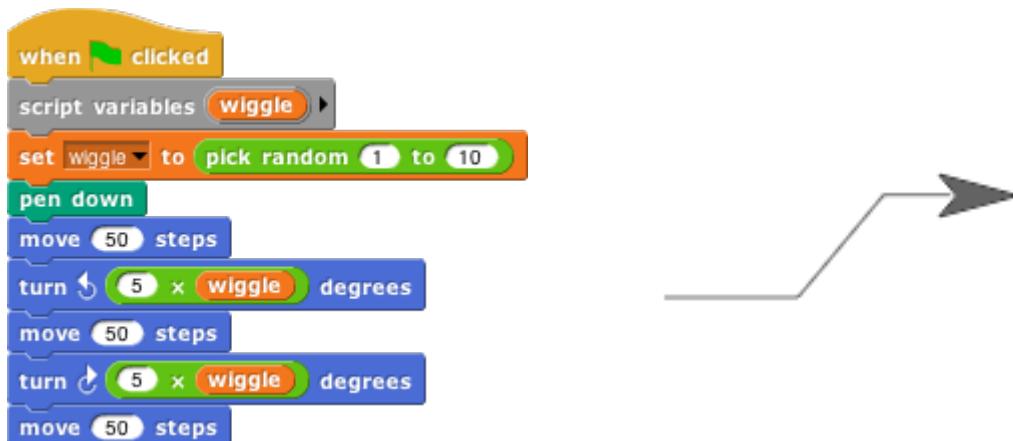
Note that you *don't* drag the variable's oval into the **set** block! You click on the downarrow in the first input slot, and you get a menu of all the available variable names.

If you do choose “For this sprite only” when creating a variable, its block in the palette looks like this:

The *location-pin* icon is a bit of a pun on a sprite-*local* variable. It’s shown only in the palette.

Script Variables

In the name example above, our project is going to carry on an interaction with the user, and we want to remember their name throughout the project. That’s a good example of a situation in which a *global* variable (the kind you make with the “**Make a variable**” button) is appropriate. Another common example is a variable called “**score**” in a game project. But sometimes you only need a variable temporarily, during the running of a particular script. In that case you can use the **script variables** block to make the variable:



As in the **for** block, you can click on an orange oval in the **script variables** block without dragging to change its name. You can also make more than one temporary variable by clicking on the right arrow at the end of the block to add another variable oval:



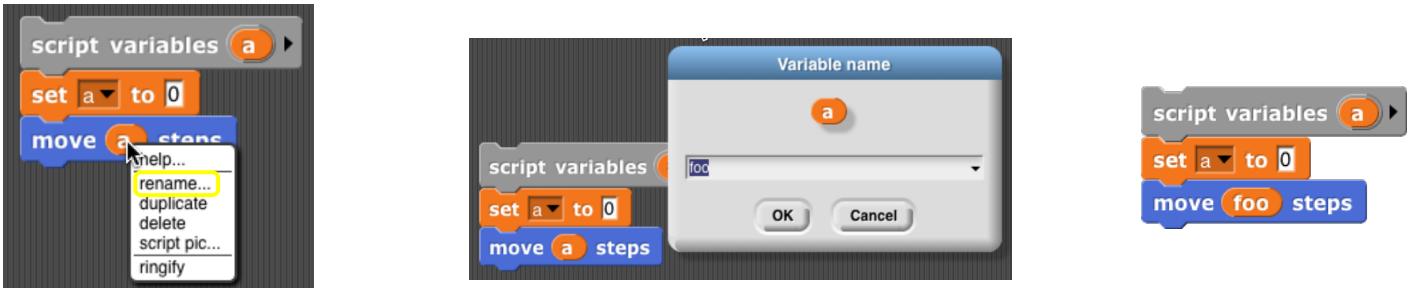
Renaming variables

There are several reasons why you might want to change the name of a variable:

1. It has a default name, such as the “**a**” in **script variables** or the “**i**” in **for**.
2. It conflicts with another name, such as a global variable, that you want to use in the same script.
3. You just decide a different name would be more self-documenting.

In the first and third case, you probably want to change the name everywhere it appears in that script, or even in all scripts. In the second case, if you’ve already used both variables in the script before realizing that they have the same name, you’ll want to look at each instance separately to decide which ones to rename. Both of these operations are possible by right-clicking or control-clicking on a variable oval.

If you right-click on an orange oval in a context in which the variable is *used*, then you are able to rename just that one orange oval:



If you right-click on the place where the variable is *defined* (a **script variables** block, the orange oval for a global variable in the Variables palette, or an orange oval that's built into a block such as the “*i*” in **for**), then you are given two renaming options, “rename” and “rename all.” If you choose “rename,” then the name is changed only in that one orange oval, as in the previous case:

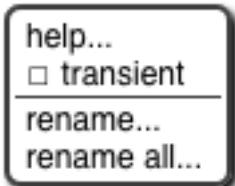


But if you choose “rename all,” then the name will be changed throughout the scope of the variable (the script for a script variable, or everywhere for a global variable):



Transient variables

So far we've talked about variables with numeric values, or with short text strings such as someone's name. But there's no limit to the amount of information you can put in a variable; in Chapter IV you'll see how to use *lists* to collect many values in one data structure, and in Chapter VIII you'll see how to read information from web sites. When you use these capabilities, your project may take up a lot of memory in the computer. If you get close to the amount of memory available to **Snap!**, then it may become impossible to save your project. (Extra space is needed to convert from **Snap!**'s internal representation to the form in which projects are exported or saved.) If your program reads a lot of data from the outside world that will still be available when you use it next, you might want to have values containing a lot of data removed from memory before saving the project. To do this, right-click or control-click on the orange oval in the Variables palette, to see this menu:



You already know about the rename options, and **help...** displays a help screen about variables in general. Here we're interested in the check box next to **transient**. If you check it, this variable's value will not be saved when you save your project. Of course, you'll have to ensure that when your project is loaded, it recreates the needed value and sets the variable to it.

F. Debugging

Snap! provides several tools to help you debug a program. They center around the idea of *pausing* the running of a script partway through, so that you can examine the values of variables.

The pause button

The simplest way to pause a program is manually, by clicking the pause button  in the top right corner of the window. While the program is paused, you can run other scripts by clicking on them, show variables on stage with the checkbox next to the variable in the Variables palette or with the **show variable** block, and do all the other things you can generally do, including modifying the paused scripts by adding or removing blocks. The button changes shape to  and clicking it again resumes the paused scripts.

Breakpoints: the **pause all** block

The pause button is great if your program seems to be in an infinite loop, but more often you'll want to set a *breakpoint*, a particular point in a script at which you want to pause. The **pause all** block, near the bottom of the Control palette, can be inserted in a script to pause when it is run. So, for example, if your program is getting an error message in a particular block, you could use **pause all** just before that block to look at the values of variables just before the error happens.

The **pause all** block turns bright cyan while paused. Also, during the pause, you can right-click on a running script and the menu that appears will give you the option to show watchers for temporary variables of the script:



But what if the block with the error is run many times in a loop, and it only errors when a particular condition is true — say, the value of some variable is negative, which shouldn't ever happen. In the iteration library (see page 23 for more about how to use libraries) is a breakpoint block that lets you set a *conditional* breakpoint, and automatically display the relevant variables before pausing. Here's a sample use of it:



(In this contrived example, variable **zot** comes from outside the script but is relevant to its behavior.) When you continue (with the pause button), the temporary variable watchers are removed by this breakpoint block before resuming the script. The breakpoint block isn't magic; you could alternatively just put a **pause all** inside an **if**.¹

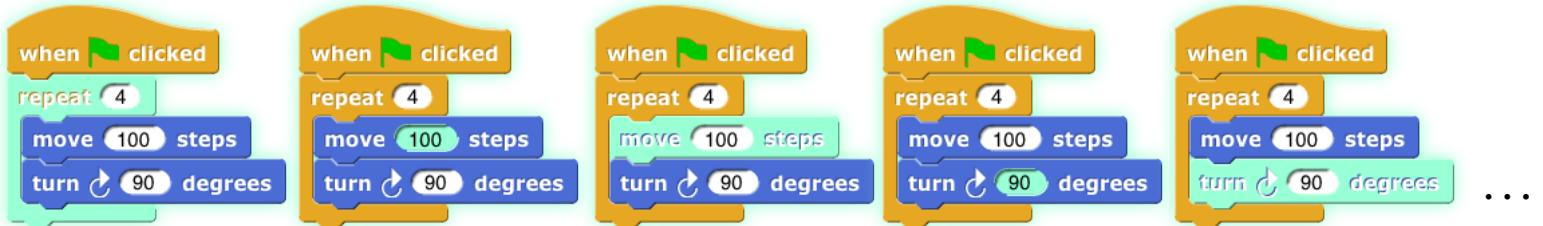
¹ The **hide variable** and **show variable** blocks can also be used to hide and show primitives in the palette. The pulldown menu doesn't include primitive blocks, but there's a generally useful technique to give a block input values it wasn't expecting using **run** or **call**:



In order to use a block as an input this way, you must explicitly put a ring around it, by right-clicking on it and choosing **ringify**. More about rings in Chapter VI. .

Visible stepping

Sometimes you're not exactly sure where the error is, or you don't understand how the program got there. To understand better, you'd like to watch the program as it runs, at human speed rather than at computer speed. You can do this by clicking the *visible stepping button* (!), before running a script or while the script is paused. The button will light up (!) and a speed control slider will appear in the toolbar. When you start or continue the script, its blocks and input slots will light up cyan one at a time:



In this simple example, the inputs to the blocks are constant values, but if an input were a more complicated expression involving several reporter blocks, each of those would light up as they are called. Note that the input to a block is evaluated before the block itself is called, so, for example, the **100** lights up before the **move**.

The speed of stepping is controlled by the slider. If you move the slider all the way to the left, the speed is zero, the pause button turns into a step button (▶), and the script takes a single step each time you push it. The name for this is *single stepping*.

If several scripts that are visible in the scripting area are running at the same time, all of them are stepped in parallel. However, consider the case of two **repeat** loops with different numbers of blocks. While not stepping, each script goes through a complete cycle of its loop in each display cycle, despite the difference in the length of a cycle. In order to ensure that the visible result of a program on the stage is the same when stepped as when not stepped, the shorter script will wait at the bottom of its loop for the longer script to catch up.

When we talk about custom blocks in Chapter III, we'll have more to say about visible stepping as it affects those blocks.

G. Etcetera

This manual doesn't explain every block in detail. There are many more motion blocks, sound blocks, costume and graphics effects blocks, and so on. You can learn what they all do by experimentation, and also by reading the "help screens" that you can get by right-clicking or control-clicking a block and selecting "help..." from the menu that appears. If you forget what palette (color) a block is, but you remember at least part of its name, type control-F and enter the name in the text block that appears in the palette area.

Here are the primitive blocks that don't exist in Scratch:

pen trails reports a new costume consisting of everything that's drawn on the stage by any sprite.

label Hello! of size 12 Print characters in the given point size on the stage, at the sprite's position and in its direction. The sprite moves to the end of the text. (Right, that's not always what you want, but you can save the sprite's position before using it, and sometimes you need to know how big the text turned out to be, in turtle steps.) If the pen is down, the text will be underlined.



See page 5.



See page 16.



runs only this script until finished.



Reporter version of the **if/else** primitive command block. Only one of the two branches is evaluated, depending on the value of the first input.



Looping block like **repeat** but with an index variable.



Extended mouse interaction events, sensing clicking, dragging, hovering, etc. The "**stopped**" option triggers when all scripts are stopped, as with the stop button; it is useful for robots whose hardware interface must be told to turn off motors. A **when I am stopped** script can run only for a limited time.



See page 72.



reports the value of a graphics effect.



Constant **true** or **false** value. See page 11.

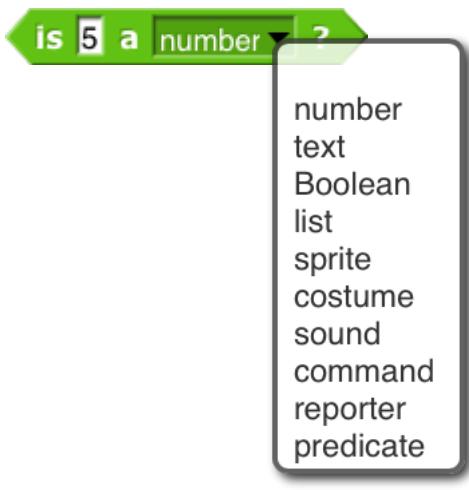


Create a primitive using JavaScript.

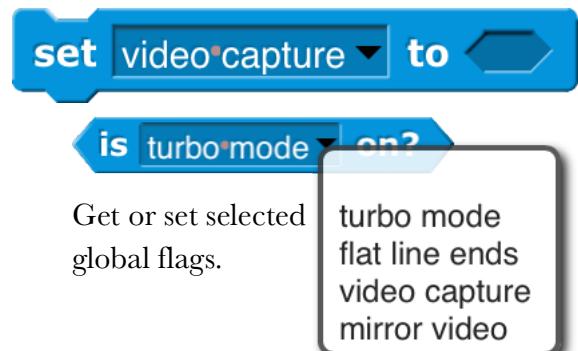




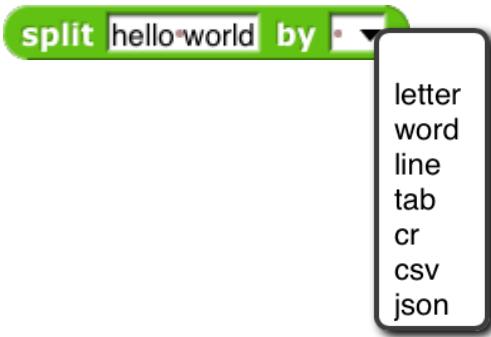
The **at** block lets you examine the screen pixel directly behind the rotation center of a sprite, the mouse, or an arbitrary (x,y) coordinate pair dropped onto the second menu slot. The first four items of the left menu let you examine the color visible at the position. The “**sprites**” option reports a list of all sprites, including this one, any point of which overlaps this sprite’s rotation center (behind or in front).



Checks the data type of a value.



Get or set selected global flags.



Turn the text into a list, using the second input as the delimiter between items. The default delimiter, indicated by the brown dot in the input slot, is a single space character. “**Letter**” puts each character of the text in its own list item. “**Word**” puts each word in an item. (Words are separated by any number of consecutive space, tab, carriage return, or newline characters.) “**Line**” is a newline character (0xa); “**tab**” is a tab character (0x9); “**cr**” is a carriage return (0xd). “**Csv**” and “**json**” split formatted text into lists of lists; see page 39.

First class list blocks (see Section IV, page 31):



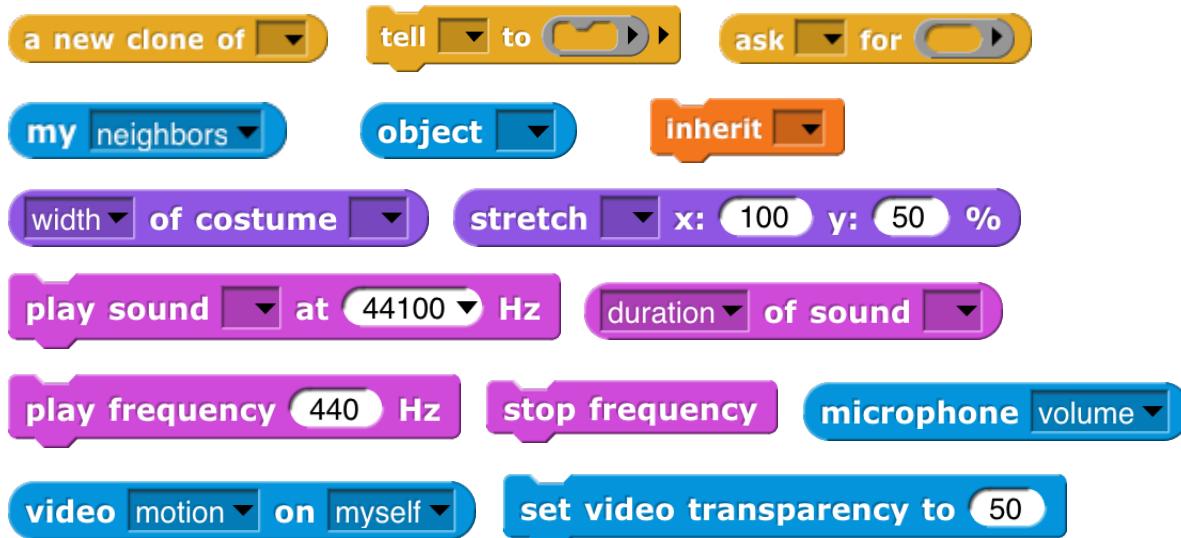
First class procedure blocks (see Section VI, page 46):



First class continuation blocks (see Section X, page 74):

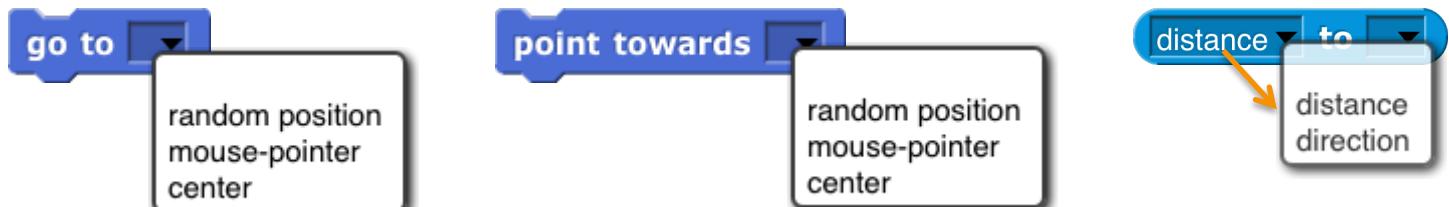


First class sprite, costume, and sound blocks (see Section VII, page 54):

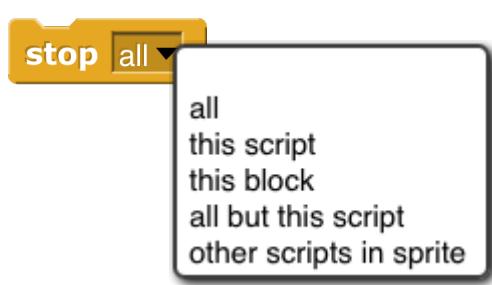


These aren't new blocks but they have a new feature:

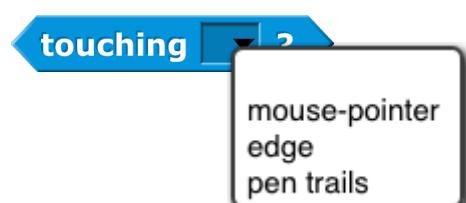
These accept two-item (x,y) lists as input, and have extended menus:



“Center” means the center of the stage, the point at coordinates (0,0). “Direction” is in the **point in direction** sense, the direction that would leave this sprite pointing toward another sprite, the mouse, or the center.



The **stop** block has two extra menu choices. **Stop this block** is used inside the definition of a custom block to stop just this invocation of this custom block and continue the script that called it. **Stop all but this script** is good at the end of a game to stop all the game pieces from moving around, but keep running this script to provide the user’s final score. The last two menu choices add a tab at the bottom of the block because the current script can continue after it.



The new “**pen trails**” option is true if the sprite is touching any drawn or stamped ink on the stage. Also, **touching** will not detect hidden sprites, but a hidden sprite can use it to detect visible sprites.

sqrt ▾ of 10

The “**neg**” option is a monadic negation operator, equivalent to . “**lg**” is \log_2 .

abs
neg
ceiling
floor
sqrt
sin
cos
tan
asin
acos
atan
ln
log
lg
 $e^$
 $10^$
 $2^$

Blocks only for the Stage:

set background color to 

change background hue ▾ by 10

set background hue ▾ to 50

II. Saving and Loading Projects and Media

After you've created a project, you'll want to save it, so that you can have access to it the next time you use **Snap!**. There are two ways to do that. You can save a project on your own computer, or you can save it at the **Snap!** web site. The advantage of saving on the net is that you have access to your project even if you are using a different computer, or a mobile device such as a tablet or smartphone. The advantage of saving on your computer is that you have access to the saved project while on an airplane or otherwise not on the net. Also, cloud projects are limited in size, but you can have all the costumes and sounds you like if you save locally. This is why we have multiple ways to save.

In either case, if you choose “Save as...” from the File menu. You’ll see something like this:



(If you are not logged in to your **Snap!** cloud account, **Computer** will be the only usable option.) The text box at the bottom right of the Save dialog allows you to enter project notes that are saved with the project.

A. Local Storage

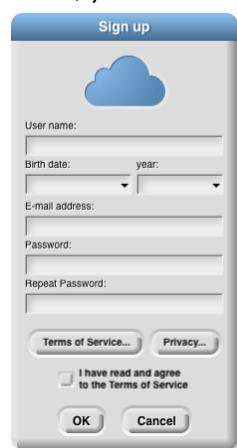
Click on **Computer** and **Snap!**'s Save Project dialog window will be replaced by your operating system's standard save window. If your project has a name, that name will be the default filename if you don't give a different name. Another, equivalent way to save to disk is to choose “Export project” from the File menu.

B. Creating a Cloud Account

The other possibility is to save your project “in the cloud,” at the **Snap!** web site. In order to do this, you need an account with us. Click on the Cloud button (cloud icon) in the Tool Bar. Choose the “Signup...” option. This will show you a window that looks like the picture at the right.

You must choose a user name that will identify you on the web site, such as **Jens** or **bh**. If you're a Scratch user, you can use your Scratch name for **Snap!** too. If you're a kid, don't pick a user name that includes your family name, but first names or initials are okay. Don't pick something you'd be embarrassed to have other users (or your parents) see! If the name you want is already taken, you'll have to choose another one. You must also supply a password.

We ask for your month and year of birth; we use this information only to decide whether to ask for your own email address or your parent's email address. (If you're a kid, you



shouldn't sign up for anything on the net, not even **Snap!**, without your parent's knowledge.) We do not store your birthdate information on our server; it is used on your own computer only during this initial signup. We do not ask for your *exact* birthdate, even for this one-time purpose, because that's an important piece of personally identifiable information.

When you click OK, an email will be sent to the email address you gave, asking you to verify (by clicking a link) that it's really your email address. We keep your email address on file so that, if you forget your password, we can send you a password-reset link. We will also email you if your account is suspended for violation of the Terms of Service. We do not use your address for any other purpose. You will never receive marketing emails of any kind through this site, neither from us nor from third parties. If, nevertheless, you are worried about providing this information, do a web search for "temporary email."

Finally, you must read and agree to the Terms of Service. A quick summary: Don't interfere with anyone else's use of the web site, and don't put copyrighted media or personally identifiable information in projects that you share with other users. And we're not responsible if something goes wrong. (Not that we *expect* anything to go wrong; since **Snap!** runs in JavaScript in your browser, it is strongly isolated from the rest of your computer. But the lawyers make us say this.)

C. Saving to the Cloud

Once you've created your account, you can log into it using the "Login..." option from the Cloud menu:



Use the user name and password that you set up earlier. If you check the "Stay signed in" box, then you will be logged in automatically the next time you run **Snap!** from the same browser on the same computer. Check the box if you're using your own computer and you don't share it with siblings. *Don't* check the box if you're using a public computer at the library, at school, etc.

Once logged in, you can choose the "Cloud" option in the "Save Project" dialog shown on page 22. You enter a project name, and optionally project notes; your project will be saved online and can be loaded from anywhere with net access. The project notes will be visible to other users if you publish your project.

D. Loading Saved Projects

Once you've saved a project, you want to be able to load it back into **Snap!**. There are two ways to do this:

1. If you saved the project in your online **Snap!** account, choose the "Open..." option from the File menu. Choose the "Cloud" button, then select your project from the list in the big text box and click OK, or choose the "Computer" button to open an operating system open dialog. (A third button, "Examples," lets you choose from example projects that we provide. You can see what each of these projects is about by clicking on it and reading its project notes.)

2. If you saved the project as an XML file on your computer, choose “Import...” from the File menu. This will give you an ordinary browser file-open window, in which you can navigate to the file as you would in other software. Alternatively, find the XML file on your desktop, and just drag it onto the **Snap!** window.

The second technique above also allows you to import media (costumes and sounds) into a project. Just choose “Import...” and then select a picture or sound file instead of an XML file.

If your project on the cloud is missing, empty, or otherwise broken: In the “Open...” box, enter your project name, then push the “Recover” button. *Do this right away*, because we save only the version before the most recent, and the latest before today. So don’t keep saving bad versions; “Recover” right away.

Snap! can also import projects created in BYOB 3.0, or in Scratch 1.4 or (with some effort; see our web site) 2.0. Almost all such projects work correctly in **Snap!**, apart from a small number of incompatible blocks. BYOB 3.1 projects that don’t use first class sprites work, too.

If you saved projects in an earlier version of **Snap!** using the “Browser” option, then a **Browser** button will be shown in the Open dialog to allow you to retrieve those projects. But you can save them only with the **Computer** and **Cloud** options.

E. Private and Public Projects

By default, a project you save in the cloud is private; only you can see it. There are two ways to make a project available to others. If you **share** a project, you can give your friends a project URL (in your browser’s URL bar after you open the project) they can use to read it. If you **publish** a project, it will appear on the **Snap!** web site, and the whole world can see it. In any case, nobody other than you can ever overwrite your project; if others ask to save it, they get their own copy in their own account.

III. Building a Block

The first version of *Snap!* was called BYOB, for “Build Your Own Blocks.” This was the first and is still the most important capability we added to Scratch. (The name was changed because a few teachers have no sense of humor. ☺ You pick your battles.) Scratch 2.0 and later also has a partial custom block capability.

A. Simple Blocks

In every palette, at or near the bottom, is a button labeled “**Make a block.**” Also, floating near the top of the palette is a plus sign.



Clicking either one will display a dialog window in which you choose the block’s name, shape, and palette/color. You also decide whether the block will be available to all sprites, or only to the current sprite and its children. Note: You can also enter the “Make a block” dialog by right-click/control-click on the script area background and then choose “**Make a block**” from the menu that appears.

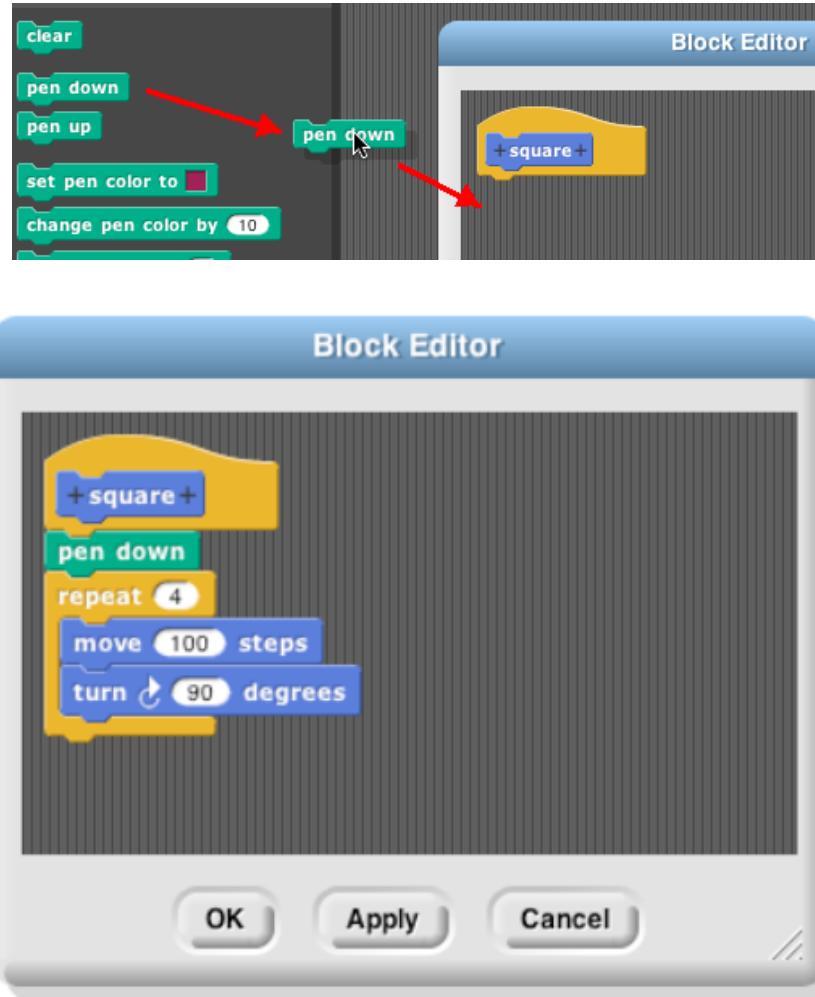


In this dialog box, you can choose the block’s palette, shape, and name. With one exception, there is one color per palette, e.g., all Motion blocks are blue. But the Variables palette includes the orange variable-related blocks and the red list-related blocks. Both colors are available, along with an “Other” option that makes grey blocks in the Variables palette for blocks that don’t fit any category.

There are three block shapes, following a convention that should be familiar to Scratch users: The jigsaw-puzzle-piece shaped blocks are Commands, and don’t report a value. The oval blocks are Reporters, and the

hexagonal blocks are Predicates, which is the technical term for reporters that report Boolean (true or false) values.

Suppose you want to make a block named “square” that draws a square. You would choose Motion, Command, and type “**square**” into the name field. When you click OK, you enter the Block Editor. This works just like making a script in the sprite’s scripting area, except that the “hat” block at the top, instead of saying something like “**when I am clicked**,” has a picture of the block you’re building. This hat block is called the *prototype* of your custom block.¹ You drag blocks under the hat to program your custom block, then click OK:



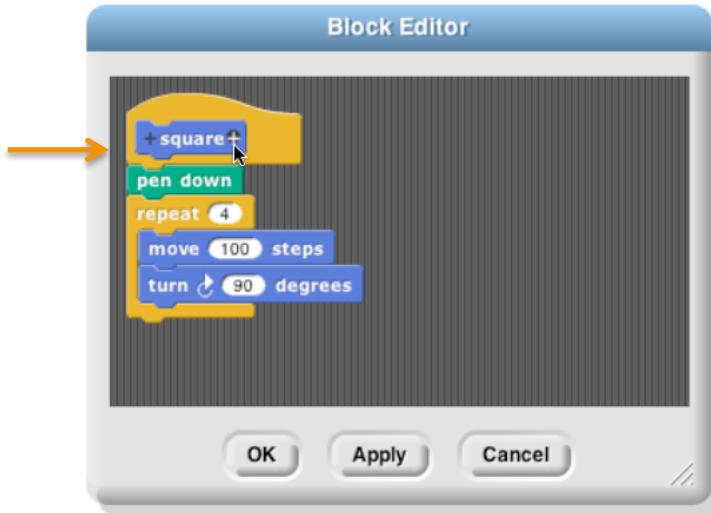
Your block appears at the bottom of the Motion palette. Here’s the block and the result of using it:



¹ This use of the word “prototype” is unrelated to the *prototyping object oriented programming* discussed later.

Custom Blocks with Inputs

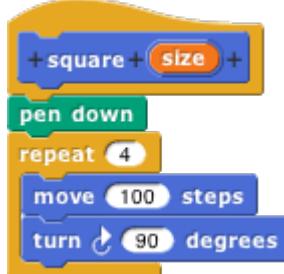
But suppose you want to be able to draw squares of different sizes. Control-click or right-click on the block, choose “edit,” and the Block Editor will open. Notice the plus signs before and after the word **square** in the prototype block. If you hover the mouse over one, it lights up:



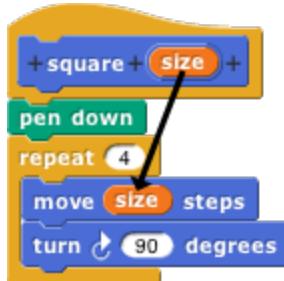
Click on the plus on the right. You will then see the “input name” dialog:



Type in the name “**size**” and click OK. There are other options in this dialog; you can choose “**title text**” if you want to add words to the block name, so it can have text after an input slot, like the “**move () steps**” block. Or you can select a more extensive dialog with a lot of options about your input name. But we’ll leave that for later. When you click OK, the new input appears in the block prototype:



You can now drag the orange variable down into the script, then click okay:



Your block now appears in the Motion palette with an input box:

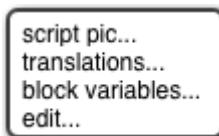


You can draw any size square by entering the length of its side in the box and running the block as usual, by double-clicking it or by putting it in a script.

Editing Block Properties

What if you change your mind about a block's color (palette) or shape (command, reporter, predicate)? If you click in the hat block at the top that holds the prototype, but not in the prototype itself, you'll see a window in which you can change the color, and *sometimes* the shape, namely, if the block is not used in any script, whether in a scripting area or in another custom block. (This includes a one-block script consisting of a copy of the new block pulled out of the palette into the scripting area, seeing which made you realize it's the wrong category. Just delete that copy (drag it back to the palette) and then change the category.)

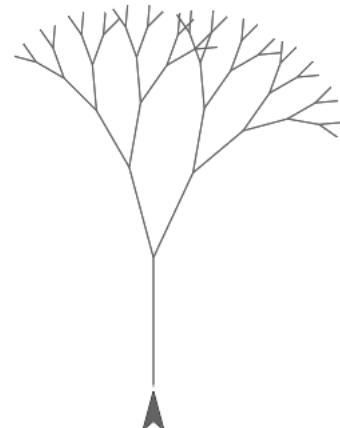
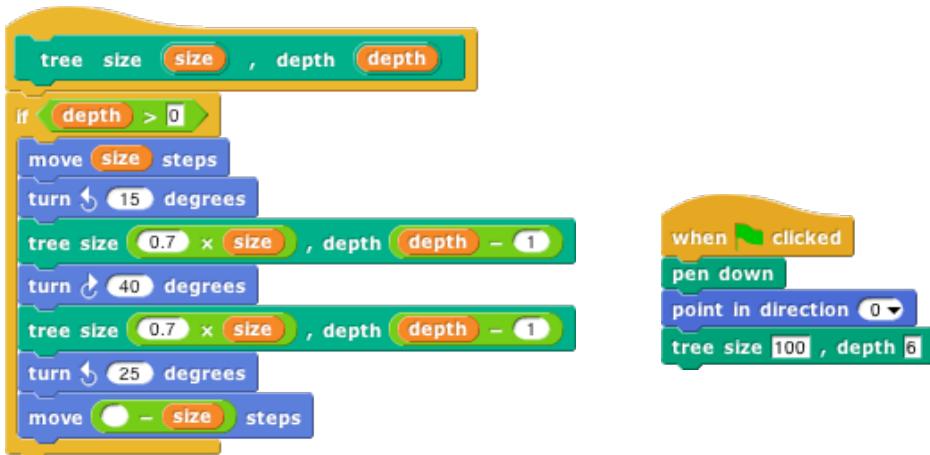
If you right-click/control-click the hat block, you get this menu:



“Script pic” exports a picture of the script. (Many of the illustrations in this manual were made that way.) “Translations” opens a window in which you can specify how your block should be translated if the user chooses a language other than the one in which you are programming. “Block variables” lets you create a variant of script variables for this block: A script variable is created when a block is called, and it disappears when that call finishes. What if you want a variable that's local to this block, as a script variable is, but doesn't disappear between invocations? That's a block variable. If the definition of a block includes a block variable, then every time that (custom) block is dragged from the palette into a script, the block variable is created. Every time *that copy* of the block is called, it uses the same block variable, which preserves its value between calls. Other copies of the block have their own block variables. “Edit” does the same thing as regular clicking, as described in the previous paragraph.

B. Recursion

Since the new custom block appears in its palette as soon as you *start* editing it, you can write recursive blocks (blocks that call themselves) by dragging the block into its own definition:



(If you added inputs to the block since opening the editor, click **Apply** before finding the block in the palette.)

If recursion is new to you, here are a few brief hints: It's crucial that the recursion have a *base case*, that is, some small(est) case that the block can handle without using recursion. In this example, it's the case `depth=0`, for which the block does nothing at all, because of the enclosing `if`. Without a base case, the recursion would run forever, calling itself over and over.

Don't try to trace the exact sequence of steps that the computer follows in a recursive program. Instead, imagine that inside the computer there are many small people, and if Theresa is drawing a tree of size 100, depth 6, she hires Tom to make a tree of size 70, depth 5, and later hires Theo to make another tree of size 70, depth 5. Tom in turn hires Tammy and Tallulah, and so on. Each little person has his or her own local variables `size` and `depth`, each with different values.

You can also write recursive reporters, like this block to compute the factorial function:



Note the use of the `report` block. When a reporter block uses this block, the reporter finishes its work and reports the value given; any further blocks in the script are not evaluated. Thus, the `if else` block in the script above could have been just an `if`, with the second `report` block below it instead of inside it, and the result would be the same, because when the first `report` is seen in the base case, that finishes the block invocation, and the second `report` is ignored. There is also a `stop this block` block that has a similar purpose, ending the block invocation early, for command blocks. (By contrast, the `stop this script` block stops not only the current block invocation, but also the entire toplevel script that called it.)

Here's a slightly more compact way to write the factorial function:

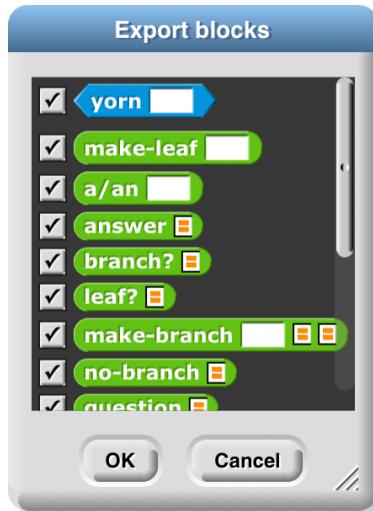


For more on recursion, see *Thinking Recursively* by Eric Roberts. (The original edition is ISBN 978-0471816522; a more recent *Thinking Recursively in Java* is ISBN 978-0471701460.)

C. Block Libraries

When you save a project (see Section II above), any custom blocks you've made are saved with it. But sometimes you'd like to save a collection of blocks that you expect to be useful in more than one project. Perhaps your blocks implement a particular data structure (a stack, or a dictionary, etc.), or they're the framework for building a multilevel game. Such a collection of blocks is called a *block library*.

To create a block library, choose “Export blocks...” from the File menu. You then see a window like this:



The window shows all of your global custom blocks. You can uncheck some of the checkboxes to select exactly which blocks you want to include in your library. (You can right-click or control-click on the export window for a menu that lets you check or uncheck all the boxes at once.) Then press OK. An XML file containing the blocks will appear in your Downloads location.

To import a block library, use the “Import...” command in the File menu, or just drag the XML file into the **Snap!** window.

Several block libraries are included with **Snap!**; for details about them, see page 23.

IV. First Class Lists

A data type is *first class* in a programming language if data of that type can be

- the value of a variable
- an input to a procedure
- the value returned by a procedure
- a member of a data aggregate
- anonymous (not named)

In Scratch, numbers and text strings are first class. You can put a number in a variable, use one as the input to a block, call a reporter that reports a number, or put a number into a list.

But Scratch's lists are not first class. You create one using the “**Make a list**” button, which requires that you give the list a name. You can't put the list into a variable, into an input slot of a block, or into a list item—you can't have lists of lists. None of the Scratch reporters reports a list value. (You can use a reduction of the list into a text string as input to other blocks, but this loses the list structure; the input is just a text string, not a data aggregate.)

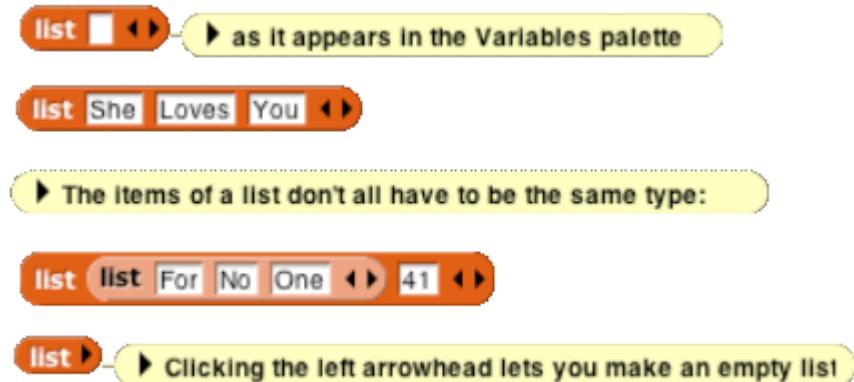
A fundamental design principle in **Snap!** is that ***all data should be first class***. If it's in the language, then we should be able to use it fully and freely. We believe that this principle avoids the need for many special-case tools, which can instead be written by **Snap!** users themselves.

Note that it's a data *type* that's first class, not an individual value. Don't think, for example, that some lists are first class, while others aren't. In **Snap!**, lists are first class, period.



A. The list Block

At the heart of providing first class lists is the ability to make an “anonymous” list—to make a list without simultaneously giving it a name. The **list** reporter block does that.



At the right end of the block are two left-and-right arrowheads. Clicking on these changes the number of inputs to **list**, i.e., the number of elements in the list you are building.

You can use this block as input to many other blocks:



Snap! does not have a “**Make a list**” button like the one in Scratch. If you want a global named “list,” make a global variable and use the **set** block to put a list into the variable.

B. Lists of Lists

Lists can be inserted as elements in larger lists. We can easily create ad hoc structures as needed:

The image shows two blocks side-by-side:

- A **list** block containing four nested lists: [list John Lennon], [list Paul McCartney], [list George Harrison], and [list Ringo Starr].
- A **table** block showing a 4x2 grid of names:

	A	B
1	John	Lennon
2	Paul	McCartney
3	George	Harrison
4	Ringo	Starr

Notice that this list is presented in a different format from the “She Loves You” list above. A two-dimensional list is called a *table* and is by default shown in *table view*. We’ll have more to say about this later.

We can also build any classic computer science data structure out of lists of lists, by defining *constructors* (blocks to make an instance of the structure), *selectors* (blocks to pull out a piece of the structure), and *mutators* (blocks to change the contents of the structure) as needed. Here we create binary trees with selectors that check for input of the correct data type; only one selector is shown but the ones for left and right children are analogous.

The image shows two scripts related to binary trees:

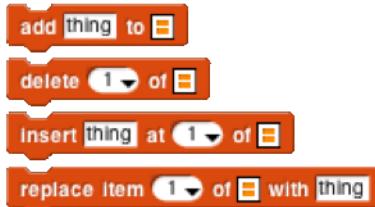
- The first script uses a **report** block to output a list of tree nodes: ["binary-tree", datum, left, right].
- The second script is a validation function for a tree node:


```

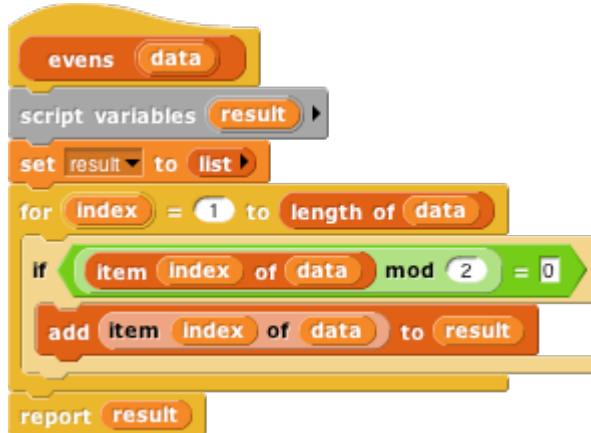
bt-datum tree
if Is tree a list?
  if item 1 of tree = "binary-tree"
    report item 2 of tree
  else
    say Join words tree isn't-a-binary-tree.
    stop script
  end
  else
    say Join words tree isn't-a-binary-tree.
    stop script
  end
end
      
```

C. Functional and Imperative List Programming

There are two ways to create a list inside a program. Scratch users will be familiar with the *imperative* programming style, which is based on a set of command blocks that modify a list:



As an example, here is a block that takes a list of numbers as input, and reports a new list containing only the even numbers from the original list:¹



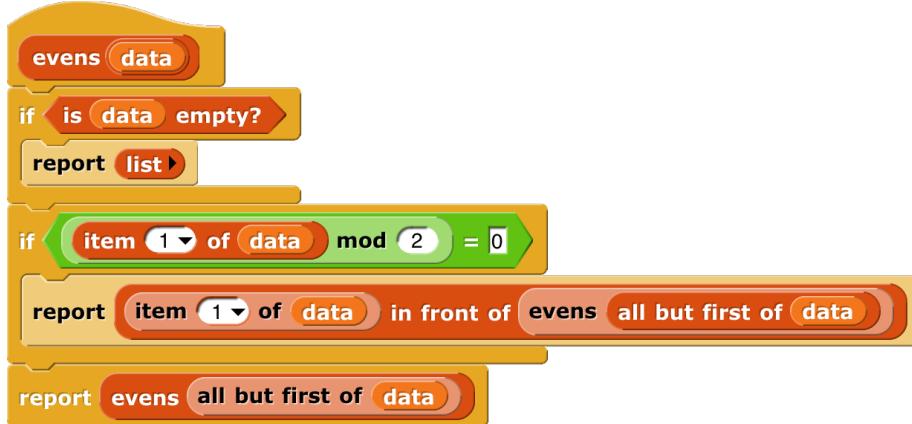
In this script, we first create a temporary variable, then put an empty list in it, then go through the items of the input list using the **add ... to (result)** block to modify the result list, adding one item at a time, and finally report the result.

Functional programming is a different approach that is becoming important in “real world” programming because of parallelism, i.e., the fact that different processors can be manipulating the same data at the same time. This makes the use of mutation (changing the value associated with a variable, or the items of a list) problematic because with parallelism it’s impossible to know the exact sequence of events, so the result of mutation may not be what the programmer expected. Even without parallelism, though, functional programming is sometimes a simpler and more effective technique, especially when dealing with recursively defined data structures. It uses reporter blocks, not command blocks, to build up a list value:

¹ Note to users of earlier versions: From the beginning, there has been a tension in our work between the desire to provide tools such as **for** (used in this example) and the higher order functions introduced on the next page as primitives, to be used as easily as other primitives, and the desire to show how readily such tools can be implemented in **Snap!** itself. This is one instance of our general pedagogic understanding that learners should both use abstractions and be permitted to see beneath the abstraction barrier. Until now, we have used the uneasy compromise of a library of tools written in **Snap!** and easily, but not easily enough, loaded into a project. By *not* loading the tools, users or teachers could explore how to program them. In 5.0 we have made them true primitives, partly because that’s what some of us wanted all along and partly because of the increasing importance of fast performance as we explore “big data” and media computation. But this is not the end of the story for us. In a later version, after we get the design firmed up, we intend to introduce “hybrid” primitives, implemented in high speed Javascript but with an “Edit” option that will open, not the primitive implementation, but the version written in **Snap!**. The trick is to ensure that this can be done without dramatically slowing users’ projects.



In a functional program, we often use recursion to construct a list, one item at a time. The **in front of** block makes a list that has one item added to the front of an existing list, *without changing the value of the original list*. A nonempty list is processed by dividing it into its first item (**item 1 of**) and all the rest of the items (**all but first of**), which are handled through a recursive call:



Snap! uses two different internal representations of lists, one (dynamic array) for imperative programming and the other (linked list) for functional programming. Each representation makes the corresponding built-in list blocks (commands or reporters, respectively) most efficient. It's possible to mix styles in the same program, but if the same list is used both ways, the program will run more slowly because it converts from one representation to the other repeatedly. (The **item () of []** block doesn't change the representation.) You don't have to know the details of the internal representations, but it's worthwhile to use each list in a consistent way.

D. Higher Order List Operations and Rings

There's an even easier way to select the even numbers from a list:



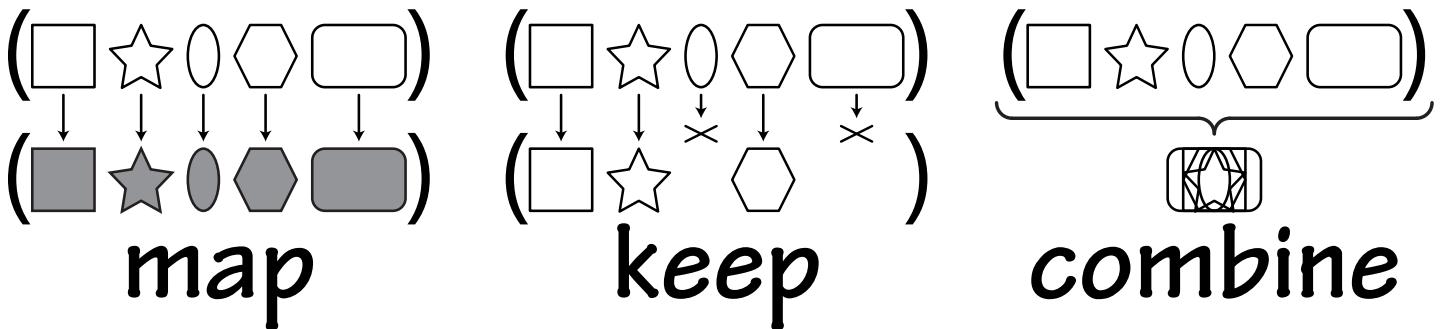
The **keep** block takes a Predicate expression as its first input, and a list as its second input. It reports a list containing those elements of the input list for which the predicate returns **true**. Notice two things about the predicate input: First, it has a grey ring around it. Second, the **mod** block has an empty input. **Keep** puts each item of its input list, one at a time, into that empty input before evaluating the predicate. (The empty input is supposed to remind you of the “box” notation for variables in elementary school: $\square + 3 = 7$.) The grey ring is part of the **keep** block as it appears in the palette:



What the ring means is that this input is a block (a predicate block, in this case, because the interior of the ring is a hexagon), rather than the value reported by that block. Here's the difference:



Evaluating the = block without a ring reports **true** or **false**; evaluating the block *with* a ring reports the block itself. This allows **keep** to evaluate the = predicate repeatedly, once for each list item. A block that takes another block as input is called a *higher order* block (or higher order procedure, or higher order function).



Snap! provides four higher order function blocks for operating on lists:



You've already seen **keep**. **Find first** is similar, but it reports just the first item that satisfies the predicate, not a list of all the matching items. It's equivalent to

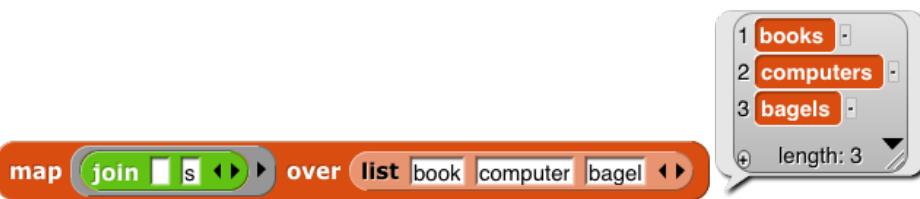
item 1 of keep items such that [list] from [list]

but faster because it stops looking as soon as it finds a match. If there are no matching items, it returns (not the empty list because a list of lists is more common than a list of Booleans, so the empty list could be a value matching a predicate).

Map takes a Reporter block and a list as inputs. It reports a new list in which each item is the value reported by the Reporter block as applied to one item from the input list. That's a mouthful, but an example will make its meaning clear:



By the way, we've been using arithmetic examples, but the list items can be of any type, and any reporter can be used. We'll make the plurals of some words:



These examples use small lists, to fit the page, but the higher order blocks work for any size list.

The third higher order block, **combine**, computes a single result from *all* the items of a list, using a *two-input* reporter as its first input. In practice, there are only a few blocks you'll ever use with **combine**:

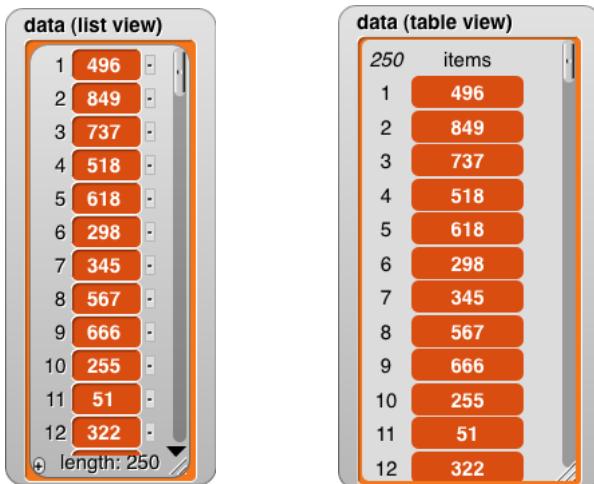
These blocks take the sum of the list items, take their product, string them into one word, combine them into a sentence (with spaces between items), see if all items of a list of Booleans are true, or see if any of the items is true.



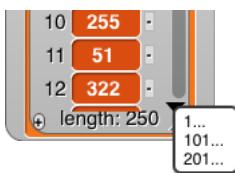
Why $+$ but not $-$? It only makes sense to combine list items using an *associative* function: one that doesn't care in what order the items are combined (left to right or right to left). $(2+3)+4 = 2+(3+4)$, but $(2-3)-4 \neq 2-(3-4)$.

E. Table View vs. List View

We mentioned earlier that there are two ways of representing lists visually. For one-dimensional lists (lists whose items are not themselves lists) the visual differences are small:



For one-dimensional lists, it's not really the appearance that's important. What matters is that the *list view* allows very versatile direct manipulation of the list through the picture: you can edit the individual items, you can delete items by clicking the tiny buttons next to each item, and you can add new items at the end by clicking the tiny plus sign in the lower left corner. (You can just barely see that the item deletion buttons have minus signs in them.) Even if you have several watchers for the same list, all of them will be updated when you change anything. On the other hand, this versatility comes at an efficiency cost; a list view watcher for a long list would be way too slow. As a partial workaround, the list view can only contain 100 items at a time; the downward-pointing arrowhead opens a menu in which you can choose which 100 to display.



By contrast, because it doesn't allow direct editing, the *table view* watcher can hold thousands of items and still scroll through them efficiently. The table view has flatter graphics for the items to remind you that they're not clickable to edit the values.

Right-clicking on a list watcher (in either form) gives you the option to switch to the other form. The right-click menu also offers an **open in dialog...** option that opens an *offstage* table view watcher, because the watchers can take up a lot of stage space that may make it hard to see what your program is actually doing. Once the offstage dialog box is open, you can close the stage watcher. There's an OK button on the offstage dialog to close it if you want. Or you can right-click it to make *another* offstage watcher, which is useful if you want to watch two parts of the list at once by having each watcher scrolled to a different place.

Table view is the default if the list has more than 100 items, or if the first item of the list is a list, in which case it makes a very different-looking two-dimensional picture:

	A	B
1	John	Lennon
2	Paul	McCartney
3	George	Harrison
4	Ringo	Starr

In this format, the column of red items has been replaced by a spreadsheet-looking display. For short, wide lists, this display makes the content of the list very clear. A vertical display, with much of the space taken up by the “machinery” at the bottom of each sublist, would make it hard to show all the text at once. (The pedagogic cost is that the *structure* is no longer explicit; we can’t tell just by looking that this is a list of row-lists, rather than a list of column-lists or a primitive two-dimensional array type. But you can choose list view to see the structure.)

Beyond such simple cases, in which every item of the main list is a list of the same length, it’s important to keep in mind that the design of table view has to satisfy two goals, not always in agreement: (1) a visually compelling display of two-dimensional arrays, and (2) highly efficient display generation, so that **Snap!** can handle very large lists, since “big data” is an important topic of study. To meet the first goal perfectly in the case of “ragged right” arrays in which sublists can have different lengths, **Snap!** would scan the entire list to find the maximum width before displaying anything, but that would violate the second goal.

Snap! uses the simplest possible compromise between the two goals: It examines only the first item of the list to decide on the format. If the first item is not a list, or is a list of one item, and the overall length is no more than 100, list view is used. If the first item is a list, then table view is used, and the number of columns in the table is equal to the number of items in the first item (sublist) of the main list.

	A	B	C
1	The	Beatles'	names
2	John	Lennon	
3	Paul	McCartney	
4	George	Harrison	
5	Ringo	Starr	

The first two examples above show that a list whose first item is a nonlist or a list of length at most one is shown in list view by default. The last two examples show that if the first item is a list of length greater than one, table view is used, with the number of columns determined by the number of items in the first sublist. In the last example above, notice the ragged right margin of items 2 and 3. This is an indication that there are more items in those sublists than in the first sublist. To see them, switch to list view.

Table views open with standard values for the width and height of a cell, regardless of the actual data. You can change these values by dragging the column letters or row numbers. Each column has its own width, but changing the height of a row changes the height for all rows. (This distinction is based not on the semantics of rows vs. columns, but on the fact that a constant row height makes scrolling through a large list more efficient.) Shift-dragging a column label will change the width of all columns.

If you tried out the adjustments in the last paragraph, you may have noticed that a column letter turns into a number when you hover over it. Labeling rows and columns differently makes cell references such as “cell 4B” unambiguous; you don’t have to have a convention about whether to say the row first or the column first. (“Cell B4” is the same as “cell 4B.”) On the other hand, to extract a value from column B in your program, you have to say **item 2 of**, not **item B of**. So it’s useful to be able to find out a column number by hovering over its letter.

Any value that can appear in a program can be displayed in a table cell:

A screenshot of the Scratch script editor. A table view is open, showing a 5x2 grid. Column A contains row numbers 1 through 5. Column B contains data: row 1 is "type example", row 2 is "number 87", row 3 is "text Rumplestiltskin", row 4 is a yellow "repeat (10)" control block, and row 5 is a yellow "sprite" icon. To the left of the table, a script is visible containing a "list" variable named "example" with items "type", "number", and "text". A "repeat (10)" loop is attached to the script, and the loop's condition is controlled by a "block" variable also named "repeat".

This display shows that the standard cell dimensions may not be enough for large value images. By expanding the entire speech balloon and then the second column and all the rows, we can make the result fit:

A screenshot of the Scratch script editor showing the same setup as above, but with the table columns and rows expanded. The second column and all rows are now fully visible, making the "repeat" block and the "sprite" icon clearly legible. The table structure remains the same: 5 rows, 2 columns, with data corresponding to the "list" variable and the "repeat" loop.

But we make an exception for cases in which the value in a cell is a list (so that the entire table is three-dimensional). Because lists are visually very big, we don’t try to fit the entire value in a cell:

A screenshot of the Scratch script editor. A table view is open with 3 rows and 2 columns. Column A contains row numbers 1, 2, and 3. Column B contains data: row 1 is a yellow "list" icon with items "name Brian Harvey", row 2 is a yellow "list" icon with items "address 784 Soda Hall Berkeley CA 94720", and row 3 is a yellow "list" icon with items "phone +1 510 642 8311". To the left of the table, a script is visible containing a "list" variable named "example" with items "name", "address", and "phone".

Even if you expand the size of the cells, **Snap!** will not display sublists of sublists in table view. There are two ways to see these inner sublists: You can switch to list view, or you can double-click on a list icon in the table to open a dialog box showing just that sub-sub-list in table view.

One last detail: If the first item of a list is a list (so table view is used), but a later item *isn’t* a list, that later item will be displayed on a red background, like an item of a single-column list:

A screenshot of the Scratch script editor. A table view is open with 2 rows and 2 columns. Column A contains row numbers 1 and 2. Column B contains data: row 1 is "foo" and "bar", and row 2 is a red "single" icon. To the left of the table, a script is visible containing a "list" variable named "example" with items "foo" and "bar".

So, in particular, if only the first item is a list, the display will look almost like a one-column display.

Comma-Separated Values

Spreadsheet and database programs generally offer the option to export their data as CSV (comma-separated values) lists. You can import these files into **Snap!** and turn them into tables (lists of lists), and you can export tables in CSV format. **Snap!** recognizes as CSV file by the extension **.csv** in its filename.

A CSV file has one line per table row, with the fields separated by commas within a row:

John, Lennon, rhythm guitar
Paul, McCartney, bass guitar
George, Harrison, lead guitar
Ringo, Starr, drums

Here's what the corresponding table looks like:

band			
4	A	B	C
1	John	Lennon	rhythm guitar
2	Paul	McCartney	bass guitar
3	George	Harrison	lead guitar
4	Ringo	Starr	drums

table view

band
1 John
2 Lennon
3 rhythm guitar
+ length: 3
1 Paul
2 McCartney
3 bass guitar
+ length: 3
1 George
2 Harrison
3 lead guitar
+ length: 4

list view

1. Make a variable with a watcher on stage: 
2. Right-click on the watcher and choose the “**import**” option. (If the variable’s value is already a list, be sure to click on the outside border of the watcher; there is a different menu if you click on the list itself.) Select the file with your csv data.
3. There is no 3; that’s it! **Snap!** will notice that the name of the file you’re importing is something **.csv** and will turn the text into a list of lists automatically.

Or, even easier, just drag and drop the file from your desktop onto the **Snap!** window, and **Snap!** will automatically create a variable named after the file and import the data into it.

If you actually want to import the raw CSV data into a variable, either change the file extension to **.txt** before loading it, or choose “**raw data**” instead of “**import**” in the watcher menu.

If you want to export a list, put a variable watcher containing the list on the stage, right-click its border, and choose “**Export**.” (Don’t right-click an item instead of the border; that gives a different menu.)

Multi-dimensional lists and JSON

CSV format is easy to read, but works only for one- or two-dimensional lists. If you have a list of lists of lists, **Snap!** will instead export your list as a JSON (JavaScript Object Notation) file. I modified my list:

replace item 1 of item 2 of band with list James Paul 

and then exported again, getting this file:

```
[["John", "Lennon", "rhythm guitar"], [["James", "Paul"], "McCartney", "bass guitar"], ["George", "Harrison", "lead guitar"], ["Ringo", "Starr", "drums"]]
```

You can also import lists, including tables, from a **.json** file. (And you can import plain text from a **.txt** file.) Drag and drop works for these formats also.

V. Typed Inputs

A. Scratch's Type Notation

Scratch block inputs come in two types: Text-or-number type and Number type. The former is indicated by a rectangular box, the latter by a rounded box: `letter 1 of world`. A third Scratch type, Boolean (true/false), can be used in certain Control blocks with hexagonal slots.

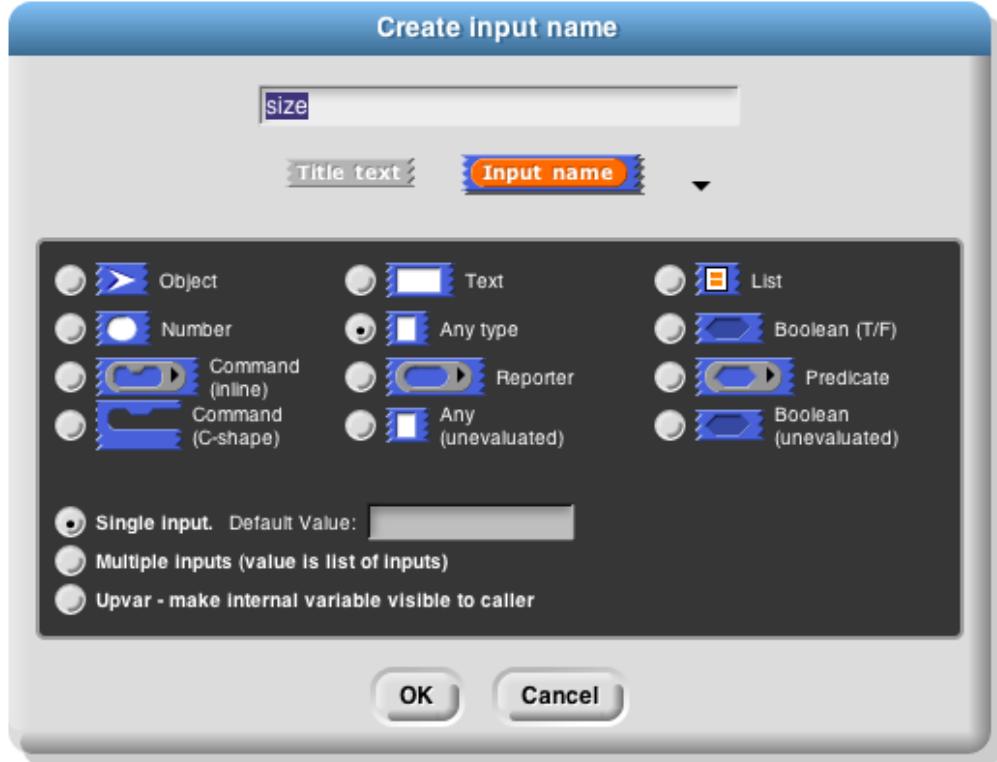
The **Snap!** types are an expanded collection including Procedure, List, and Object types. Note that, with the exception of Procedure types, all of the input type shapes are just reminders to the user of what the block expects; they are not enforced by the language.

B. The Snap! Input Type Dialog

In the Block Editor input name dialog, there is a right-facing arrowhead after the “**Input name**” option:



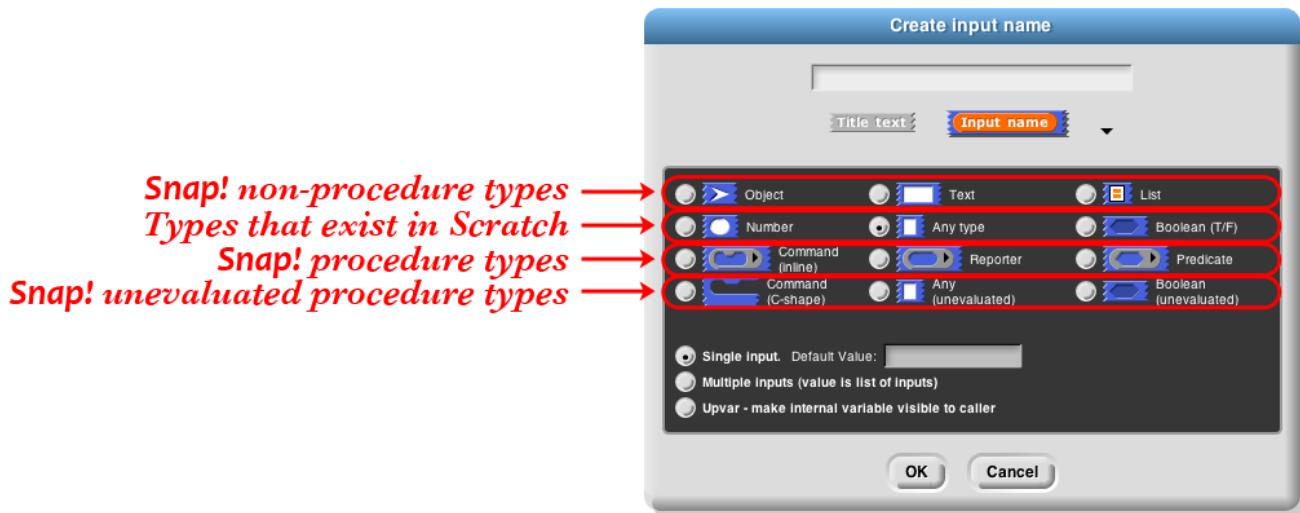
Clicking that arrowhead opens the “long” input name dialog:



There are twelve input type shapes, plus three mutually exclusive categories, listed in addition to the basic choice between title text and an input name. The default type, the one you get if you don't choose anything

else, is “Any,” meaning that this input slot is meant to accept any value of any type. If the **size** input in your block should be an oval-shaped numeric slot rather than a generic rectangle, click “**Number**.”

The arrangement of the input types is systematic. As the pictures on this and the next page show, each row of types is a category, and parts of each column form a category. Understanding the arrangement will make it a little easier to find the type you want.



The second row of input types contains the ones found in Scratch: Number, Any, and Boolean. (The reason these are in the second row rather than the first will become clear when we look at the column arrangement.) The first row contains the new **Snap!** types other than procedures: Object, Text, and List. The last two rows are the types related to procedures, discussed more fully below.

The List type is used for first class lists, discussed in Section IV above. The red rectangles inside the input slot are meant to resemble the appearance of lists as **Snap!** displays them on the stage: each element in a red rectangle.

The Object type is for sprites, costumes, sounds, and similar data types.

The Text type is really just a variant form of the Any type, using a shape that suggests a text input.¹

Procedure Types

Although the procedure types are discussed more fully later, they are the key to understanding the column arrangement in the input types. Like Scratch, **Snap!** has three block shapes: jigsaw-piece for command blocks, oval for reporters, and hexagonal for predicates. (A *predicate* is a reporter that always reports **true** or **false**.) In **Snap!** these blocks are first class data; an input to a block can be of Command type, Reporter type, or Predicate type. Each of these types is directly below the type of value that that kind of block reports, except for Commands, which don’t report a value at all. Thus, oval Reporters are related to the Any type, while hexagonal Predicates are related to the Boolean (true or false) type.

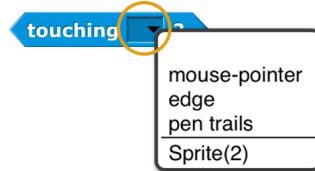
¹ In Scratch, every block that takes a Text-type input has a default value that makes the rectangles for text wider than tall. The blocks that aren’t specifically about text either are of Number type or have no default value, so those rectangles are taller than wide. At first some of us thought that Text was a separate type that always had a wide input slot; it turns out that this isn’t true in Scratch (delete the default text and the rectangle narrows), but we thought it a good idea anyway, so we allow Text-shaped boxes even for empty input slots. (This is why Text comes just above Any in the input type selection box.)

The unevaluated procedure types in the fourth row are explained in Section VI.E. below. In one handwavy sentence, they combine the *meaning* of the procedure types with the *appearance* of the reported value types two rows higher. (Of course, this isn't quite right for the C-shaped command input type, since commands don't report values. But you'll see later that it's true in spirit.)

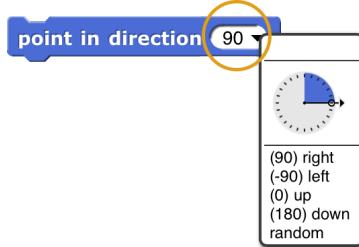


Pulldown inputs

Certain primitive blocks have *pulldown* inputs, either *read-only*, like the input to the **touching** block:

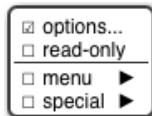


(indicated by the input slot being the same (blue, in this case) color as the body of the block), or *writeable*, like the input to the **point in direction** block:

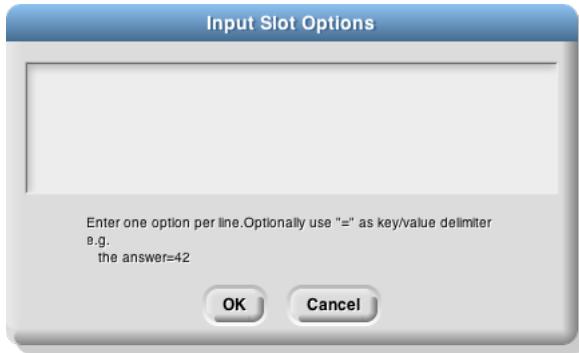


(indicated by the white input slot), which means that the user can enter an arbitrary input instead of using the pulldown menu.

Custom blocks can also have such inputs. This is an experimental feature, and the user interface is likely to change! To make a pulldown input, open the long form input dialog and control-click/right-click in the dark grey area. You will see this menu:



Click the **read-only** checkbox if you want a read-only pulldown input. Then control-click/right-click again and choose **options...** to get this dialog box:

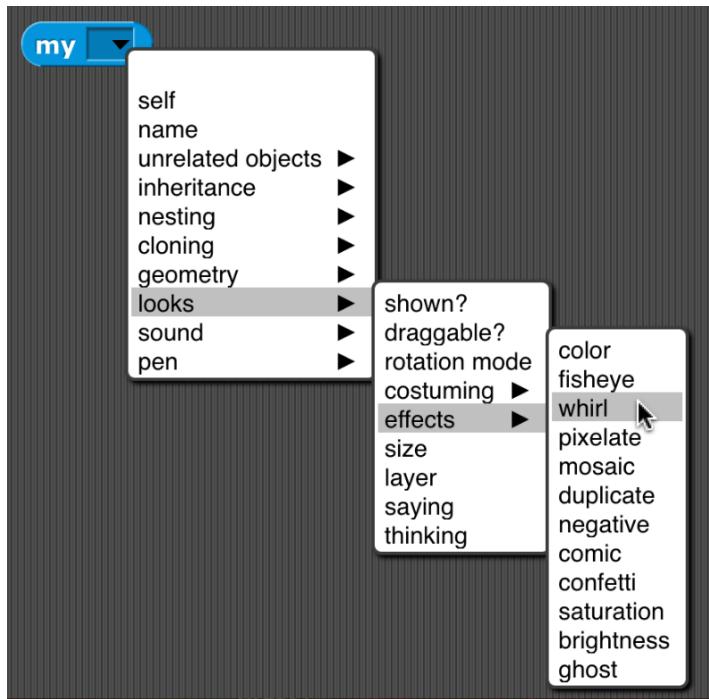


Each line in the text box represents one menu item. If the line does not contain any of the characters `=~{}` then the text is both what's shown in the menu and the value of the input if that entry is chosen.

If the line contains an equal sign `=`, then the text to the left of the equal sign is shown in the menu, and the text to the right is what appears in the input slot if that entry is chosen, and is also the value of the input as seen by the procedure.

If the line consists of a tilde `~`, then it represents a separator (a horizontal line) in the menu, used to divide long menus into visible categories. There should be nothing else on the line. This separator is not choosable, so there is no input value corresponding to it.

If the line ends with the two characters equal sign and open brace `={`, then it represents a *submenu*. The text before the equal sign is a name for the submenu, and will be displayed in the menu with an arrowhead `▶` at the end of the line. This line is not clickable, but hovering the mouse over it displays the submenu next to the original menu. A line containing a close brace `}` ends the submenu; nothing else should be on that line. Submenus may be nested to arbitrary depth.



It is also possible to get the special menus used in some primitive blocks, by choosing from the **menu** submenu: broadcast messages, sprites and stage, costumes, sounds, variables that can be **set** in this scope, the **play note** piano keyboard, or the **point in direction** 360° dial. Finally, you can make the input box accept more than one line of text (that is, text including a newline character) from the **special** submenu, either “**multi-line**” for regular text or “**code**” for monospace - font computer code.

Input variants

We now turn to the three mutually exclusive options that come below the type array.

The “**Single input**” option: In Scratch all inputs are in this category. There is one input slot in the block as it appears in its palette. If a single input is of type Any, Number, Text, or Boolean, then you can specify a default value that will be shown in that slot in the palette, like the “10” in the **move (10) steps** block. In the prototype block at the top of the script in the Block editor, an input with name “size” and default value 10 looks like this:



The “**Multiple inputs**” option: The **list** block introduced earlier accepts any number of inputs to specify the items of the new list. To allow this, **Snap!** introduces the arrowhead notation ($\blacktriangleleft \blacktriangleright$) that expands and contracts the block, adding and removing input slots. Custom blocks made by the **Snap!** user have that capability, too. If you choose the “**Multiple inputs**” button, then arrowheads will appear after the input slot in the block. More or fewer slots (as few as zero) may be used. When the block runs, all of the values in all of the slots for this input name are collected into a list, and the value of the input as seen inside the script is that list of values:



The ellipsis (...) in the orange input slot name box in the prototype indicates a multiple or *variadic* input.

The third category, “**Upvar - make internal variable visible to caller**,” isn’t really an input at all, but rather a sort of output from the block to its user. It appears as an orange variable oval in the block, rather than as an input slot. Here’s an example; the uparrow (\uparrow) in the prototype indicates this kind of internal variable name:



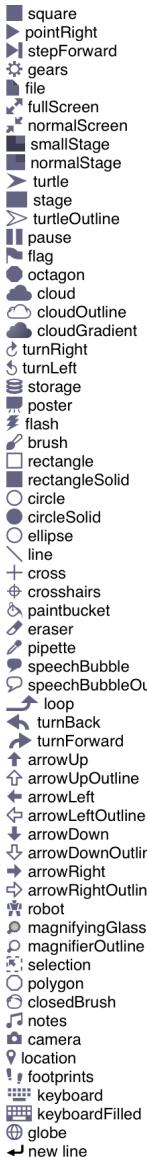
The variable **i** (in the block on the right above) can be dragged from the **for** block into the blocks used in its C-shaped command slot. Also, by clicking on the orange **i**, the user can change the name of the variable as seen in the calling script (although the name hasn’t changed inside the block’s definition). This kind of variable is called an *upvar* for short, because it is passed *upward* from the custom block to the script that uses it.

Note about the example: **for** is a primitive block, but it doesn’t need to be. You’re about to see how it can be written in **Snap!**. Just give it a different name, such as **my for** as above.

Prototype Hints

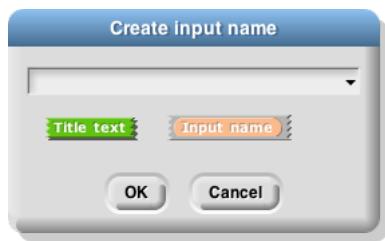
We have noted three notations that can appear in an input slot in the prototype to remind you of what kind of input this is. Here is the complete list of such notations:

=	default value	...	multiple input	↑	upvar	#	number	
λ	procedure types	:	list		?	Boolean	>	object



Title Text and Symbols

Some primitive blocks have symbols as part of the block name: Custom blocks can use symbols too. In the Block Editor, click the plus sign in the prototype at the point where you want to insert the symbol. Then click the **title text** picture below the text box that's expecting an input slot name. The dialog will then change to look like this:



The important part to notice is the arrowhead that has appeared at the right end of the text box. Click it to see the menu at the left.

Choose one of the symbols. The result will look kind of ugly in the prototype:



but the actual block will have the symbol you want:



The available symbols are, pretty much, the ones that are used in **Snap!** icons.

But I'd like the arrow symbol bigger, and yellow, so I edit its name:



This says to make the symbol **1.5** times as big as the letters in the block text, using a color with red-green-blue values of **255-255-150** (each between 0 and 255). Here's the result:



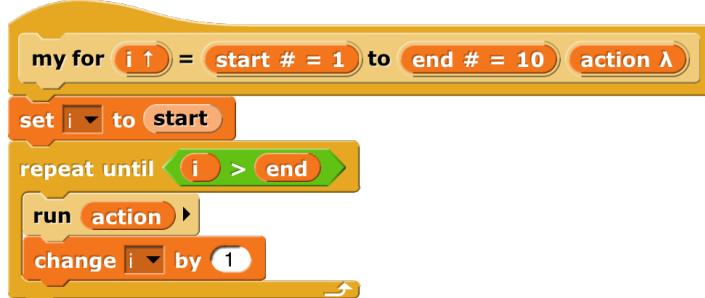
The size and color controls can also be used with text: **\$foo-8-255-120-0** will make a huge orange "foo."

Note the last entry in the symbol menu: "new line." This can be used in a block with many inputs to control where the text continues on another line, instead of letting **Snap!** choose the line break itself.

VI. Procedures as Data

A. Call and Run

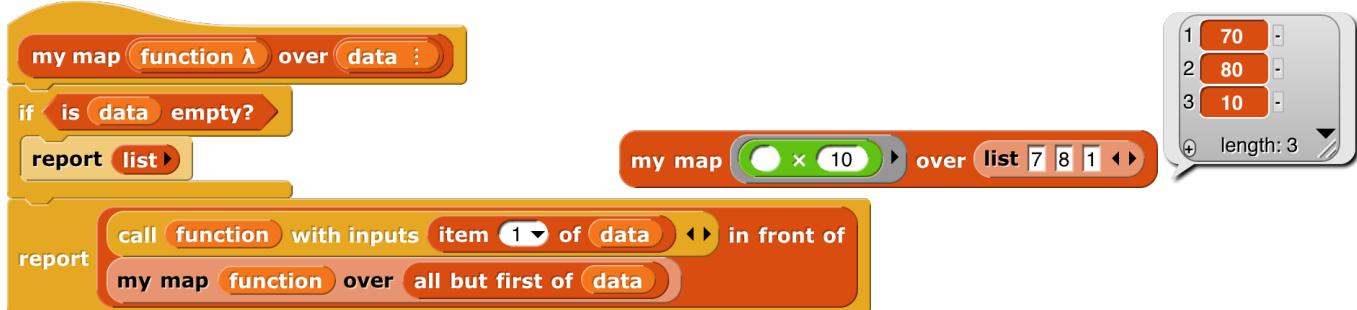
In the **for** block example above, the input named **action** has been declared as type “Command (C-shaped)”; that’s why the finished block is C-shaped. But how does the block actually tell **Snap!** to carry out the commands inside the C-slot? Here is a simple version of the block script:



This is simplified because it assumes, without checking, that the ending value is greater than the starting value; if not, the block should (depending on the designer’s purposes) either not run at all, or change the variable by -1 instead of by 1.

The important part of this script is the **run** block near the end. This is a **Snap!** built-in command block that takes a Command-type value (a script) as its input, and carries out its instructions. (In this example, the value of the input **action** is the script that the user puts in the C-slot of the **my for** block.) There is a similar **call** reporter block for invoking a Reporter or Predicate block. The **call** and **run** blocks are at the heart of **Snap!**’s first class procedure feature; they allow scripts and blocks to be used as data — in this example, as an input to a block — and eventually carried out under control of the user’s program.

Here’s another example, this time using a Reporter-type input in a **map** block (see page 35):



Here we are calling the Reporter “multiply by 10” three times, once with each item of the given list as its input, and collecting the results as a list. (The reported list will always be the same length as the input list.) Note that the multiplication block has two inputs, but here we have specified a particular value for one of them (10), so the **call** block knows to use the input value given to it just to fill the other (empty) input slot in the multiplication block. In the **my map** definition, the input **function** is declared to be type Reporter, and **data** is of type List.

Call/Run with inputs

The **call** block (like the **run** block) has a right arrowhead at the end; clicking on it adds the phrase “with inputs” and then a slot into which an input can be inserted:



If the left arrowhead is used to remove the last input slot, the “with inputs” disappears also. The right arrowhead can be clicked as many times as needed for the number of inputs required by the reporter block being called.

If the number of inputs given to **call** (not counting the Reporter-type input that comes first) is the same as the number of empty input slots, then the empty slots are filled from left to right with the given input values. If **call** is given exactly one input, then *every* empty input slot of the called block is filled with the same value:



If the number of inputs provided is neither one nor the number of empty slots, then there is no automatic filling of empty slots. (Instead you must use explicit parameters in the ring, as discussed in subsection C below.)

An even more important thing to notice about these examples is the *ring* around the Reporter-type input slots in **call** and **map** above. This notation indicates that *the block itself*, not the number or other value that the block would report when called, is the input. If you want to use a block itself in a non-Reporter-type (e.g., Any-type) input slot, you can enclose it explicitly in a ring, found at the top of the Operators palette.



As a shortcut, if you right-click or control-click on a block (such as the **+** block in this example), one of the choices in the menu that appears is “**ringify**” and/or “**unringify**.” The ring indicating a Reporter-type or Predicate-type input slot is essentially the same idea for reporters as the C-shaped input slot with which you’re already familiar; with a C-shaped slot, it’s *the script* you put in the slot that becomes the input to the C-shaped block.

There are three ring shapes. All are oval on the outside, indicating that the ring reports a value, the block or script inside it, but the inside shapes are command, reporter, or predicate, indicating what kind of block or script is expected. Sometimes you want to put something more complicated than a single reporter inside a reporter ring; if so, you can use a script, but the script must **report** a value, as in a custom reporter definition.

Variables in Ring Slots

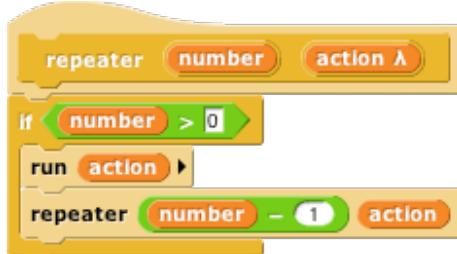
Note that the **run** block in the definition of the **my for** block (page 46) doesn’t have a ring around its input variable **action**. When you drag a variable into a ringed input slot, you generally *do* want to use *the value* of the variable, which will be the block or script you’re trying to run or call, rather than the orange variable reporter itself. So **Snap!** automatically removes the ring in this case. If you ever do want to use the variable *block itself*, rather than the value of the variable, as a Procedure-type input, you can drag the variable into the input slot, then control-click or right-click it and choose “**ringify**” from the menu that appears. (Similarly, if you ever want to call a function that will report a block to use as the input, such as **item 1 of** applied to a list of *blocks*, you can choose “**unringify**” from the menu. Almost all the time, though, **Snap!** does what you mean without help.)

B. Writing Higher Order Procedures

A *higher order procedure* is one that takes another procedure as an input, or that reports a procedure. In this document, the word “procedure” encompasses scripts, individual blocks, and nested reporters. (Unless specified otherwise, “reporter” includes predicates. When the word is capitalized inside a sentence, it means specifically oval-shaped blocks. So, “nested reporters” includes predicates, but “a Reporter-type input” doesn’t.)

Although an Any-type input slot (what you get if you use the small input-name dialog box) will accept a procedure input, it doesn’t automatically ring the input as described above. So the declaration of Procedure-type inputs makes the use of your custom higher order block much more convenient.

Why would you want a block to take a procedure as input? This is actually not an obscure thing to do; the primitive conditional and looping blocks (the C-shaped ones in the Control palette) take a script as input. Users just don't usually think about it in those terms! We could write the **repeat** block as a custom block this way, if **Snap!** didn't already have one:

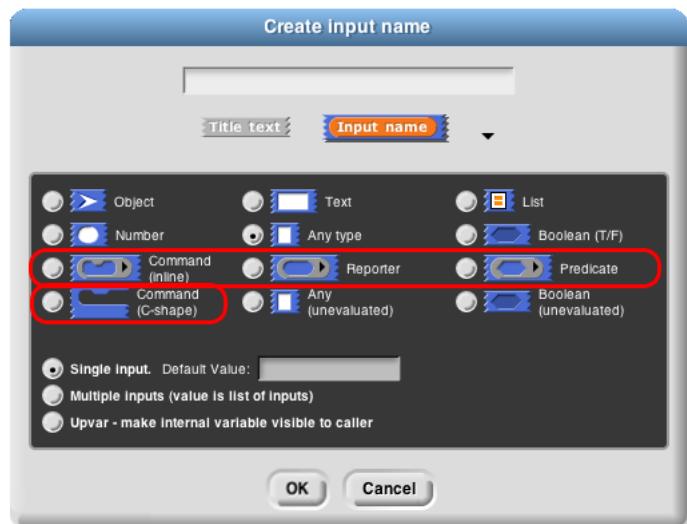


The lambda (λ) next to **action** in the prototype indicates that this is a C-shaped block, and that the script enclosed by the C when the block is used is the input named **action** in the body of the script. The only way to make sense of the variable **action** is to understand that its value is a script.

To declare an input to be Procedure-type, open the input name dialog as usual, and click on the arrowhead:



Then, in the long dialog, choose the appropriate Procedure type. The third row of input types has a ring in the shape of each block type (jigsaw for Commands, oval for Reporters, and hexagonal for Predicates). In practice, though, in the case of Commands it's more common to choose the C-shaped slot on the fourth row, because this "container" for command scripts is familiar to Scratch users. Technically the C-shaped slot is an *unevaluated* procedure type, something discussed in Section E below. The two Command-related input types (inline and C-shaped) are connected by the fact that if a variable, an **item (#) of [list]** block, or a custom Reporter block is dropped onto a C-shaped slot, it turns into an inline slot, as in the **repeater** block's recursive call above. (Other built-in Reporters can't report scripts, so they aren't accepted in a C-shaped slot.)



Why would you ever choose an inline Command slot rather than a C shape? Other than the **run** block discussed below, the only case I can think of is something like the C/C++/Java **for** loop, which actually has three command script inputs (and one predicate input), only one of which is the “featured” loop body:



Okay, now that we have procedures as inputs to our blocks, how do we use them? We use the blocks **run** (for commands) and **call** (for reporters). The **run** block’s script input is an inline ring, not C-shaped, because we anticipate that it will be rare to use a specific, literal script as the input. Instead, the input will generally be a variable whose *value* is a script.

The **run** and **call** blocks have arrowheads at the end that can be used to open slots for inputs to the called procedures. How does **Snap!** know where to use those inputs? If the called procedure (block or script) has empty input slots, **Snap!** “does the right thing.” This has several possible meanings:

1. If the number of empty slots is exactly equal to the number of inputs provided, then **Snap!** fills the empty slots from left to right:



2. If exactly one input is provided, **Snap!** will fill any number of empty slots with it:

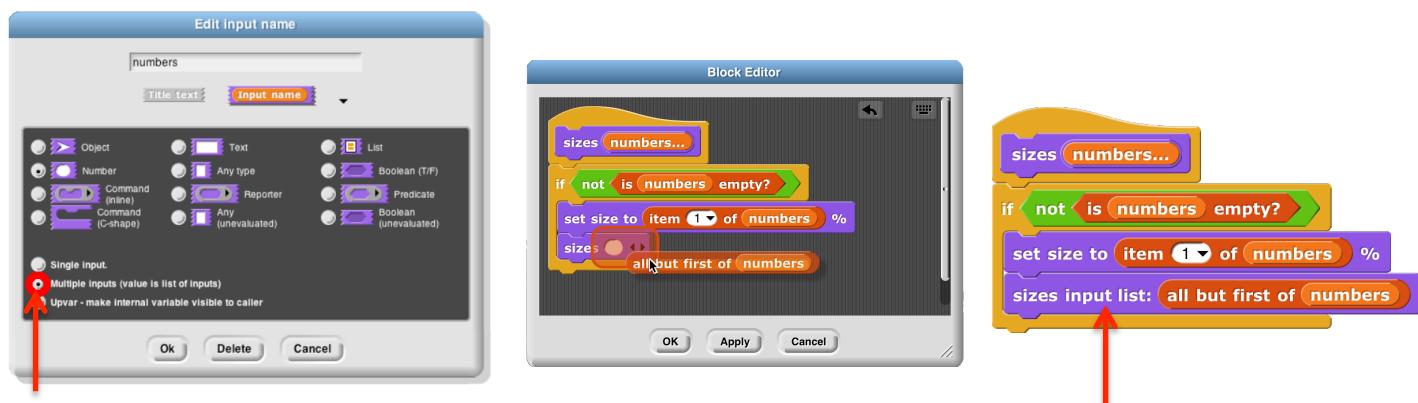


3. Otherwise, **Snap!** won’t fill any slots, because the user’s intention is unclear.

If the user wants to override these rules, the solution is to use a ring with explicit input names that can be put into the given block or script to indicate how inputs are to be used. This will be discussed more fully below.

Recursive Calls to Multiple-Input Blocks

A relatively rare situation not yet considered here is the case of a recursive block that has a variable number of inputs. Let’s say the user of your project calls your block with five inputs one time, and 87 inputs another time. How do you write the recursive call to your block when you don’t know how many inputs to give it? The answer is that you collect the inputs in a list (recall that, when you declare an input name to represent a variable number of inputs, your block sees those inputs as a list of values in the first place), and then, in the recursive call, you drop that input list *onto the arrowheads* that indicate a variable-input slot, rather than onto the input slot:



Note that the halo you see while dragging onto the arrowheads is red instead of white, and covers the input slot as well as the arrowheads. And when you drop the expression onto the arrowheads, the words “**input list:**” are added to the block text and the arrowheads disappear (in this invocation only) to remind you that the list represents all of the multiple inputs, not just a single input. The items in the list are taken *individually* as inputs to the script. Since **numbers** is a list of numbers, each individual item is a number, just what **sizes** wants. This block will take any number of numbers as inputs, and will make the sprite grow and shrink accordingly:



The user of this block calls it with any number of *individual numbers* as inputs. But inside the definition of the block, all of those numbers form *a list* that has a single input name, **numbers**. This recursive definition first checks to make sure there are any inputs at all. If so, it processes the first input (**item 1 of** the list), then it wants to make a recursive call with all but the first number. But **sizes** doesn’t take a list as input; it takes numbers as inputs! So this would be wrong:



C. Formal Parameters

The rings around Procedure-type inputs have an arrowhead at the right. Clicking the arrowhead allows you to give the inputs to a block or script explicit names, instead of using empty input slots as we’ve done until now.



The names **#1**, **#2**, etc. are provided by default, but you can change a name by clicking on its orange oval in the **input names** list. Be careful not to *drag* the oval when clicking; that’s how you use the input inside the ring. The names of the input variables are called the *formal parameters* of the encapsulated procedure.

Here’s a simple but contrived example using explicit names to control which input goes where inside the ring:



Here we just want to put one of the inputs into two different slots. If we left all three slots empty, **Snap!** would not fill any of them, because the number of inputs provided (2) would not match the number of empty slots (3).

Here is a more realistic, much more advanced example:

The Scratch script for the `crossproduct lists...` block is as follows:

```

when green flag clicked
  if [lists empty?]
    report [list v] [list v]
  else
    script variables [small]
      set [small v] to [crossproduct input list: [all but first of lists]]
      report [map v]
        map [newitem in front of v over [small]] [over]
          input names: [newitem]
          item [1 v] of [lists]
    end
  end
end

```

The output of the script is shown in the bottom right, where the input lists are `[a b]` and `[1 2 3]`, resulting in the following table:

	A	B
1	a	1
2	a	2
3	a	3
4	b	1
5	b	2
6	b	3

This is the definition of a block that takes any number of lists, and reports the list of all possible combinations of one item from each list. The important part for this discussion is that near the bottom there are two *nested* calls to `map`, the higher order function that applies an input function to each item of an input list. In the inner block, the function being mapped is `in front of`, and that block takes two inputs. The second, the empty List-type slot, will get its value in each call from an item of the inner `map`'s list input. But there is no way for the outer `map` to communicate values to empty slots of the `in front of` block. We must give an explicit name, `newitem`, to the value that the outer `map` is giving to the inner one, then drag that variable into the `in front of` block.

By the way, once the called block provides names for its inputs, *Snap!* will not automatically fill empty slots, on the theory that the user has taken control. In fact, that's another reason you might want to name the inputs explicitly: to stop *Snap!* from filling a slot that should really remain empty.

D. Procedures as Data

Here's an example of a situation in which a procedure must be explicitly marked as data by pulling a ring from the Operators palette and putting the procedure (block or script) inside it:

The Scratch script consists of a list of procedures and a call to one of them:

```

list [not v]
list [move 10 steps v if on edge, bounce v]

```

The second procedure block is expanded to show its contents:

```

1 [not v]
2 [move 10 steps v if on edge, bounce v]

```

The output of the list is `length: 2`.

Here, we are making a list of procedures. But the **list** block accepts inputs of any type, so its input slots are not ringed. We must say explicitly that we want the block *itself* as the input, rather than whatever value would result from evaluating the block.

Besides the **list** block in the example above, other blocks into which you may want to put procedures are **set** (to set the value of a variable to a procedure), **say** and **think** (to display a procedure to the user), and **report** (for a reporter that reports a procedure):



E. Special Forms

The primitive **if else** block has two C-shaped command slots and chooses one or the other depending on a Boolean test. Because Scratch doesn't emphasize functional programming, it lacks a corresponding reporter block to choose between two expressions. **Snap!** has one, but we could write our own:

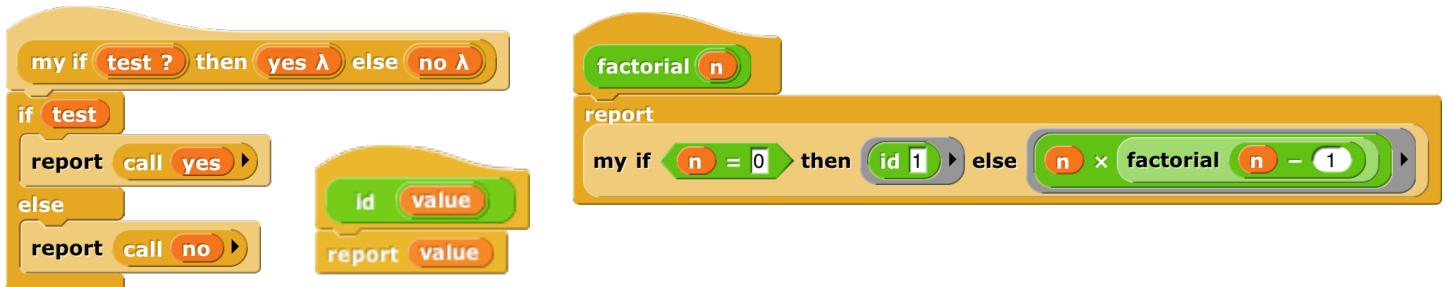


Our block works for these simple examples, but if we try to use it in writing a recursive operator, it'll fail:



The problem is that when any block is called, all of its inputs are computed (evaluated) before the block itself runs. The block itself only knows the values of its inputs, not what expressions were used to compute them. In particular, all of the inputs to our **if then else** block are evaluated first thing. That means that even in the base case, **factorial** will try to call itself recursively, causing an infinite loop. We need our **if then else** block to be able to select only one of the two alternatives to be evaluated.

We have a mechanism to allow that: declare the **then** and **else** inputs to be of type Reporter rather than type Any. Then, when calling the block, those inputs will be enclosed in a ring so that the expressions themselves, rather than their values, become the inputs:



In this version, the program works, with no infinite loop. But we've paid a heavy price: this reporter-**if** is no longer as intuitively obvious as the Scratch command-**if**. You have to know about procedures as data, about rings, and about a trick to get a constant value in a ringed slot. (The **id** block implements the identity function, which reports its input. We need it because rings take only reporters as input, not numbers.) What we'd like is a reporter-**if** that *behaves* like this one, delaying the evaluation of its inputs, but *looks* like our first version, which was easy to use except that it didn't work.

Such blocks are indeed possible. A block that seems to take a simple expression as input, but delays the evaluation of that input by wrapping an “invisible ring” around it (and, if necessary, an **id**-like transformation of constant data into constant functions) is called a *special form*. To turn our **if** block into a special form, we edit the block's prototype, declaring the inputs **yes** and **no** to be of type “**Any (unevaluated)**” instead of type Reporter. The script for the block is still that of the second version, including the use of **call** to evaluate either **yes** or **no** but not both. But the slots appear as white Any-type rectangles, not Reporter-type rings, and the **factorial** block will look like our first attempt.

In a special form's prototype, the unevaluated input slot(s) are indicated by a lambda (λ) next to the input name, just as if they were declared as Procedure type. They *are* Procedure type, really; they're just disguised to the user of the block.

Special forms trade off implementor sophistication for user sophistication. That is, you have to understand all about procedures as data to make sense of the special form implementation of **my if then else**. But any experienced Scratch programmer can *use* **my if then else** without thinking at all about how it works internally.

Special Forms in Scratch

Special forms are actually not a new invention in **Snap!**. Many of Scratch's conditional and looping blocks are really special forms. The hexagonal input slot in the **if** block is a straightforward Boolean value, because the value can be computed once, before the **if** block makes its decision about whether or not to run its action input. But the **forever if**, **repeat until**, and **wait until** blocks' inputs can't be Booleans; they have to be of type “Boolean (unevaluated),” so that Scratch can evaluate them over and over again. Since Scratch doesn't have custom C-shaped blocks, it can afford to handwave away the distinction between evaluated and unevaluated Booleans, but **Snap!** can't. The pedagogic value of special forms is proven by the fact that no Scratcher ever notices that there's anything strange about the way in which the hexagonal inputs in the Control blocks are evaluated.

Also, the C-shaped slot familiar to Scratch users is an unevaluated procedure type; you don't have to use a ring to keep the commands in the C-slot from being run before the C-shaped block is run. Those commands themselves, not the result of running them, are the input to the C-shaped Control block. (This is taken for granted by Scratch users, especially because Scratchers don't think of the contents of a C-slot as an input at all.) This is why it makes sense that “C-shaped” is on the fourth row of types in the long form input dialog, with other unevaluated types.

VII. Object Oriented Programming with Sprites

Object oriented programming is a style based around the abstraction *object*: a collection of *data* and *methods* (procedures, which from our point of view are just more data) that you interact with by sending it a *message* (just a name, maybe in the form of a text string, and perhaps additional inputs). The object responds to the message by carrying out a method, which may or may not report a value back to the asker. Some people emphasize the *data hiding* aspect of OOP (because each object has local variables that other objects can access only by sending request messages to the owning object) while others emphasize the *simulation* aspect (in which each object abstractly represents something in the world, and the interactions of objects in the program model real interactions of real people or things). Data hiding is important for large multi-programmer industrial projects, but for **Snap!** users it's the simulation aspect that's important. Our approach is therefore less restrictive than that of some other OOP languages; we give objects easy access to each others' data and methods.

Technically, object oriented programming rests on three legs: (1) *Message passing*: There is a notation by which any object can send a message to another object. (2) *Local state*: Each object can remember the important past history of the computation it has performed. (“Important” means that it need not remember every message it has handled, but only the lasting effects of those messages that will affect later computation.) (3) *Inheritance*: It would be impractical if each individual object had to contain methods, many of them identical to those of other objects, for all of the messages it can accept. Instead, we need a way to say that this new object is just like that old object except for a few differences, so that only those differences need be programmed explicitly.

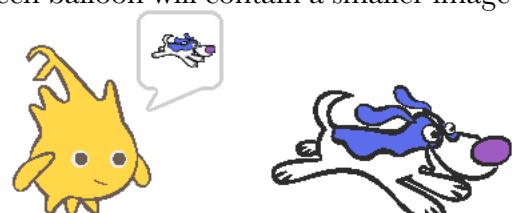
A. First Class Sprites

Like Scratch, **Snap!** comes with things that are natural objects: its sprites. Each sprite can own local variables; each sprite has its own scripts (methods). A Scratch animation is plainly a simulation of the interaction of characters in a play. There are two ways in which Scratch sprites are less versatile than the objects of an OOP language. First, Scratch message passing is weak in three respects: Messages can only be **broadcast**, not addressed to an individual sprite; messages can't take inputs; and methods can't return values to their caller. Second, and more basic, in the OOP paradigm objects are *data*; they can be the value of a variable, an element of a list, and so on, but that's not the case for Scratch sprites.

Snap! sprites are first class data. They can be created and deleted by a script, stored in a variable or list, and sent messages individually. The children of a sprite can inherit sprite-local variables, methods (sprite-local procedures), and other attributes (e.g., **x position**).

The fundamental means by which programs get access to sprites is the **my** reporter block. It has a dropdown-menu input slot that, when clicked, gives access to all the sprites, plus the stage. **my self** reports a single sprite, the one asking the question. **my othersprites** reports a list of all sprites other than the one asking the question. **my neighbors** reports a list of all sprites that are *near* the one asking — the ones that are candidates for having collided with this one, for example. The **my** block has many other options, discussed below. If you know the name of a particular sprite, the **object** reporter will report the sprite itself.

An object or list of objects reported by **my** or **object** can be used as input to any block that accepts any input type, such as **set**'s second input. If you **say** an object, the resulting speech balloon will contain a smaller image of the object's costume or (for the stage) background.



B. Permanent and Temporary Clones

The **a new clone of myself** block is used to create and report an instance (a clone) of any sprite. (There is also a command version, for historical reasons.) There are two different kinds of situations in which clones are used. One is that you've made an example sprite and, when you start the project, you want a fairly large number of essentially identical sprites that behave like the example. (Hereafter we'll call the example sprite the "parent" and the others the "children.") Once the game or animation is over, you don't need the copies any more. (As we'll see, "copies" is the wrong word because the parent and the children *share* a lot of properties. That's why we use the word "clones" to describe the children rather than "copies.") These are *temporary* clones. They are automatically deleted when the user presses either the green flag or the red stop sign. In Scratch 2.0 and later, all clones are temporary.

The other kind of situation is what happens when you want specializations of sprites. For example, let's say you have a sprite named Dog. It has certain behaviors, such as running up to a person who comes near it. Now you decide that the family in your story really likes dogs, so they adopt a lot of them. Some are cocker spaniels, who wag their tails when they see you. Others are rottweilers, who growl at you when they see you. So you make a clone of Dog, rename it Cocker Spaniel, and give it a new costume and a script for what to do when someone gets near. You make another clone of Dog, rename it Rottweiler, and give it a new costume, etc. Then you make three clones of Cocker Spaniel (so there are four altogether) and two clones of Rottweiler. Maybe you hide the Dog sprite after all this, since it's no breed in particular. Each dog has its own position, special behaviors, and so on. You want to save all of these dogs in the project. These are *permanent* clones. In BYOB 3.1, the predecessor to **Snap!**, all clones are permanent.

One advantage of temporary clones is that they don't slow down **Snap!** even when you have a lot of them. (If you're curious, one reason is that permanent clones appear in the sprite corral, where their pictures have to be updated to reflect the clone's current costume, direction, and so on.) We have tried to anticipate your needs, as follows: When you make a clone in a script, using the **a new clone of myself** block, it is "born" temporary. But when you make a clone from the user interface, for example by right-clicking on a sprite and choosing "clone," it is born permanent. The reason this makes sense is that you don't create 100 *kinds* of dogs automatically. Each kind has many different characteristics, programmed by hand. But when your project is running, it might create 100 rottweilers, and those will be identical unless you change them in the program.

You can change a temporary sprite to permanent by right-clicking it and choosing "edit." (It's called "edit" rather than, say, "permanent" because it also shifts the scripting area to reflect that sprite, as if you'd pressed its button in the sprite corral.) You can change a permanent sprite to temporary by right-clicking it and choosing "release."

C. Sending Messages to Sprites

The messages that a sprite accepts are the blocks in its palettes, including both all-sprites and this-sprite-only blocks. (For custom blocks, the corresponding methods are the scripts as seen in the Block Editor.)

The way to send a message to a sprite (or the stage) is with the **tell** block (for command messages) or the **ask** block (for reporter messages).





A small point to note in the examples above: most of the primitive blocks' dropdown menus include an empty entry at the top, which can be selected for use in higher order procedures like the **for each** and **map** examples. Each of the sprites in **my neighbors** or **my other sprites** is used to fill the blank space in turn.

By the way, if you want a list of *all* the sprites, including this sprite, you can use either of these:



Tell and **ask** wait until the other sprite has carried out its method before this sprite's script continues. (That has to be the case for **ask**, since we want to do something with the value it reports.) So **tell** is analogous to **broadcast and wait**. Sometimes the other sprite's method may take a long time, or may even be a **forever** loop, so you want the originating script to continue without waiting. For this purpose we have the **launch** block:



Launch is analogous to **broadcast** without the “wait.”

Snap! 4.1, following BYOB 3.1, used an extension of the **of** block to provide access to other sprites' methods. That interface was designed back when we were trying hard to avoid adding new primitive blocks; it allowed us to write **ask** and **tell** as tool procedures in **Snap!** itself. That technique still works, but is deprecated, because nobody understood it, and now we have the more straightforward primitives.

Polymorphism

Suppose you have a Dog sprite with two clones CockerSpaniel and PitBull. In the Dog sprite you define this method (“For this sprite only” block):



Note the *location* (map-pin) symbol before the block's name. The symbol is not part of the block title; it's a visual reminder that this is a sprite-local block. Sprite-local variables are similarly marked.

But you don't define **greet as friend** or **greet as enemy** in Dog. Each kind of dog has a different behavior. Here's what a CockerSpaniel does:



And here's what a PitBull does:

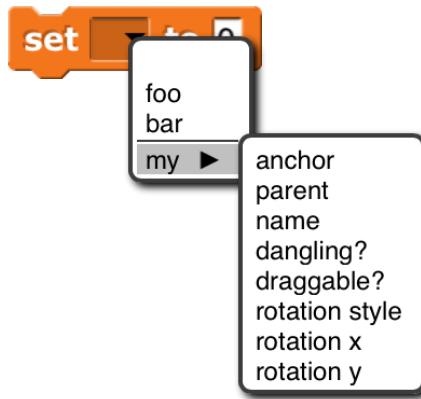


Greet is defined in the Dog sprite. If Fido is a particular cocker spaniel, and you ask Fido to **greet** someone, Fido inherits the **greet** method from Dog, but Dog itself couldn't actually run that method, because Dog doesn't have **greet as friend** or **greet as enemy**. And perhaps only individual dogs such as Fido have **friend?** methods. Even though the **greet** method is defined in the Dog sprite, when it's running it remembers what specific dog sprite called it, so it knows which **greet as friend** to use. Dog's **greet** block is called a *polymorphic* method, because it means different things to different dogs, even though they all share the same script.

D. Local State in Sprites: Variables and Attributes

A sprite's memory of its own past history takes two main forms. It has *variables*, created explicitly by the user with the "Make a variable" button; it also has *attributes*, the qualities every sprite has automatically, such as position, direction, and pen color. Each variable can be examined using its own orange oval block; there is one **set** block to modify all variables. Attributes, however, have a less uniform programming interface in Scratch. A sprite's direction can be examined with the **direction** block, and modified with the **point in direction** block. It can also be modified less directly using the blocks **turn**, **point towards**, and **if on edge, bounce**. There is no way for a script to examine a sprite's pen color, but there are blocks **set pen color to <color>**, **set pen color to <number>**, and **change pen color** to modify it. A sprite's name can be neither examined nor modified by scripts; it can be modified by typing a new name directly into the box that displays the name, above the scripting area. The block, if any, that examines a variable or attribute is called its *getter*; a block (there may be more than one, as in the examples above) that modifies a variable or attribute is called a *setter*.

In **Snap!** we have allow virtually all attributes to be examined. But instead of adding dozens of reporters, we use a more uniform interface for attributes: The **my** block's menu includes many of the attributes of a sprite. It serves as a general getter for those attributes, e.g., **my [anchor]** to find the sprite, if any, to which this sprite is attached in a nesting arrangement (see page 9). Similarly, the same **set** block used to set variable values allows setting some sprite attributes:



E. Prototyping: Parents and Children

Most current OOP languages use a *class/instance* approach to creating objects. A class is a particular *kind of object*, and an instance is an *actual object* of that type. For example, there might be a Dog class, and several instances Fido, Spot, and Runt. The class typically specifies the methods shared by all dogs (RollOver, SitUpAndBeg, Fetch, and so on), and the instances contain data such as species, color, and friendliness. **Snap!** uses a different approach called *prototyping*, in which there is no distinction between classes and instances. Prototyping is better suited to an experimental, tinkering style of work: You make a single dog sprite, with both methods (blocks) and

data (variables); you can actually watch it and interact with it on the stage; and when you like it, you use it as the prototype from which to clone other dogs. If you later discover a bug in the behavior of dogs, you can edit a method in the parent, and all of the children will automatically share the new version of the method block. Experienced class-instance programmers may find prototyping strange at first, but it is actually a more expressive system, because you can easily simulate a class-instance hierarchy by hiding the prototype sprite! Prototyping is also a better fit with the Scratch design principle that everything in a project should be concrete and visible on the stage; in class-instance OOP the programming process begins with an abstract, invisible entity, the class, that must be designed before any concrete objects can be made.¹

There are three ways to make a child sprite. If you control-click or right-click on a sprite in the “sprite corral” at the bottom right corner of the window, you get a menu that includes “clone” as one of the choices. There is an **a new clone of** block in the Control palette that creates and reports a child sprite. And sprites have a “parent” attribute that can be set, like any attribute, thereby *changing* the parent of an existing sprite.



F. Inheritance by Delegation

A clone *inherits* properties of its parent. “Properties” include scripts, custom blocks, variables, named lists, system attributes, costumes, and sounds. Each individual property can be shared between parent and child, or not shared (with a separate one in the child). The getter block for a shared property, in the child’s palette, is displayed in a lighter color; separate properties of the child are displayed in the traditional colors.

When a new clone is created, by default it shares only its methods, wardrobe, and jukebox with its parent. All other properties are copied to the clone, but not shared. (One exception is that a new *permanent* clone is given a random position. Another is that *temporary* clones share the scripts in their parent’s scripting area. A third is that sprite-local variables that the parent creates *after* cloning are shared with its children.) If the value of a shared property is changed in the parent, then the children see the new value. If the value of a shared property is changed in the *child*, then the sharing link is broken, and a new private version is created in that child. (This is the mechanism by which a child chooses not to share a property with its parent.) “Changed” in this context means using the **set** or **change** block for a variable, editing a block in the Block Editor, editing a costume or sound, or inserting, deleting, or reordering costumes or sounds. To change a property from unshared to shared, the child uses the **inherit** command block. The pulldown menu in the block lists all the things this sprite can

¹ Some languages popular in the “real world” today, such as JavaScript, claim to use prototyping, but their object system is much more complicated than what we are describing (we’re guessing it’s because they were designed by people too familiar with class-instance programming); that has, in some circles, given prototyping a bad name. Our prototyping design comes from Object Logo, and before that, from Henry Lieberman. [Lieberman, H., Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems, First Conference on Object-Oriented Programming Languages, Systems, and Applications [OOPSLA-86], ACM SigCHI, Portland, OR, September, 1986. Also in *Object-Oriented Computing*, Gerald Peterson, Ed., IEEE Computer Society Press, 1987.]

inherit from its parent (which might be nothing, if this sprite has no parent) and is not already inheriting. But that would prevent telling a child to inherit, so if the **inherit** block is inside a ring, its pulldown menu includes all the things a child could inherit from this sprite. Right-clicking on the scripting area of a permanent clone gives a menu option to share the entire collection of scripts from its parent, as a temporary clone does.

The rules are full of details, but the basic idea is simple: Parents can change their children, but children can't directly change their parents. That's what you'd expect from the word "inherit": the influence just goes in one direction. When a child changes some property, it's declaring independence from its parent (with respect to that one property). What if you really want the child to be able to make a change in the parent (and therefore in itself and all its siblings)? Remember that in this system any object can **tell** any other object to do something:



When a sprite gets a message for which it doesn't have a corresponding block, the message is *delegated* to that sprite's parent. When a sprite does have the corresponding block, then the message is not delegated. If the script that implements a delegated message refers to **my (self)**, it means the child to which the message was originally sent, not the parent to which the message was delegated.



G. List of attributes

At the right is a picture of the dropdown menu of attributes in the **my** block. Several of these are not real attributes, but things related to attributes:

- **self**: this sprite
- **neighbors**: a list of *nearby* sprites
- **other sprites**: a list of all sprites except myself
- **stage**: the stage, which is first-class, like a sprite
- **clones**: a list of my *temporary* clones
- **other clones**: a list of my *temporary* siblings
- **parts**: a list of sprites whose **anchor** attribute is this sprite
- **children**: a list of all my clones, temporary and permanent

The others are individual attributes:

- **anchor**: the sprite of which I am a (nested) part
- **parent**: the sprite of which I am a clone
- **temporary?**: am I a temporary clone?
- **name**: my name (same as parent's name if I'm temporary)
- **costumes**: a list of the sprite's costumes
- **sounds**: a list of the sprite's sounds
- **dangling?**: True if I am a part and not in synchronous orbit
- **draggable?**: True if the user can move me with the mouse
- **width, height**: The width or height of my costume *as seen right now*, taking rotation into account.
- **rotation x, rotation y**: when reading with **my**, the same as **x position, y position**. When **set**, changes the sprite's rotation center *without moving the sprite*, like dragging the rotation center in the paint editor.
- **center x, center y**: the x and y position of the center of my bounding box, rounded off—the geometric center of the costume.

neighbors
self
other sprites
clones
other clones
parts
anchor
stage
children
parent
temporary?
name
costume
costumes
sounds
dangling?
draggable?
width
height
rotation style
rotation x
rotation y
center x
center y

H. First Class Costumes and Sounds

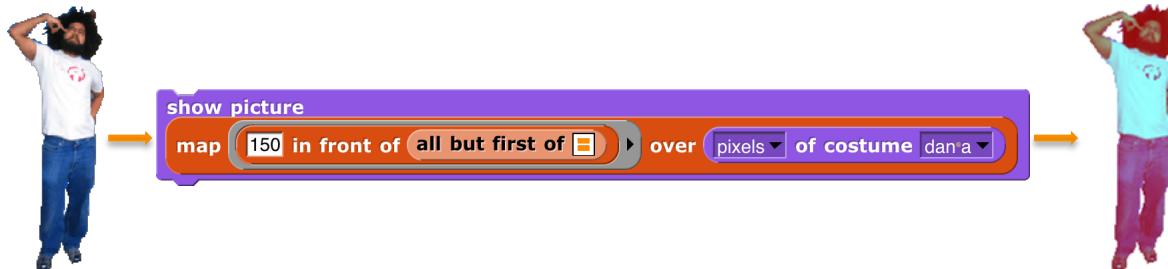
Costumes and sounds don't have methods, as sprites do; you can't ask them to do things. But they *are* first class: you can make a list of them, put them in variables, use them as input to a procedure, and so on. **My [costumes]** and **my [sounds]** report lists of them.

Media Computation with Costumes

The components of a costume are its name, width, height, and pixels. The block  gives access to these components using its left menu. From its right menu you can choose the current costume, the Turtle costume, or any costume in the sprite's wardrobe. Since costumes are first class, you can also drop an expression whose value is a costume on that second input slot. (Due to a misfeature, even though you can select Turtle in the right menu, the block reports 0 for its width and height, and an empty string for the other components.) The costume's width and height are in its standard orientation, regardless of the sprite's current direction. (This is different from the *sprite's* width and height, reported by the **my** block.)

But the really interesting part of a costume is its bitmap, a list of *pixels*. (A pixel, short for "picture element," represents one dot on your display.) Each pixel is itself a list of four items, the red, green, and blue components of its color (in the range 0-255) and what is standardly called its "transparency" but should be called its opacity, in the range 0-100, in which 0 means that the pixel is invisible and 100 means that it's fully opaque: you can't see anything from a rearward layer at that point on the stage. (Costume pixels typically have an opacity of 0 only for points inside the bounding box of the costume but not actually part of the costume; points in the interior of a costume typically have an opacity of 100. Intermediate values appear mainly at the edge of a costume, or at sharp boundaries between colors inside the costume, where they are used to reduce "jaggies": the staircase-like shape of a diagonal line displayed on an array of discrete rectangular screen coordinates. Note that the opacity of a *sprite* pixel is determined by combining the costume's opacity with the sprite's ghost effect. (The latter really is a measure of transparency: 0 means opaque and 100 means invisible.)

The bitmap is a one-dimensional list of pixels, not an array of *height* rows of *width* pixels each. That's why the pixel list has to be combined with the dimensions to produce a costume. This choice partly reflects the way bitmaps are stored in the computer's hardware and operating system, but also makes it easy to produce transformations of a costume with **map**:



In this simplest possible transformation, the red value of all the pixels have been changed to a constant 150. Colors that were red in the original (such as the logo printed on the t-shirt) become closer to black (the other color components being near zero); the blue jeans become purple (blue plus red); perhaps counterintuitively, the white t-shirt, which has the maximum value for all three color components, loses some of its red and becomes cyan, the color opposite red on the color wheel. In reading the code, note that the function that is the first input to **map** is applied to a single pixel, whose first item is its red component. Also note that this process works only on bitmap costumes; if you call **pixels of** on a vector costume (one with "svg" in the corner of its picture), it will be converted to pixels first.

The **show picture** block is in the “Pixels” library. It abbreviates an invocation of the **switch to costume** block with an expression whose value is a costume dropped onto its input menu slot:



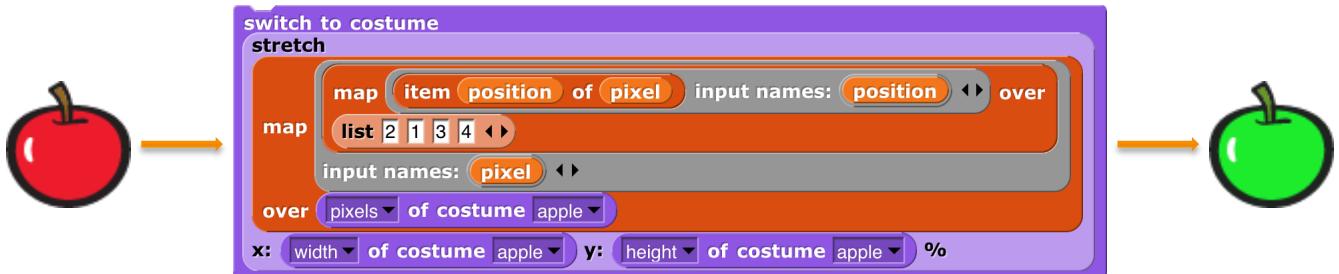
The important point to see here is that a bitmap (list of pixels) is not, by itself, a costume. The **stretch** block creates a costume by combining a bitmap, a width, and a height. (For the moment, ignore the “%” at the end of its title text; it’s relevant to a completely different use of the **stretch** block, coming up.) Note that costumes computed in this way don’t have names. Note also that the use of **switch to costume** does *not* add the computed costume to the sprite’s wardrobe; to do that, say **add [my costume] to [my costumes]**

Here’s a more interesting example of color manipulation:



Each color value is constrained to be 0, 80, 160, or 240. This gives the picture a more cartoonish look.

Here’s one way to exchange red and green values, using nested **map** blocks with explicit formal parameters in the two rings:



It’s the list **list [2 1 3 4]** that determines the rearrangement of colors: green→red, red→green, and the other two unchanged.

When the **stretch** block is given a list (presumably a list of pixels) as its first input, as above, then its other two inputs are the exact width and height of the desired costume, measured in units of one pixel. But **stretch** will also take a costume as its first input, either by selecting a costume from the menu or by dropping a costume-valued expression such as **item [3] of [my costumes]** onto it. In that case, the other two inputs are percents of the original width and height, as advertised, so you can make fun house mirror versions of costumes:



The resulting costumes can be used with **switch to costume** and so on.

Finally, you can use pictures from your computer’s camera in your projects using these blocks:

video motion **on myself**

set video transparency to **50**

Using the **video on** block turns on the camera and displays what it sees on the stage, regardless of the inputs given. The camera remains on until you click the red stop button, your program runs the **stop all** block, or you turn it off explicitly with the  block. The video image on the stage is partly ghosted, to an extent determined by the **set video transparency** block, whose input really is transparency and not opacity. (Small numbers make the video more visible.) By default, the video image is mirrored, like the selfie camera on your cell phone: When you raise your left hand, your image raises its right hand. You can control this mirroring with the  block.

The **video snap on** block then takes a still picture from the camera, and trims it to fit on the selected sprite. (**Video snap on stage** means to use the entire stage-sized rectangle.) For example, here's a camera snapshot trimmed to fit Alonzo:



The “Video Capture” project in the Examples collection repeatedly takes such trimmed snapshots and has the Alonzo sprite use the current snapshot as its costume, so it looks like this:



(The picture above was actually taken with transparency set to 50, to make the background more visible for printing.) Because the sprite is always still in the place where the snapshot was taken, its costume exactly fits in with the rest of the full-stage video. If you were to add a **move 100 steps** block after the **switch to costume**, you'd see something like this:



This time, the sprite's costume was captured at one position, and then the sprite is shown at a different position. (You probably wouldn't want to do this, but perhaps it's helpful for explanatory purposes.)

What you *would* want to do is push the sprite around the stage:

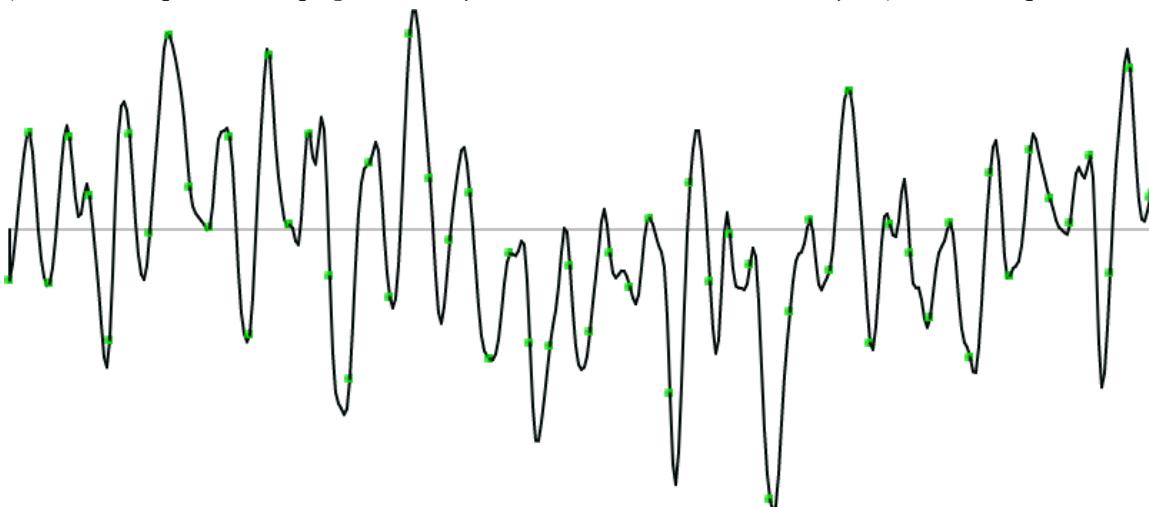


(Really these should be Jens's picture; it's his project. But he's vacationing. ☺) **Video motion** compares two snapshots a moment apart, looking only at the part within the given trim (here **myself**, meaning the current sprite, not the person looking into the camera), to detect a difference between them. It reports a number, measuring the number of pixels through which some part of the picture has moved. **Video direction** also compares two snapshots to detect motion, but what it reports is the direction (in the **point in direction** sense) of the motion. So the script above moves the sprite in the direction in which it's being pushed, but only if a significant amount of motion is found; otherwise the sprite would jiggle around too much. And yes, you can run the second script without the first to push a balloon around the stage.

Media Computation with Sounds

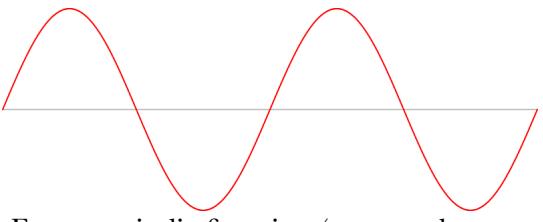
The starting point for computation with sound is the **microphone** block. It starts by recording a brief sample of sound from your microphone. (How brief? On my computer, 0.010667 seconds, but you'll see shortly how to find out or control the sample size on your computer.)

Just as the *pixel* is the smallest piece of a picture, the *sample* is the smallest piece of a sound. It says here: **microphone sample rate** 48000 that on my computer, 48,000 samples are recorded per second, so each sample is $1/48,000$ of a second. The value of a sample is between -1 and 1, and represents the sound pressure on the microphone—how hard the air is pushing—at that instant. (You can skip the next page or so if you know about Fourier analysis.) Here's a picture of 400 samples:



volume
note
frequency
samples
sample rate
spectrum
resolution

In this graph, the *x* axis represents the time at which each sample was measured; the *y* axis measures the value of the sample at that time. The first obvious thing about this graph is that it has a lot of ups and downs. The most basic up-and-down function is the *sine wave*:



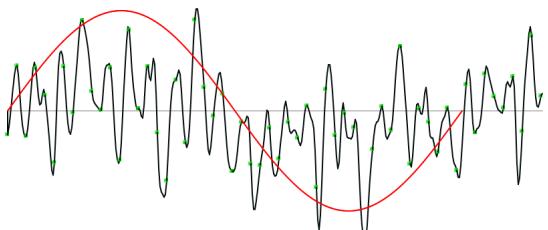
Every periodic function (more or less, any sample that sounds like music rather than sounding like static) is composed of a sum of sine waves of different frequencies.

Look back at the graph of our sampled sound. There is a green dot every seven samples. There's nothing magic about the number seven; I tried different values until I found one that looked right. What "right" means is that, for the first few dots at least, they coincide almost perfectly with the high points and low points of the graph. Near the middle (horizontally) of the graph, the green dots don't seem anywhere near the high and low points, but if you find the very lowest point of the graph, about 2/3 of the way along, the dots start lining up almost perfectly again.

The red graph above shows two *cycles* of a sine wave. One cycle goes up, then down, then up again. The amount of time taken for one cycle is the *period* of the sine function. If the green dots match both ups and downs in the captured sound, then two dots—14 samples, or $14/48000$ of a second—represent the period. The first cycle and a half of the graph looks like it could be a pure sine wave, but after that, the tops and bottoms don't line up, and there are peculiar little jiggles, such as the one before the fifth green dot. This happens because sine waves of different periods are added together.

It turns out to be more useful to measure the reciprocal of the period, in our case, $48000/14$ or about 3429 *cycles per second*. Another name for "cycles per second" is "Hertz," abbreviated Hz, so our sound has a component at 3249 Hz. As a musical note, that's about an A (a little flat), four octaves above middle C. (Don't worry too much about the note being a little off; remember that the 14-sample period was just eyeballed and is unlikely to be exactly right.)

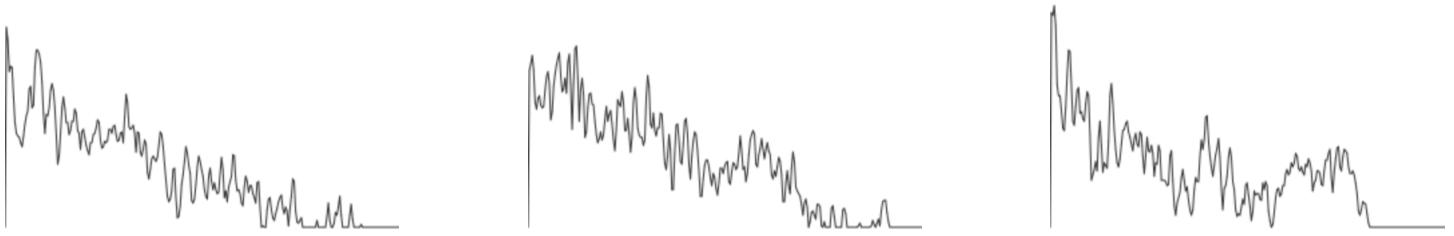
Four octaves above middle C is really high! That would be a shrill-sounding note. But remember that a complex waveform is the sum of multiple sine waves at different frequency. Here's a different up-and-down regularity:



It's not obvious, but in the left part of the graph, it's more above the x axis than below it. Toward the right, it seems to be more below than above the axis. At the very right it looks like it might be climbing again.

The period of the red sine wave is 340 samples, or $340/48000$ second. That's a frequency of about 141 Hz, about D below middle C. Again, this is measuring by eyeball, but likely to be close to the right frequency.

All this eyeballing doesn't seem very scientific. Can't we just get the computer to find all the relevant frequencies? Yes, we can, using a mathematical technique called *Fourier analysis*. (Jean-Baptiste Joseph Fourier, 1768–1830, made many contributions to mathematics and physics, but is best known for working out the nature of periodic functions as a sum of sine waves.) Luckily we don't have to do the math; the **microphone** block will do it for us, if we ask for **microphone spectrum**:



These are frequency spectra from (samples of) three different songs. The most obvious thing about these graphs is that their overall slope is downward; the loudest frequency is the lowest frequency. That's typical of music.

The next thing to notice is that there's a regularity in the spacing of spikes in the graph. This is partly just an artifact; the frequency (horizontal) axis isn't continuous. There are a finite number of "buckets" (default: 512), and all the frequencies within a bucket contribute to the amplitude (vertical axis) of that bucket. The spectrum is a list of that many amplitudes. But the patterns of alternating rising and falling values are real; the frequencies that are multiples of the main note being sampled will have higher amplitude than other frequencies.

Samples and **spectrum** are the two most detailed representations of a sound. But the **microphone** block has other, simpler options also:

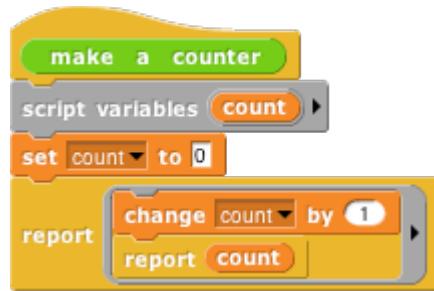
volume	the instantaneous volume when the block is called
note	the MIDI note number (as in play note) of the main note heard
frequency	the frequency in Hz of the main note heard
sample rate	the number of samples being collected per second
resolution	the size of the array in which data are collected (typically 512, must be a power of 2)

VIII. OOP with Procedures

The idea of object oriented programming is often taught in a way that makes it seem as if a special object oriented programming language is necessary. In fact, any language with first class procedures and lexical scope allows objects to be implemented explicitly; this is a useful exercise to help demystify objects.

The central idea of this implementation is that an object is represented as a *dispatch procedure* that takes a message as input and reports the corresponding method. In this section we start with a stripped-down example to show how local state works, and build up to full implementations of class-instance and prototyping OOP.

A. Local State with Script Variables



This script implements an object *class*, a type of object, namely the counter class. In this first simplified version there is only one method, so no explicit message passing is necessary. When the **make a counter** block is called, it reports a procedure, the ringed script inside its body. That procedure implements a specific counter object, an *instance* of the counter class. When invoked, a counter instance increases and reports its count variable. Each counter has its own local count:



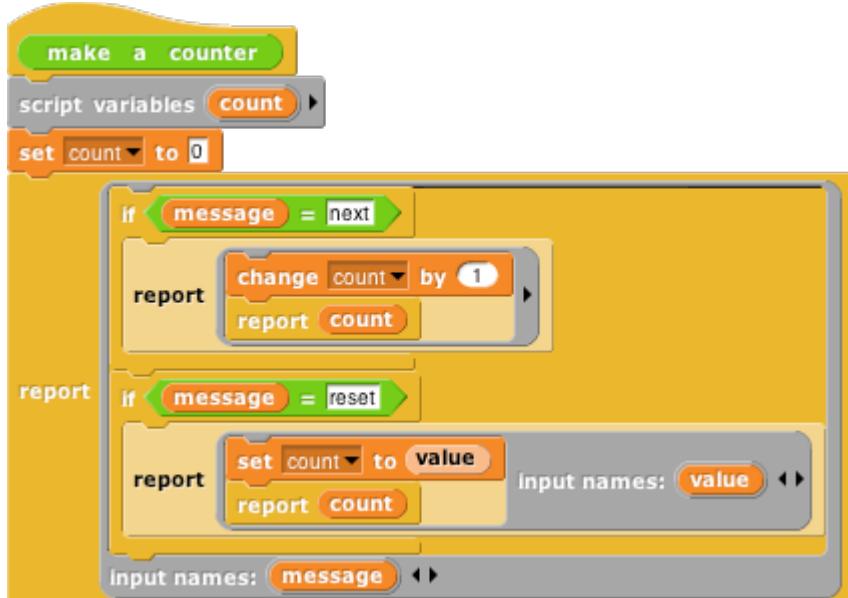
This example will repay careful study, because it isn't obvious why each instance has a separate count. From the point of view of the **make a counter** procedure, each invocation causes a new **count** variable to be created. Usually such *script variables* are temporary, going out of existence when the script ends. But this one is special, because **make a counter** returns *another script* that makes reference to the **count** variable, so it remains active.

(The **script variables** block makes variables local to a script. It can be used in a sprite's script area or in the Block Editor. Script variables can be “exported” by being used in a reported procedure, as here.)

In this approach to OOP, we are representing both classes and instances as procedures. The **make a counter** block represents the class, while each instance is represented by a nameless script created each time **make a counter** is called. The script variables created inside the **make a counter** block but outside the ring are *instance variables*, belonging to a particular counter.

B. Messages and Dispatch Procedures

In the simplified class above, there is only one method, and so there are no messages; you just call the instance to carry out its one method. Here is a more refined version that uses message passing:



Again, the **make a counter** block represents the **counter** class, and again the script creates a local variable **count** each time it is invoked. The large outer ring represents an instance. It is a *dispatch procedure*: it takes a **message** (just a text word) as input, and it reports a method. The two smaller rings are the methods. The top one is the **next** method; the bottom one is the **reset** method. The latter requires an input, named **value**.

In the earlier version, calling the instance did the entire job. In this version, calling the instance gives access to a method, which must then be called to finish the job. We can provide a block to do both procedure calls in one:

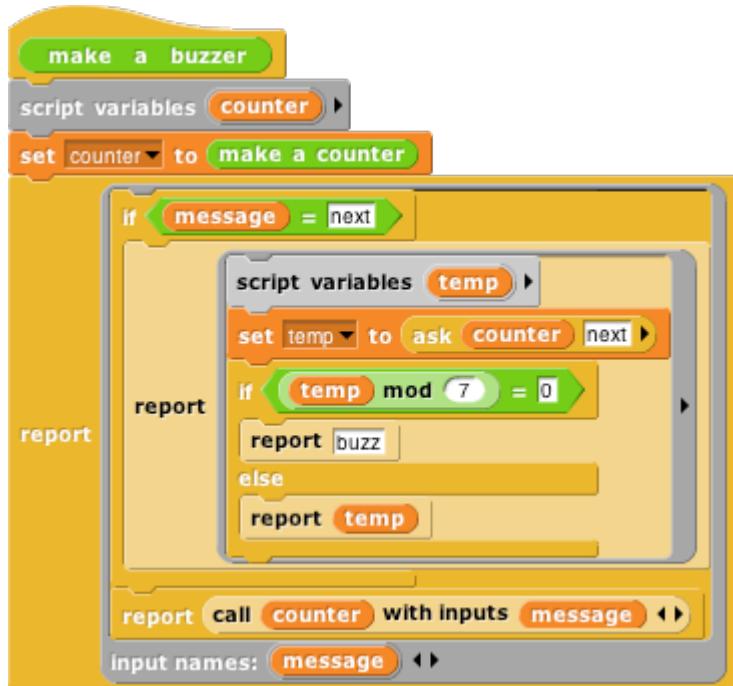


The **ask** block has two required inputs: an object and a message. It also accepts optional additional inputs, which **Snap!** puts in a list; that list is named **args** inside the block. **Ask** has two nested **call** blocks. The inner one calls the object, i.e., the dispatch procedure. The dispatch procedure always takes exactly one input, namely the message. It reports a method, which may take any number of inputs; note that this is the situation in which we drop a list of values onto the arrowheads of a multiple input (in the outer **call** block). Note also that this is one of the rare cases in which we must unringify the inner **call** block, whose *value when called* gives the method.



C. Inheritance via Delegation

So, our objects now have local state variables and message passing. What about inheritance? We can provide that capability using the technique of *delegation*. Each instance of the child class contains an instance of the parent class, and simply passes on the messages it doesn't want to specialize:

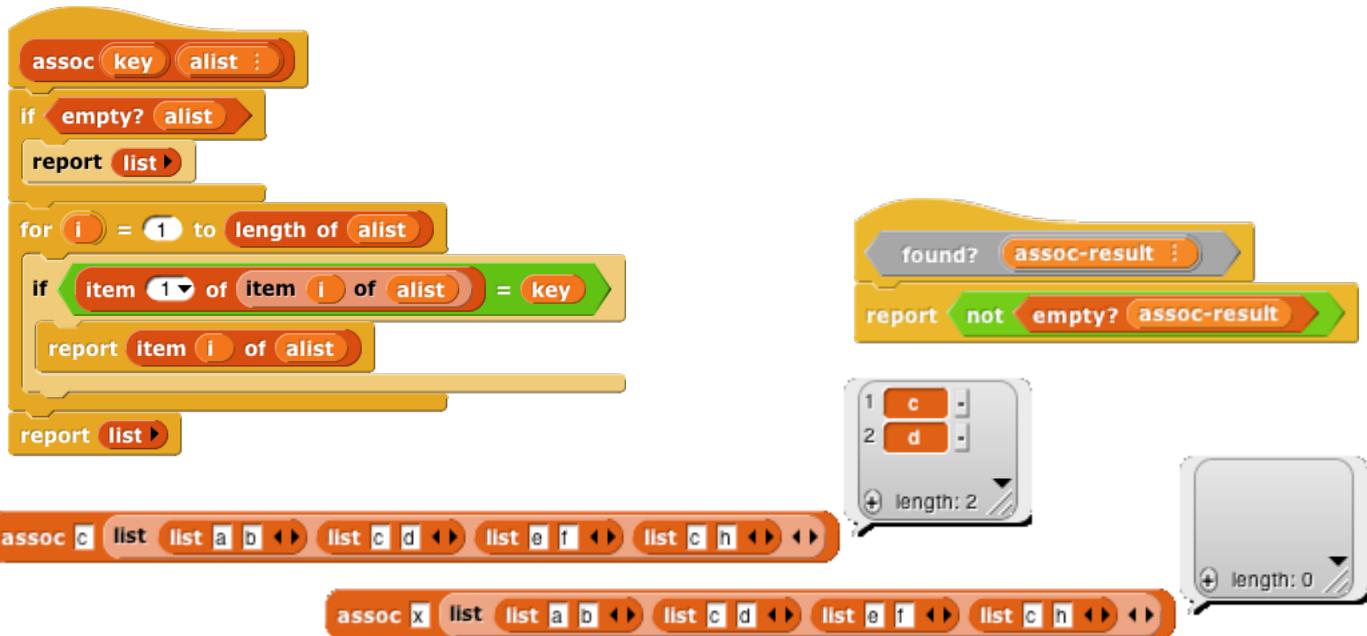


This script implements the **buzzer** class, which is a child of **counter**. Instead of having a count (a number) as a local state variable, each buzzer has a **counter** (an object) as a local state variable. The class specializes the **next** method, reporting what the counter reports unless that result is divisible by 7, in which case it reports “**buzz**.” (Yeah, it should also check for a digit 7 in the number, but this code is complicated enough already.) If the message is anything other than **next**, though, such as **reset**, then the buzzer simply invokes its **counter**'s dispatch procedure. So the counter handles any message that the buzzer doesn't handle explicitly. (Note that in the non-**next** case we **call** the counter, not **ask** it something, because we want to report a method, not the value that the message reports.) So, if we **ask** a **buzzer** to **reset** to a value divisible by 7, it will end up reporting that number, not “**buzz**.”

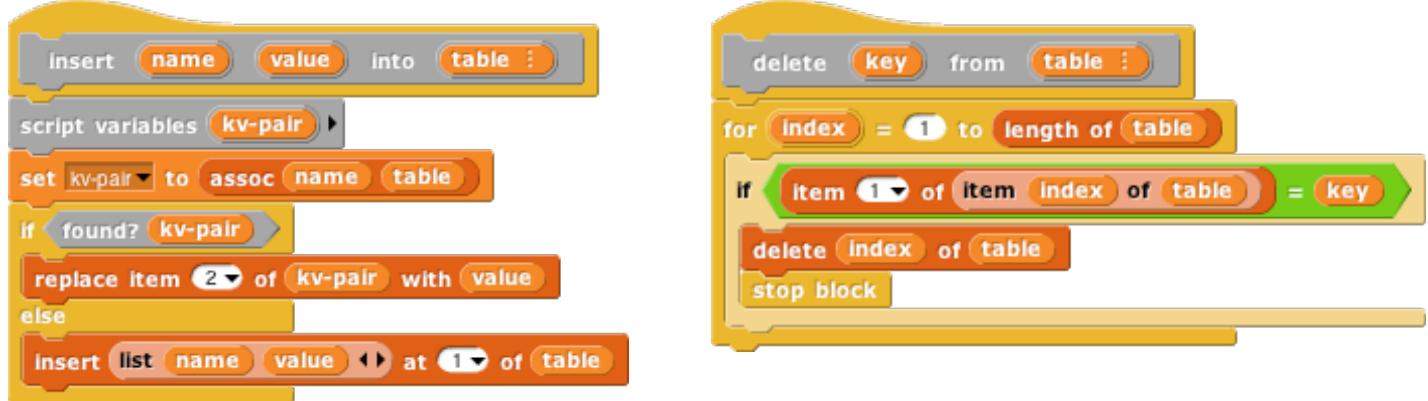
D. An Implementation of Prototyping OOP

In the class-instance system above, it is necessary to design the complete behavior of a class before you can make any instances of the class. This is okay for top-down design, but not great for experimentation. Here we sketch the implementation of a *prototyping* OOP system: You make an object, tinker with it, make clones of it, and keep tinkering. Any changes you make in the parent are inherited by its children. In effect, that first object is both the class and an instance of the class. In the implementation below, children share properties (methods and local variables) of their parent unless and until a child changes a property, at which point that child gets a private copy. (If a child wants to change something for its entire family, it must ask the parent to do it.)

Because we want to be able to create and delete properties dynamically, we won't use **Snap!** variables to hold an object's variables or methods. Instead, each object has two *tables*, called **methods** and **data**, each of which is an *association list*: a list of two-item lists, in which each of the latter contains a *key* and a corresponding *value*. We provide a lookup procedure to locate the key-value pair corresponding to a given key in a given table.



There are also commands to insert and delete entries:

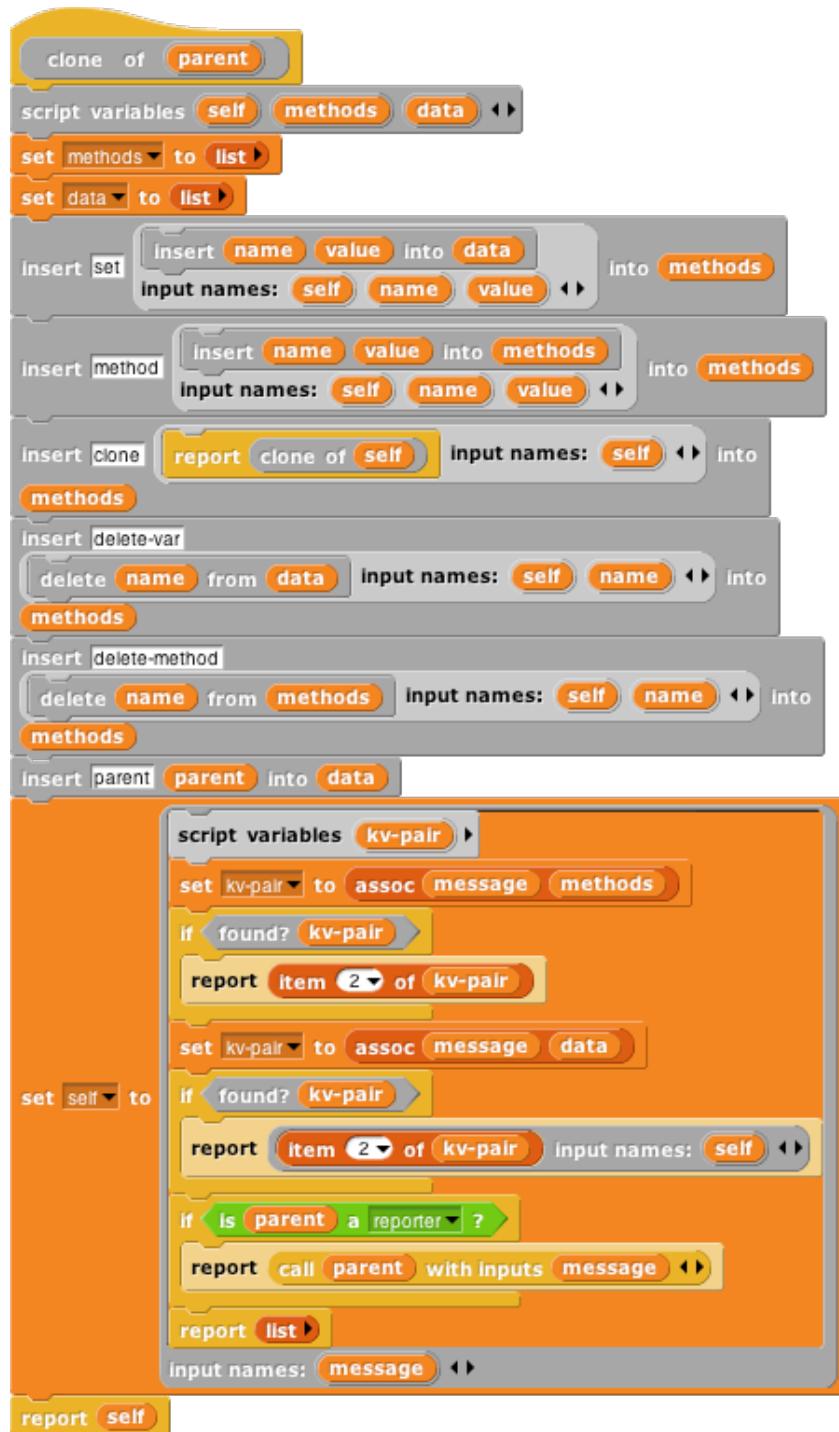


As in the class-instance version, an object is represented as a dispatch procedure that takes a message as its input and reports the corresponding method. When an object gets a message, it will first look for that keyword in its **methods** table. If it's found, the corresponding value is the method we want. If not, the object looks in its **data** table. If a value is found there, what the object returns is *not* that value, but rather a reporter method that, when called, will report the value. This means that what an object returns is *always* a method.

If the object has neither a method nor a datum with the desired name, but it does have a parent, then the parent (that is, the parent's dispatch procedure) is invoked with the message as its input. Eventually, either a match is found, or an object with no parent is found; the latter case is an error, meaning that the user has sent the object a message not in its repertoire.

Messages can take any number of inputs, as in the class-instance system, but in the prototyping version, every method automatically gets the object to which the message was originally sent as an extra first input. We must do this so that if a method is found in the parent (or grandparent, etc.) of the original recipient, and that method refers to a variable or method, it will use the child's variable or method if the child has its own version.

The **clone of** block below takes an object as its input and makes a child object. It should be considered as an internal part of the implementation; the preferred way to make a child of an object is to send that object a **clone** message.



Every object is created with predefined methods for **set**, **method**, **delete-var**, **delete-method**, and **clone**. It has one predefined variable, **parent**. Objects without a parent are created by calling **new object**:



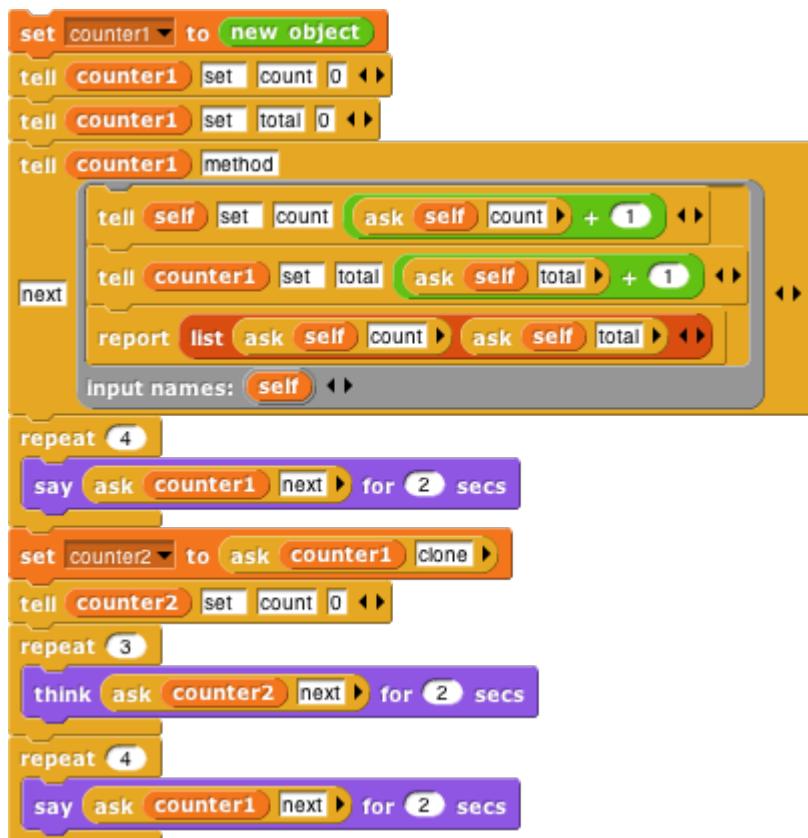
As before, we provide procedures to call an object's dispatch procedure and then call the method. But in this version, we provide the desired object as the first method input. We provide one procedure for Command methods and one for Reporter methods:



(Remember that the “**Input list:**” variant of the **run** and **call** blocks is made by dragging the input expression over the arrowheads rather than over the input slot.)

The script below demonstrates how this prototyping system can be used to make counters. We start with one prototype counter, called **counter1**. We count this counter up a few times, then create a child **counter2** and give it its own **count** variable, but *not* its own **total** variable. The **next** method always sets **counter1**'s **total** variable, which therefore keeps count of the total number of times that *any* counter is incremented. Running this script should [**say**] and [**think**] the following lists:

[1 1] [2 2] [3 3] [4 4] (1 5) (2 6) (3 7) [5 8] [6 9] [7 10] [8 11]



IX. The Outside World

The facilities discussed so far are fine for projects that take place entirely on your computer's screen. But you may want to write programs that interact with physical devices (sensors or robots) or with the World Wide Web. For these purposes **Snap!** provides a single primitive block:

url snap.berkeley.edu

This might not seem like enough, but in fact it can be used to build the desired capabilities.

A. The World Wide Web

The input to the **url** block is the URL (Uniform Resource Locator) of a web page. The block reports the body of the Web server's response (minus HTTP header), *without interpretation*. This means that in most cases the response is a description of the page in HTML (HyperText Markup Language) notation. Often, especially for commercial web sites, the actual information you're trying to find on the page is actually at another URL included in the reported HTML. The Web page is typically a very long text string, and so the primitive **split** block is useful to get the text in a manageable form, namely, as a list of lines:



The screenshot shows a list of numbered lines of code in a scrollable window. The code is primarily HTML, including DOCTYPE, head, meta, title, and link tags. Lines 13 through 22 contain JavaScript code for Google Analytics tracking. The status bar at the bottom right indicates a length of 384.

```
1 <!DOCTYPE html>
2 <!-- Woot, not HTML2! -->
3 <html><head>
4 <meta http-equiv="content-type" content="text/html; charset=utf-8">
5 <title>Snap! (Build Your Own Blocks) 4.0</title>
6 <meta coding="utf-8">
7 <link rel="shortcut icon" href="snapsource/favicon.ico">
8     <link href="liam4/bootstrap.css" rel="stylesheet">
9     <link rel="stylesheet" href="liam4/index.css">
10
11 <script type="text/javascript">
12
13 var _gaq = _gaq || [];
14 _gaq.push(['_setAccount', 'UA-30925559-2']);
15 _gaq.push(['_trackPageview']);
16
17 (function() {
18     var ga = document.createElement('script'); ga.type = 'text/javascript'; ga.as
19     ga.src = ('https:' == document.location.protocol ? 'https://ssl' : 'http://www')
20     '.google-analytics.com/ga.js';
21     var s = document.getElementsByTagName('script')[0]; s.parentNode.insertE
22 })();
23
```

length: 384

split url snap.berkeley.edu by line ▾

The second input to **split** is the character to be used to separate the text string into a list of lines, or one of a set of common cases (such as **line**, which separates on carriage return and/or newline characters).

This might be a good place for a reminder that list-view watchers scroll through only 100 items at a time. The downarrow near the bottom right corner of the speech balloon in the picture presents a menu of hundred-item ranges. (This may seem unnecessary, since the scroll bar should allow for any number of items, but doing it this way makes **Snap!** much faster.) In table view, the entire list is included.

If you include a protocol name in the input to the `url` block (such as `http://` or `https://`), that protocol will be used. If not, the block first tries HTTPS and then, if that fails, HTTP.

A security restriction in JavaScript limits the ability of one web site to initiate communication with another site. There is an official workaround for this limitation called the CORS protocol (Cross-Origin Resource Sharing), but the target site has to allow `snap.berkeley.edu` explicitly, and of course most don't. To get around this problem, you can use third-party sites ("cors proxies") that are not limited by JavaScript and that forward your requests.

B. Hardware Devices

Another JavaScript security restriction prevents `Snap!` from having direct access to devices connected to your computer, such as sensors and robots. (Mobile devices such as smartphones may also have useful devices built in, such as accelerometers and GPS receivers.) The `url` block is also used to interface `Snap!` with these external capabilities.

The idea is that you run a separate program that both interfaces with the device and provides a local HTTP server that `Snap!` can use to make requests to the device. *Unlike Snap! itself, these programs have access to anything on your computer, so you have to trust the author of the software!* Our web site, `snap.berkeley.edu`, provides links to drivers for several devices, including, at this writing, the Lego NXT, Finch, Hummingbird, and Parallax S2 robots; the Nintendo Wiimote and Leap Motion sensors, the Arduino microcomputer, and Super-Awesome Sylvia's Water Color Bot. The same server technique can be used for access to third party software libraries, such as the speech synthesis package linked on our web site.

Most of these packages require some expertise to install; the links are to source code repositories. This situation will improve with time.

C. Date and Time

The `current` block in the Sensing palette can be used to find out the current date or time. Each call to this block reports one component of the date or time, so you will probably combine several calls, like this:



for Americans, or like this:

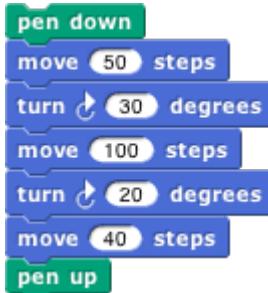


for Europeans.

X. Continuations

Blocks are usually used within a script. The *continuation* of a block within a particular script is the part of the computation that remains to be completed after the block does its job. A continuation can be represented as a ringed script. Continuations are always part of the interpretation of any program in any language, but usually these continuations are implicit in the data structures of the language interpreter or compiler. Making continuations explicit is an advanced but versatile programming technique that allows users to create control structures such as nonlocal exit and multithreading.

In the simplest case, the continuation of a command block may just be the part of the script after the block. For example, in the script



the continuation of the **move 100 steps** block is



But some situations are more complicated. For example, what is the continuation of **move 100 steps** in the following script?



That's a trick question; the **move** block is run four times, and it has a different continuation each time. The first time, its continuation is

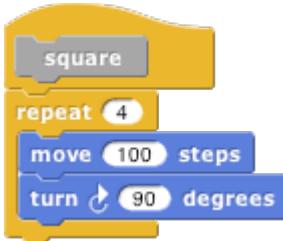


Note that there is no **repeat 3** block in the actual script, but the continuation has to represent the fact that there are three more times through the loop to go. The fourth time, the continuation is just



What counts is not what's physically below the block in the script, but what computational work remains to be done.

When a block is used inside a custom block, its continuation may include parts of more than one script. For example, if we make a custom **square** block



and then use that block in a script:



then the continuation of the first use of **move 100 steps** is



in which part comes from inside the **square** block and part comes from the call to **square**. Nevertheless, ordinarily when we *display* a continuation we show only the part within the current script.

The continuation of a command block, as we've seen, is a simple script with no input slots. But the continuation of a reporter block has to do something with the value reported by the block, so it takes that value as input. For example, in the script



the continuation of the **3+4** block is



Of course the name **result** in that picture is arbitrary; any name could be used, or no name at all by using the empty-slot notation for input substitution.

A. Continuation Passing Style

Like all programming languages, **Snap!** evaluates compositions of nested reporters from the inside out. For example, in the expression **(3 × (4 + 5))** **Snap!** first adds 4 and 5, then multiplies 3 by that sum. This often means that the order in which the operations are done is backwards from the order in which they appear in the expression: When reading the above expression you say “times” before you say “plus.” In English, instead of saying “three times four plus five,” which actually makes the order of operations ambiguous, you could say, “take the sum of four and five, and then take the product of three and that sum.” This sounds more awkward, but it has the virtue of putting the operations in the order in which they're actually performed.

That may seem like overkill in a simple expression, but suppose you're trying to convey the expression

```
factorial 3 × factorial [2 + 2 + 5]
```

to a friend over the phone. If you say “factorial of three times factorial of two plus two plus five” you might mean any of these:

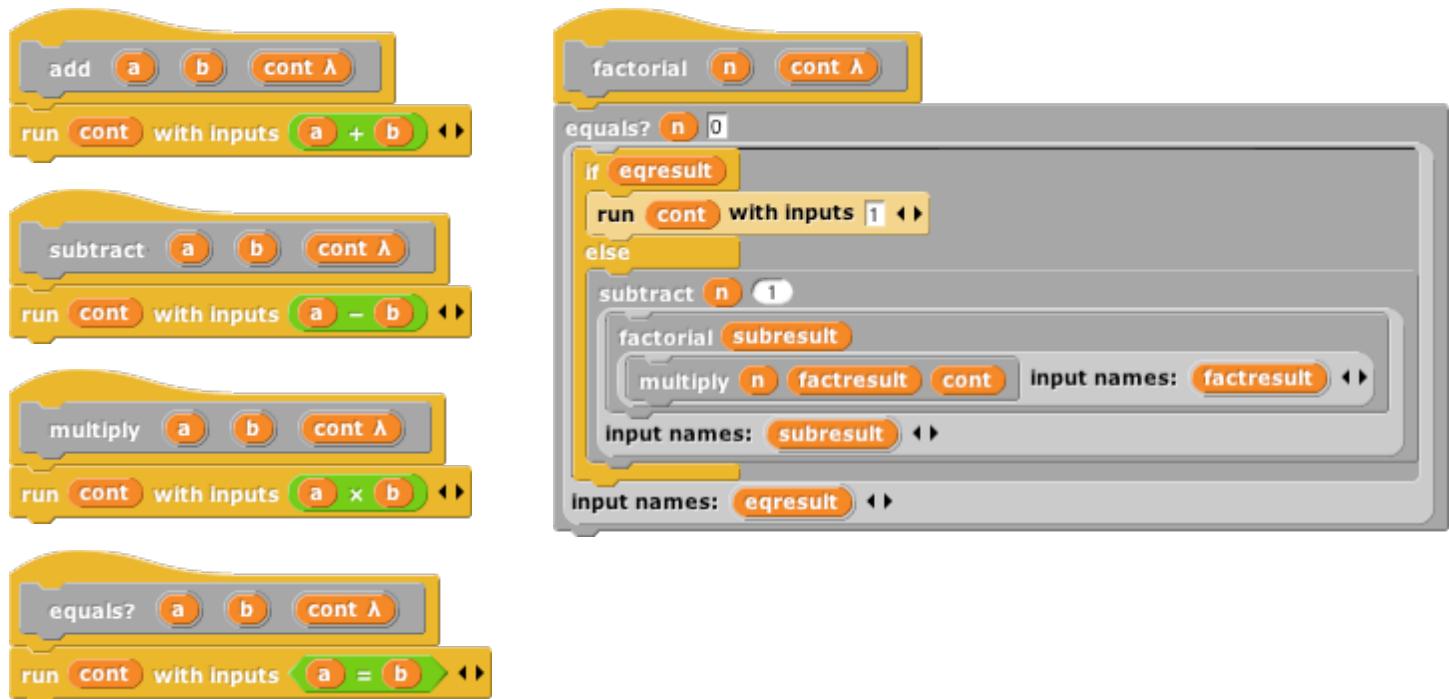
```
factorial 3 × factorial [2 + 2 + 5]
```

```
factorial 3 × factorial [2] + [2 + 5]
```

```
factorial 3 × factorial [2] + [2 + 5]
```

```
factorial 3 × factorial [2] + [2 + 5]
```

Wouldn't it be better to say, “Add two and two, take the factorial of that, add five to that, multiply three by that, and take the factorial of the result”? We can do a similar reordering of an expression if we first define versions of all the reporters that take their continuation as an explicit input. In the following picture, notice that the new blocks are *commands*, not reporters.



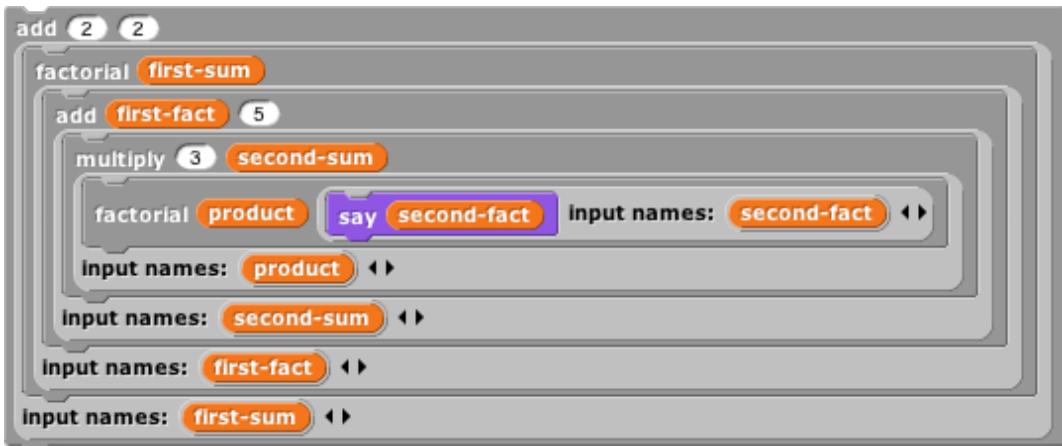
We can check that these blocks give the results we want:

```
factorial 5 say [ ]
```



120

The original expression can now be represented as



If you read this top to bottom, don't you get "Add two and two, take the factorial of that, add five to that, multiply three by that, and take the factorial of the result"? Just what we wanted! This way of working, in which every block is a command that takes a continuation as one of its inputs, is called *continuation-passing style (CPS)*. Okay, it looks horrible, but it has subtle virtues. One of them is that each script is just one block long (with the rest of the work buried in the continuation given to that one block), so each block doesn't have to remember what else to do — in the vocabulary of this section, the (implicit) continuation of each block is empty. Instead of the usual picture of recursion, with a bunch of little people all waiting for each other, with CPS what happens is that each little person hands off the problem to the next one and goes to the beach, so there's only one active little person at a time. In this example, we start with Alfred, an **add** specialist, who computes the value 4 and then hands off the rest of the problem to Francine, a **factorial** specialist. She computes the value 24, then hands the problem off to Anne, another **add** specialist, who computes 29. And so on, until finally Sam, a **say** specialist, says the value $2.107757298379527 \times 10^{132}$, which is a very large number!



Go back to the definitions of these blocks. The ones, such as **add**, that correspond to primitive reporters are simple; they just call the reporter and then call their continuation with its result. But the definition of **factorial** is more interesting. It doesn't just call our original **factorial** reporter and send the result to its continuation. CPS is used inside **factorial** too! It says, "See if my input is zero. Send the (true or false) result to **if**. If the result is **true**, then call my continuation with the value 1. Otherwise, subtract 1 from my input. Send the result of that to **factorial**, with a continuation that multiplies the smaller number's factorial by my original input. Finally, call my continuation with the product." You can use CPS to unwind even the most complicated branched recursions.

By the way, I cheated a bit above. The **if/else** block should also use CPS; it should take one true/false input and *two continuations*. It will go to one or the other continuation depending on the value of its input. But in fact the C-shaped blocks (or E-shaped, like **if/else**) are really using CPS in the first place, because they implicitly wrap rings around the sub-scripts within their branches. See if you can make an explicitly CPS **if/else** block.

B. Call/Run w/Continuation

To use explicit continuation passing style, we had to define special versions of all the reporters, **add** and so on. **Snap!** provides a primitive mechanism for capturing continuations when we need to, without using continuation passing throughout a project.

Here's the classic example. We want to write a recursive block that takes a list of numbers as input, and reports the product of all the numbers:



But we can improve the efficiency of this block, in the case of a list that includes a zero; as soon as we see the zero, we know that the entire product is zero.



But this is not as efficient as it might seem. Consider, as an example, the list 1,2,3,0,4,5. We find the zero on the third recursive call (the fourth call altogether), as the first item of the sublist 0,4,5. What is the continuation of the **report 0** block? It's



Even though we already know that **result** is zero, we're going to do three unnecessary multiplications while unwinding the recursive calls.

We can improve upon this by capturing the continuation of the top-level call to **product**:



The  **w/continuation** block takes as its input a one-input script, as shown in the **product** example. It calls that script with *the continuation of the call-with-continuation block itself* as its input. In this case, that continuation is



reporting to whichever script called **product**. If the input list doesn't include a zero, then nothing is ever done with that continuation, and this version works just like the original **product**. But if the input list is 1,2,3,0,4,5, then three recursive calls are made, the zero is seen, and **product-helper runs the continuation**, with an input of 0. The continuation immediately reports that 0 to the caller of **product**, *without* unwinding all the recursive calls and without the unnecessary multiplications.

I could have written **product** a little more simply using a Reporter ring instead of a Command ring:



but it's customary to use a script to represent the input to **call w/continuation** because very often that input takes the form



so that the continuation is saved permanently and can be called from anywhere in the project. That's why the input slot in **call w/continuation** has a Command ring rather than a Reporter ring.

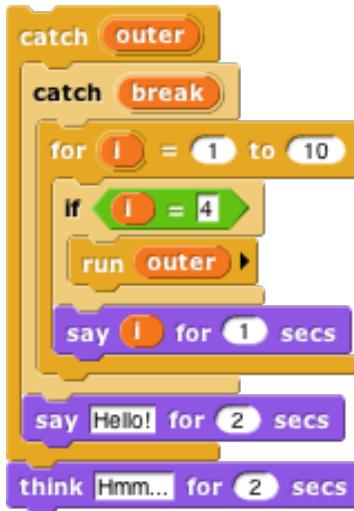
First class continuations are an experimental feature in **Snap!** and there are many known limitations in it. One is that it doesn't work to run an empty continuation; that is, if a **run w/continuation** block is the last one in a script, the continuation will fail. Also, the display of reporter continuations shows only the single block in which the **call w/continuation** is an input.

Nonlocal exit

Many programming languages have a **break** command that can be used inside a looping construct such as **repeat** to end the repetition early. Using first class continuations, we can generalize this mechanism to allow nonlocal exit even within a block called from inside a loop, or through several levels of nested loops:¹



The upvar **break** has as its value a continuation that can be called from anywhere in the program to jump immediately to whatever comes after the **catch** block in its script. Here's an example with two nested invocations of **catch**, with the upvar renamed in the outer one:



As shown, this will say 1, then 2, then 3, then exit both nested **catches** and think "Hmm." If in the **run** block the variable **break** is used instead of **outer**, then the script will say 1, 2, 3, and "Hello!" before thinking "Hmm."

There are corresponding **catch** and **throw** blocks for reporters. The **catch** block is a reporter that takes an expression as input instead of a C-shaped slot. But the **throw** block is a command; it doesn't report a value to its own continuation, but instead reports a value (which it takes as an additional input, in addition to the **catch** tag) to the corresponding **catch** block's continuation:



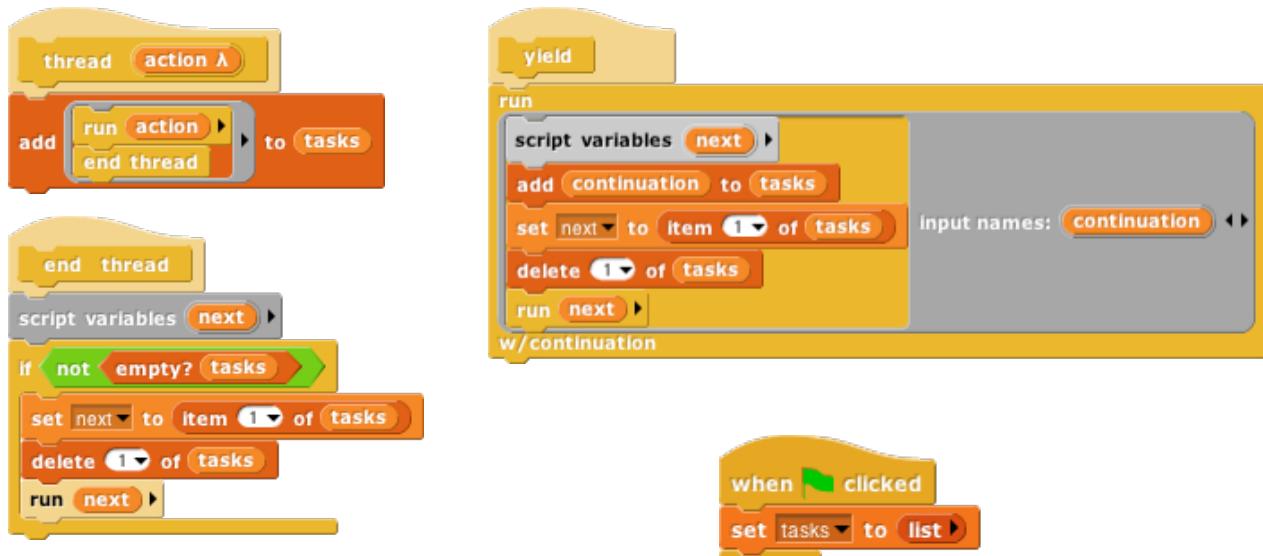
Without the **throw**, the inner **call** reports 5, the **+** block reports 8, so the **catch** block reports 8, and the **×** block reports 80. With the **throw**, the inner **call** doesn't report at all, and neither does the **+** block. The **throw** block's input of 20 becomes the value reported by the **catch** block, and the **×** block multiplies 10 and 20.

¹ As of now there has to be another command that does nothing to end the **catch** definition, because of the bug mentioned earlier. These blocks are found in the Iteration/Composition library.

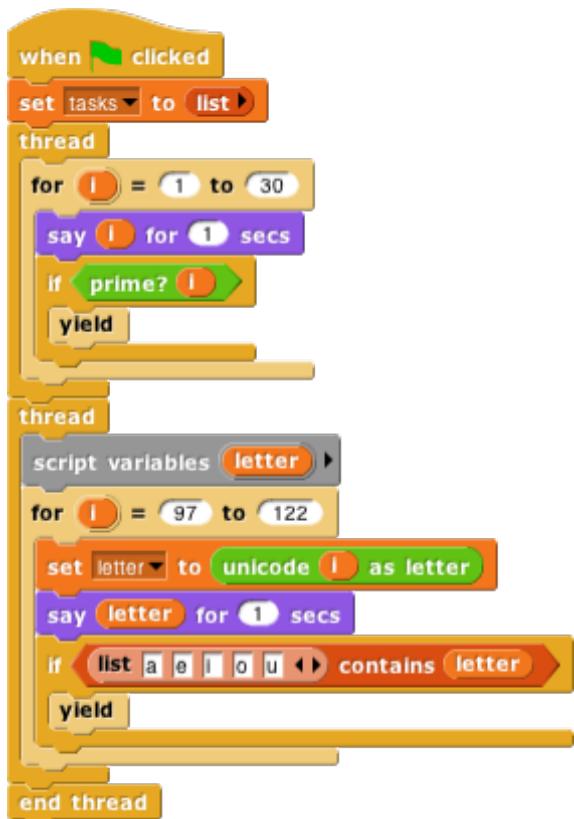
Creating a Thread System

Snap! can be running several scripts at once, within a single sprite and across many sprites. If you only have one computer, how can it do many things at once? The answer is that only one is actually running at any moment, but **Snap!** switches its attention from one script to another frequently. At the bottom of every looping block (**repeat**, **repeat until**, **forever**), there is an implicit “yield” command, which remembers where the current script is up to, and switches to some other script, each in turn. At the end of every script is an implicit “end thread” command (a *thread* is the technical term for the process of running a script), which switches to another script without remembering the old one.

Since this all happens automatically, there is generally no need for the user to think about threads. But, just to show that this, too, is not magic, here is an implementation of a simple thread system. It uses a global variable named **tasks** that initially contains an empty list. Each use of the C-shaped **thread** block adds a continuation (the ringed script) to the list. The **yield** block uses **run w/continuation** to create a continuation for a partly done thread, adds it to the task list, and then runs the first waiting task. The **end thread** block (which is automatically added at the end of every thread’s script by the **thread** block) just runs the next waiting task.



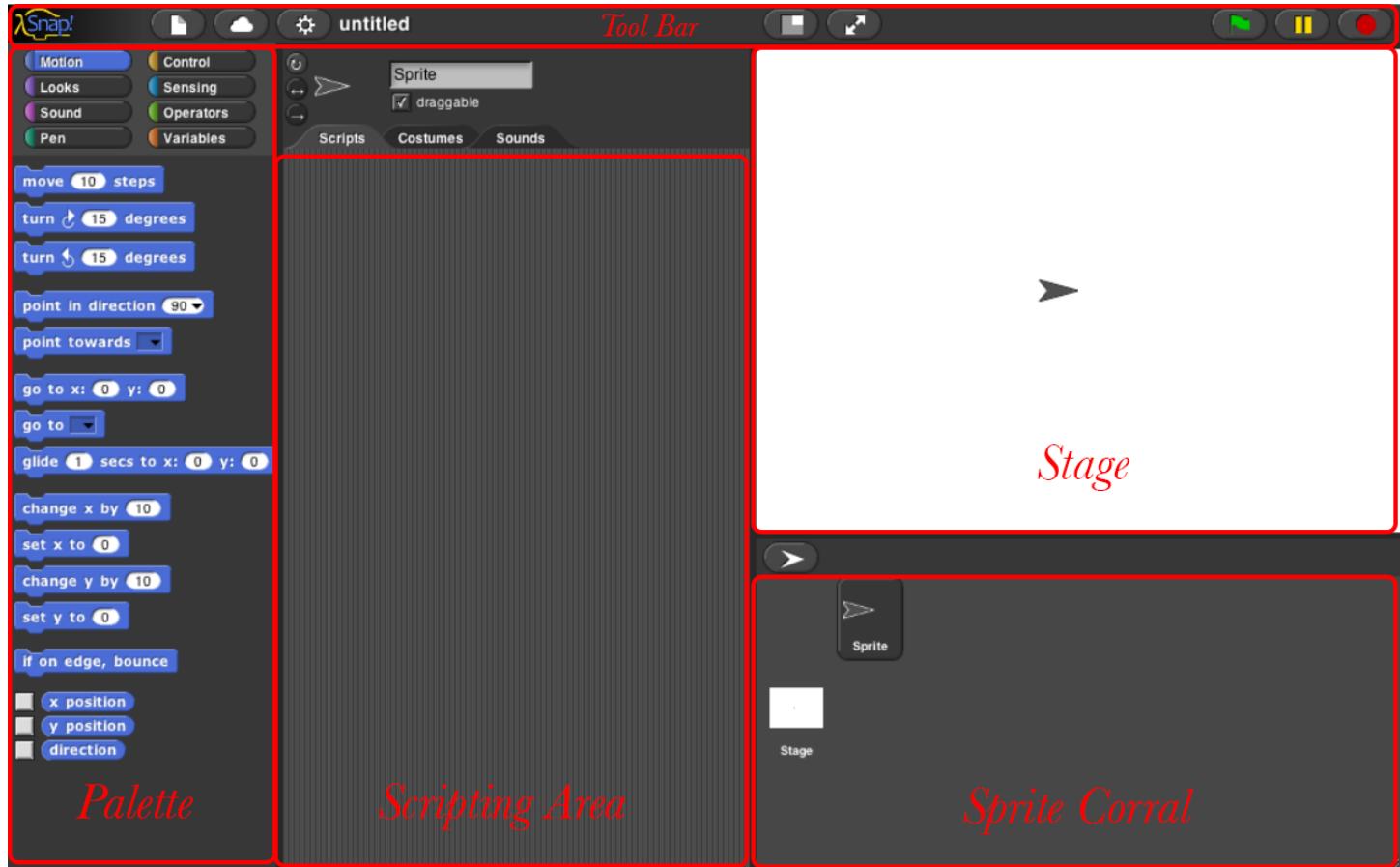
Here is a sample script using the thread system. One thread **says** numbers; the other **says** letters. The number thread yields after every prime number, while the letter thread yields after every vowel. So the sequence of speech balloons is 1,2,a,3,b,c,d,e,4,5,f,g,h,i,6,7,j,k,l,m,n,o,8,9,10,11, p,q,r,s,t,u,12,13,v,w,x,y,z,14,15,16,17,18,...30.



If we wanted this to behave exactly like **Snap!**’s own threads, we’d define new versions of **repeat** and so on that run **yield** after each repetition.

XI. User Interface Elements

In this section we describe in detail the various buttons, menus, and other clickable elements of the **Snap!** user interface. Here again is the map of the **Snap!** window:

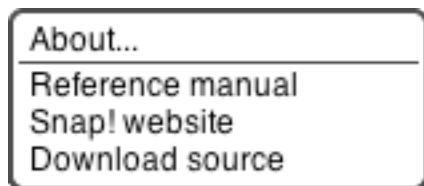


A. Tool Bar Features

Holding down the Shift key while clicking on any of the menu buttons gives access to an extended menu with options, shown in red, that are experimental or for use by the developers. We're not listing those extra options here because they change frequently and you shouldn't rely on them. But they're not secrets.

The **Snap!** Logo Menu

The **Snap!** logo at the left end of the tool bar is clickable. It shows a menu of options about **Snap!** itself:



The **About** option displays information about **Snap!** itself, including version numbers for the source modules, the implementors, and the license (AGPL: you can do anything with it except create proprietary versions, basically).

The **Reference manual** option downloads a copy of the latest revision of this manual in PDF format.

The **Snap! website** option opens a browser window pointing to `snap.berkeley.edu`, the web site for Snap!.

The **Download source** option opens a browser window displaying the Github repository of the source files for Snap!. At the bottom of the page are links to download the latest official release. Or you can navigate around the site to find the current development version. You can read the code to learn how Snap! is implemented, host a copy on your own computer (this is one way to keep working while on an airplane), or make a modified version with customized features. (However, access to cloud accounts is limited to the official version hosted at Berkeley.)

The File Menu

The file  icon shows a menu mostly about saving and loading projects:



The **Project notes** option opens a window in which you can type notes about the project: How to use it, what it does, whose project you modified to create it, if any, what other sources of ideas you used, or any other information about the project. This text is saved with the project, and is useful if you share it with other users.

The **New** option starts a new, empty project. Any project you were working on before disappears, so you are asked to confirm that this is really what you want. (It disappears only from the current working Snap! window; you should save the current project, if you want to keep it, before using **New**.)

Note the **^N** at the end of the line. This indicates that you can type control-N as a shortcut for this menu item. Alas, this is not the case in every browser. Some Mac browsers require command-N (**⌘N**) instead, while others open a new browser window instead of a new project. You'll have to experiment. In general, the keyboard shortcuts in Snap! are the standard ones you expect in other software.

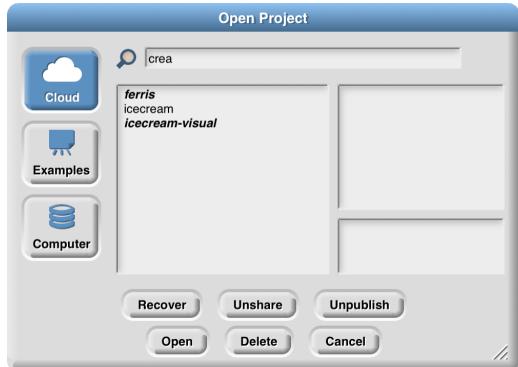
The **Open...** option shows a project open dialog box in which you can choose a project to open:



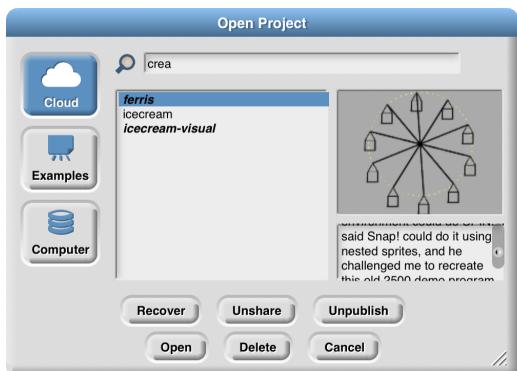
In this dialog, the three large buttons at the left select a source of projects: **Cloud** means your Snap! account's cloud storage. **Examples** means a collection of sample projects we provide. **Computer** is for projects saved on your own computer; when you click it, this dialog is replaced with your computer's system dialog for opening

files. The text box to the right of those buttons is an alphabetical listing of projects from that source; selecting a project by clicking shows its thumbnail (a picture of the stage when it was saved) and its project notes at the right.

The search bar at the top can be used to find a project by name or text in the project notes. So in this example:



I was looking for my ice cream projects and typed “crea” in the search bar, then wondered why “ferris” matched. But then when I clicked on ferris I saw this:



My search matched the word “recreate” in the project notes.

The six buttons at the bottom select an action to perform on the selected project. In the top row, **Recover** looks in your cloud account for older versions of the chosen project. **If your project is damaged, don't keep saving broken versions! Use Recover first thing.** You will see a list of saved versions; choose one to open it. Typically, you'll see the most recent version before the last save, and the newest version saved before today. Then come buttons **Share/Unshare** and **Publish/Unpublish**. The labelling of the buttons depends on your project's publication status. If a project is neither shared nor published (the ones in lightface type in the project list), it is private and nobody can see it except you, its owner. If it is shared (**boldface** in the project list), then when you open it you'll see a URL like this one:

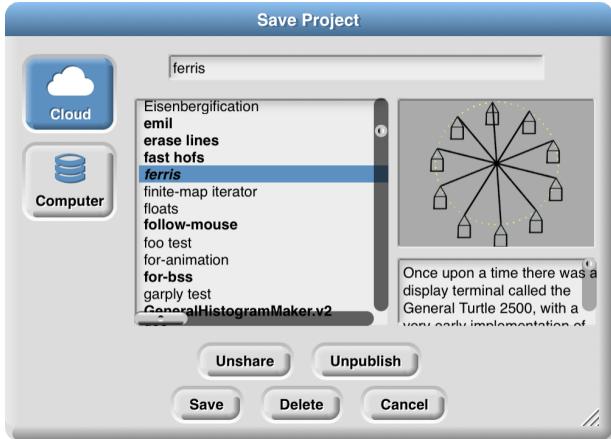
<https://snap.berkeley.edu/snapsource/snap.html#present:Username=bh&ProjectName=count%20change>
but with your username and project name. (“%20” in the project name represents a space, which can't be part of a URL.) Anyone who knows this URL can see your project. Finally, if your project is published (**bold italic** in the list), then your project is shown on the **Snap!** web site for all the world to see. (In all of these cases, you are the only one who can *write* to (save) your project. If another user saves it, a separate copy will be saved in that user's account. Projects remember the history of who created the original version and any other “remix” versions along the way.

In the second row, the first button, **Open**, loads the project into **Snap!** and closes the dialog box. The next button (if **Cloud** is the source) is **Delete**, and if clicked it deletes the selected project. Finally, the **Cancel** button

closes the dialog box without opening a project. (It does not undo any sharing, unsharing, or deletion you've done.)

Back to the File menu, the **Save** menu option saves the project to the same source and same name that was used when opening the project. (If you opened another user's shared project or an example project, the project will be saved to your own cloud account. You must be logged in to save to the cloud.)

The **Save as...** menu option opens a dialog box in which you can specify where to save the project:



This is much like the **Open** dialog, except for the horizontal text box at the top, into which you type a name for the project. You can also publish, unpublish, share, unshare, and delete projects from here. There is no **Recover** button.

The **Import...** menu option is for bringing some external resource into the current project, or it can load an entirely separate project, from your local disk. You can import costumes (any picture format that your browser supports), sounds (again, any format supported by your browser), and block libraries or sprites (XML format, previously exported from **Snap!** itself). Imported costumes and sounds will belong to the currently selected sprite; imported blocks are global (for all sprites). Using the **Import** option is equivalent to dragging the file from your desktop onto the **Snap!** window.

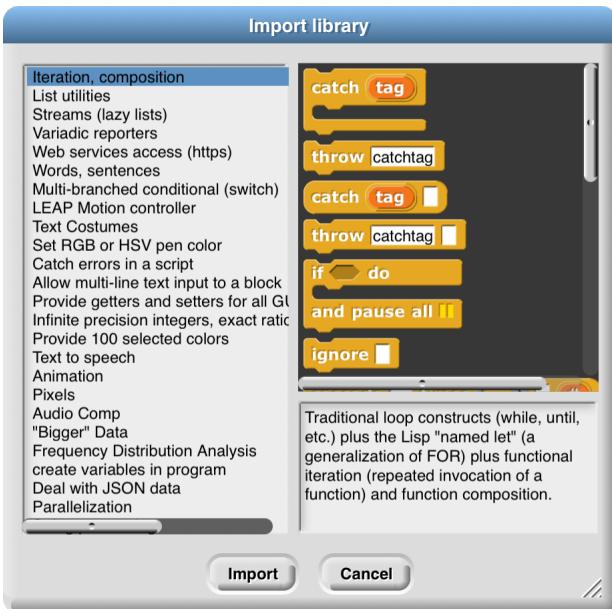
Depending on your browser, the **Export project...** option either directly saves to your disk or opens a new browser tab containing your complete project in XML notation (a plain text format). You can then use the browser's Save feature to save the project as an XML file, which should be named `something.xml` so that **Snap!** will recognize it as a project when you later drag it onto a **Snap!** window. This is an alternative to saving the project to your cloud account: keeping it on your own computer. It is equivalent to choosing **Computer** from the Save dialog described earlier.

The **Export blocks...** option is used to create a block library. It presents a list of all the global (for all sprites) blocks in your project, and lets you select which to export. It then opens a browser tab with those blocks in XML format, or stores directly to your local disk, as with the **Export project** option. Block libraries can be imported with the **Import** option or by dragging the file onto the **Snap!** window. This option is shown only if you have defined custom blocks.

The **Unused blocks...** option presents a listing of all the global blocks in your project that aren't used anywhere, and offers to delete them. As with Export blocks, you can choose a subset to delete with checkboxes. This option is shown only if you have defined custom blocks.

The **Export summary...** option creates a web page, in HTML, with all of the information about your project: its name, its project notes, a picture of what's on its stage, definitions of global blocks, and then per-sprite information: name, wardrobe (list of costumes), and local variables and block definitions. The page can be converted to PDF by the browser; it's intended to meet the documentation requirements of the Advanced Placement Computer Science Principles create task.

The **Libraries...** option presents a menu of useful, optional block libraries:



When you click on the one-line description of a library, you are shown the actual blocks in the library and a longer explanation of its purpose. You can browse the libraries to find one that will satisfy your needs.

The libraries and their contents may change, but as of this writing the iteration library has these blocks:



Catch and **throw** provide a nonlocal exit facility. You can drag the **tag** from a **catch** block to a **throw** inside its C-slot, and the **throw** will then jump directly out to the matching catch without doing anything in between.

If do and pause all is for setting a breakpoint while debugging code. The idea is to put **show variable** blocks for local variables in the C-slot; the watchers will be deleted when the user continues from the pause.

Ignore is used when you need to call a reporter but you don't care about the value it reports. (For example, you are writing a script to time how long the reporter takes.)

The **cascade** blocks take an initial value and call a function repeatedly on that value, $f(f(f(\dots(x))))$.

The **compose** block takes two functions and reports the function $f(g(x))$.

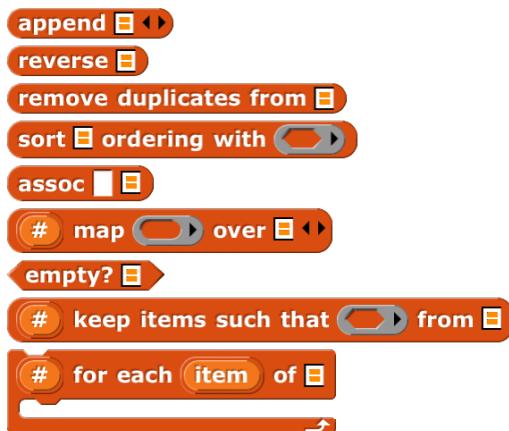
The first three **repeat** blocks are variants of the primitive **repeat until** block, giving all four combinations of whether the first test happens before or after the first repetition, and whether the condition must be true or false to continue repeating. The last **repeat** block is like the **repeat** primitive, but makes the number of repetitions so far available to the repeated script. The next two blocks are variations

on **for**: the first allows an explicit step instead of using ± 1 , and the second allows any values, not just numbers; inside the script you say



replacing the grey block in the picture with an expression to give the next desired value for the loop index.

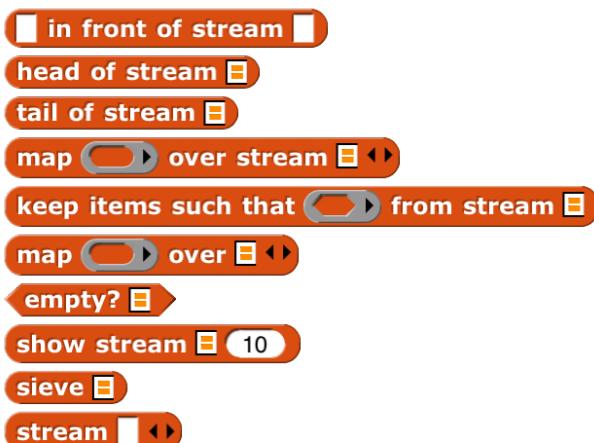
The list library has these blocks:



The **append** block takes any number of list inputs and reports a list of all the items of all the input lists. **Reverse** reports a list with the items of the input list in reverse order. **Remove duplicates from** reports a list in which no two items are equal. The **sort** block takes a list and a two-input comparison predicate, such as $<$, and reports a list with the items sorted according to that comparison. The **assoc** block is for looking up a key in an *association list*: a list of two-item lists. In each two-item list, the first is a *key* and the second is a *value*. The inputs are a key and an association list; the block reports the first key-value pair whose key is equal to the input key. **Empty?** is a helper for the remaining three blocks.

The other three blocks are versions of the list tools that provide a **#** variable containing the position in the input list of the currently considered item. This version of **map** also allows multiple list inputs, in which case the mapping function must take as many inputs as there are lists; it will be called with all the first items, all the second items, and so on.

The stream library has these blocks:



Streams are a special kind of list whose items are not computed until they are needed. This makes certain computations more efficient, and also allows the creation of lists with infinitely many items, such as a list of all the positive integers. The first five blocks are stream versions of the list blocks **in front of**, **item 1 of**, **all but first of**, **map**, and **keep**. The next two blocks, **map** and **empty?**, are the same ones in the tools library, included here because they are used in implementing the stream functions. **Show stream** takes a stream and a number as inputs, and reports an ordinary list of the first n items of the stream. **Sieve** is an example block that takes as input the stream of integers starting with 2 and reports the stream of all the prime numbers. **Stream** is like the primitive **list**; it makes a finite stream from explicit items.

The variadic library has these blocks:



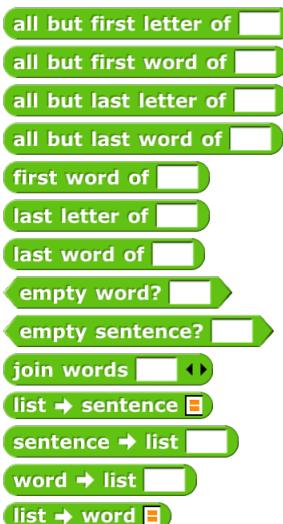
These are versions of the associative operators **+**, **\times** , **and**, and **or** that take any number of inputs instead of exactly two inputs. As with any variadic input, you can also drop a list of values onto the arrowheads instead of providing the inputs one at a time.

The web services library has these blocks:



The first block is a generalization of the primitive **url** block, allowing more control over the various options in web requests: GET, POST, PUT, and DELETE, and fine control over the content of the message sent to the server. **Current location** reports your latitude and longitude. The **key:value:** block is just a constructor for an abstract data type used with the other blocks.

The word and sentence library has these blocks:



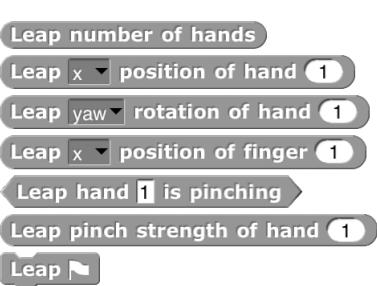
These blocks make it more convenient to think of a text string either as a *word* made of letters or as a *sentence* made of words, rather than of individual characters. Each word of a sentence is a string of characters not including a space. That is, the sentence-related blocks split the string up at space characters. Any number of consecutive spaces counts as a single separator. The names should be self-explanatory. There's no **first letter of** block because **letter 1 of** serves that need. **Join words** is like the primitive **join** except that it puts a space in the reported value between each of the inputs. The last four blocks convert back and forth between text strings (words or sentences) and lists; it's more convenient to treat a sentence as a list of words rather than a string of words because in the former case you can use the higher order functions **map**, **keep**, and **combine** on them.

The multiple-branch conditional library has these blocks:



The **catch** and **throw** blocks duplicate ones in the iteration library, and are included because they are used to implement the others. The **cases** block sets up a multi-branch conditional, similar to **cond** in Lisp or **switch** in C-family languages. The first branch is built into the **cases** block; it consists of a Boolean test in the first hexagonal slot and an action script, in the G-slot, to be run if the test reports **true**. The remaining branches go in the variadic hexagonal input at the end; each branch consists of an **else if** block, which includes the Boolean test and the corresponding action script, except possibly for the last branch, which can use the unconditional **else** block. As in other languages, once a branch succeeds, no other branches are tested.

The LEAP Motion library has these blocks:



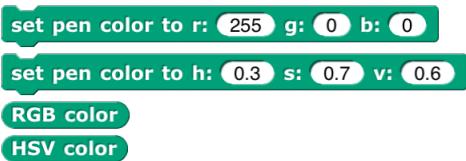
The LEAP Motion controller is a piece of hardware that senses the position of the user's hands, modeling the angles of each knuckle. It's one of many external devices that **Snap!** can support. Generally, to use a device with **Snap!**, you must download and install a separate controller program. That's true for the LEAP too, but that controller is installed as part of setting up the device, so all that's needed is the block library to go with it.

The text costume library has only one block:

costume from text [A size 72]

It reports a costume that can be used with the **switch to costume** block to make a button: **Snap!**

The pen color library has these blocks:



The first two are commands to set the pen color; the last two report the pen color. The two supported color space representations are RGB (red-green-blue), in which each color intensity is a number between 0 and 255, and HSV (hue-saturation-value), in which each dimension is a fraction between 0 and 1. The reporters report three-item lists.

The error catching library has these blocks:



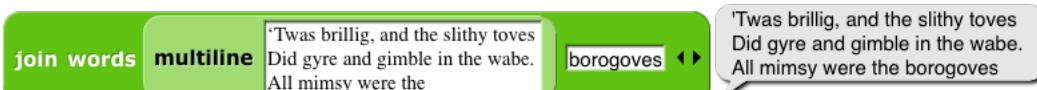
The **safely try** block allows you to handle errors that happen when your program is run within the program, instead of stopping the script with a red halo and an obscure error message. The block runs the script in its first C-slot. If it finishes without an error, nothing else happens. But if an error happens, the code in the second C-slot is run. While that second script is running, the variable **error** contains the text of the error message that would have been displayed if you weren't catching the error.

The **error** block is sort of the opposite: it lets your program *generate* an error message, which will be displayed with a red halo unless it is caught by **safely try**. The **let** block is used in implementing the library; it's like a **script variables** followed by a **set**.

The multi-line library has only one block:

multiline []

It's used to allow a text input to be more than one line long. With this library you can say



The multi-line library is semi-obsolete because you can now make multi-line input slots in your custom blocks; see page 43. But it's still useful when you need a multi-line input to a primitive block.

The system getter/setter library has these blocks:



The purpose of this library is to allow program access to the settings controlled by user interface elements, such as the settings menu. The **setting** block reports a setting; the **set flag** block sets yes-or-no options that have checkboxes in the user interface, while the **set value** block controls settings with numeric or text values, such as project name.

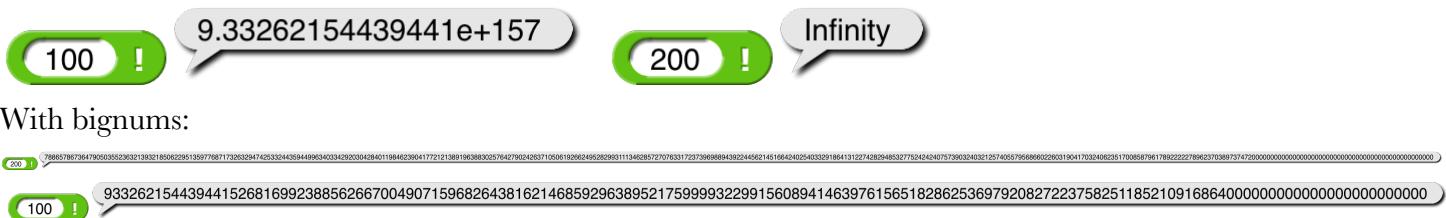
Certain settings are ordinarily remembered on a per-user basis, such as the “zoom blocks” value. But when these settings are changed by this library, the change is in effect only while the project using the library is loaded. No permanent changes are made.

The infinite precision integer library has these blocks:



The **USE BIGNUMS** block takes a Boolean input, to turn the infinite precision feature on or off. When on, all of the arithmetic operators are redefined to accept and report integers of any number of digits (limited only by the memory of your computer) and, in fact, the entire Scheme numeric tower, with exact rationals and with complex numbers. The **Scheme number** block has a list of functions applicable to Scheme numbers, including subtype predicates such as **rational?** and **infinite?**, and selectors such as **numerator** and **real-part**.

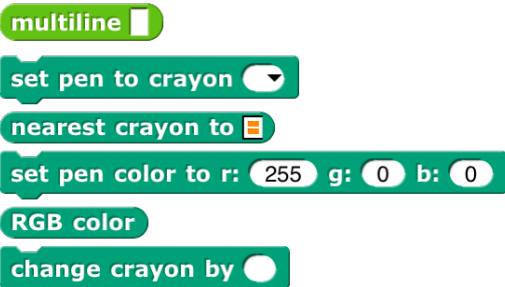
The `!bignum` block computes the factorial function, used to test whether the bignum package is turned on. Without bignums:



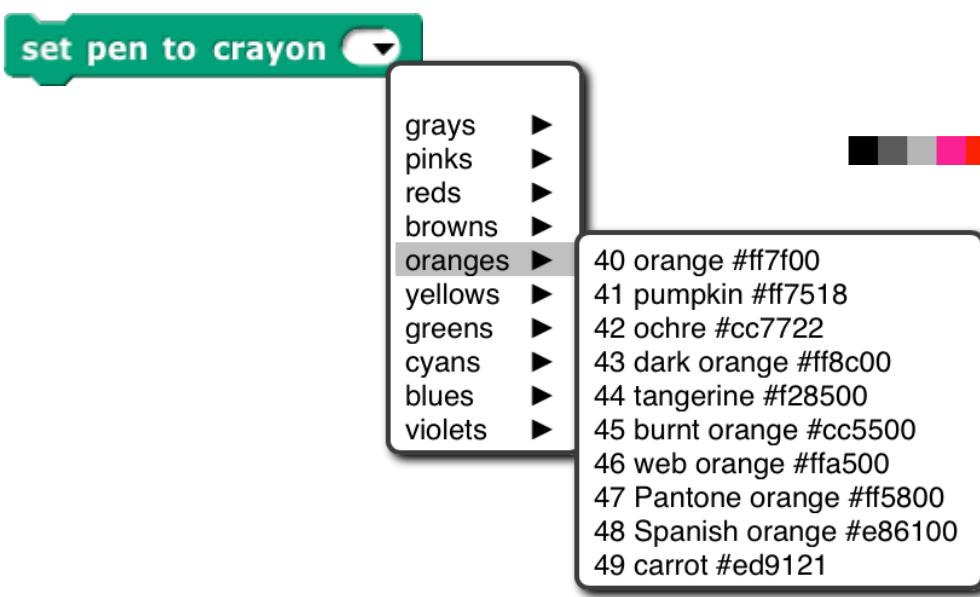
With bignums:

The 375-digit value of $200!$ isn't readable on this page, but if you shift-right-click on the block and choose "script pic with result," you can open the resulting picture in a browser window and scroll through it. (These values end with a bunch of zero digits. That's not roundoff error; the prime factors of $100!$ and $200!$ include many copies of 2 and 5.) The **if-then-else** reporter is used inside $!$, and the block with no name is a way to enter things like **$3/4$** and **$4+7i$** into numeric input slots by converting the slot to Any type.

The crayon library has these blocks:



This library is an alternative to the **set pen hue** block with what is hoped to be a more interesting set of colors. For one thing, the rainbow colors are not equally spaced in color space; half the hue spectrum is some flavor of green, while brown isn't a rainbow color at all, nor are the shades of gray. The crayon library provides the equivalent of a box of 100 crayons. They are divided into color groups, so the menu in the **set pen to crayon** input slot has submenus, as shown below. The colors are chosen so that starting from crayon 0, **change crayon by 10** rotates through an interesting, basic set of ten colors:



Using **change crayon by 5** instead gives ten more colors, for a total of 20:



The **set pen to crayon** and **change crayon by** blocks are the only ones meant for users of the library; the others are helper functions used within the library itself. Of those, **nearest crayon to** is the only crayon-specific one; it takes a three-item RGB coordinate list (each

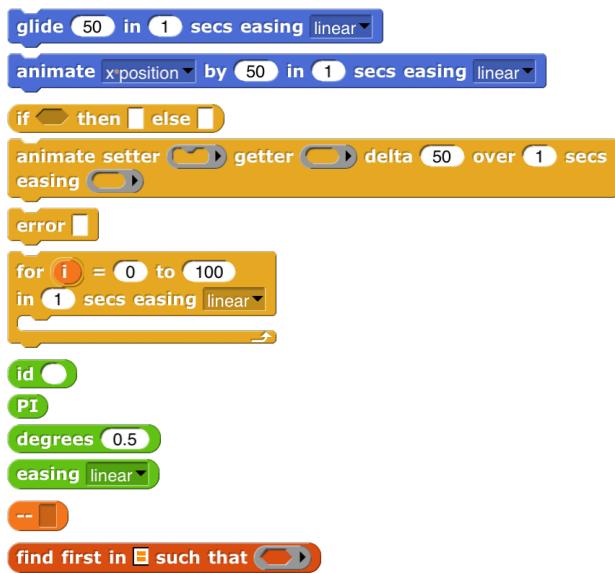
in the range [0-255]) and reports the crayon number with the color closest to the input color. It's needed so that you can use **change crayon by** even if the pen color wasn't set to a crayon previously. (Why didn't we use the colors of the 100-crayon Crayola™ box? A few reasons, one of which is that some Crayola colors aren't representable on RGB screens. Some year when you have nothing else to do, look up "color space" on Wikipedia. Also "crayon." Oh, it's deliberate that **change crayon by 5** doesn't include white, since that's the usual stage background color. White is crayon 14.)

The speech synthesis library has these blocks:



This library interfaces with a capability in up-to-date browsers, so it might not work for you. It works best if the accent matches the text!

The animation library has these blocks:



Despite the name, this isn't only about graphics; you can animate the values of a variable, or anything else that's expressed numerically.

The central idea of this library is an *easing function*, a reporter whose domain and range are real numbers between 0 and 1 inclusive. The function represents what fraction of the "distance" (in quotes because it might be any numeric value, such as temperature in a simulation of weather) from here to there should be covered in what fraction of the time. A linear easing function means steady progression. A quadratic easing function means starting slowly and accelerating. (Note that, since it's a requirement that $f(0)=0$ and $f(1)=1$, there is only one linear easing function, $f(x)=x$, and similarly for other categories.) The block `easing linear` reports some of the common easing functions.

The two Motion blocks in this library animate a sprite. **Glide** always animates the sprite's motion. **Animate**'s first pulldown menu input allows you to animate horizontal or vertical motion, but will also animate the sprite's direction or size. The **animate** block in Control lets you animate any numeric quantity with any easing function. The getter and setter inputs are best explained by example:

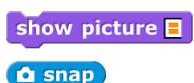


is equivalent to



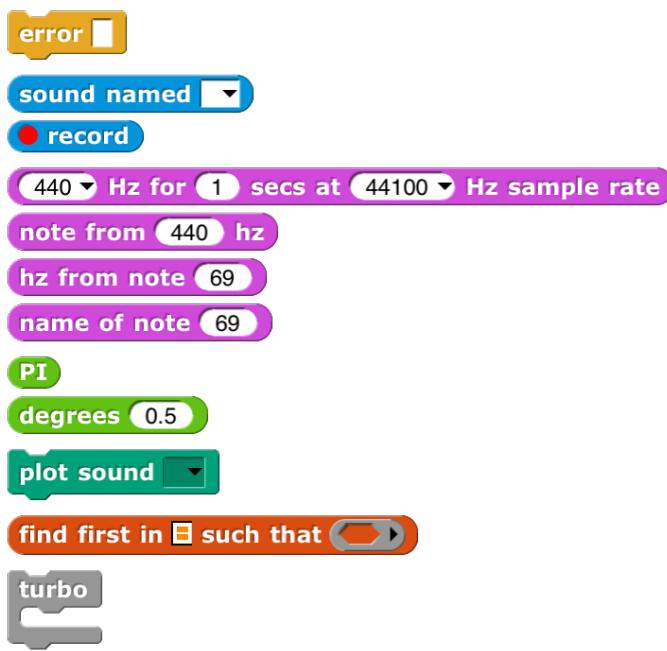
The other eight blocks in the library are helpers for these four.

The pixels library has these blocks:



Costumes are first class data in **Snap!**. Most of the processing of costume data is done by primitive blocks in the Looks category. (See page 60.) This library provides **show picture**, which abbreviates a **switch to costume** and a **stretch** as explained in the section mentioned above, and **snap**, which takes a picture using your computer's camera and reports it as a costume.

The sound manipulation library includes these blocks:



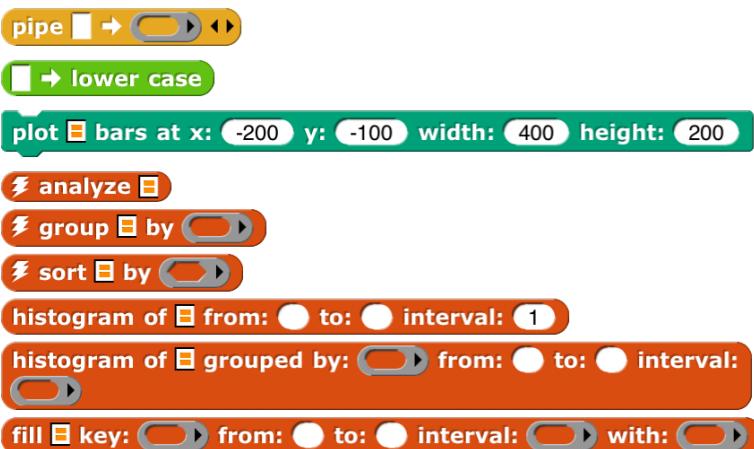
This library takes a sound, one that you record or one from our collection of sounds, and manipulates it by systematically changing the intensity of the samples in the sound and by changing the sampling rate at which the sound is reproduced. Many of the blocks are helpers for the **plot sound** block, used to plot the waveform of a sound. The **play** block plays a sound. **__ Hz for** reports a sine wave as a list of samples.

The bigger data library includes these *experimental* blocks:



They are the same as the corresponding blocks in the list utilities library, but using a partial compilation technique to allow some use cases to run very fast, even on extremely large lists. Only primitive functions can be used in the function input slots. Save your project before experimenting with these. Blocks starting with the lightning bolt symbol in other libraries are also partially compiled for speed.

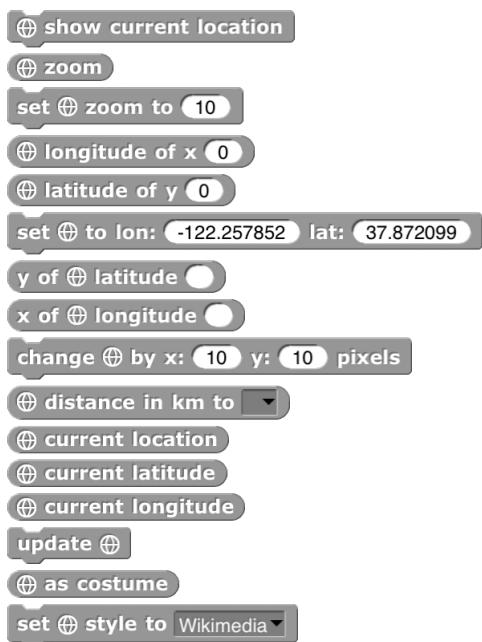
The frequency distribution analysis library has these blocks:



This is a collection of tools for analyzing large data sets and plotting histograms of how often some value is found in some column of the table holding the data.

For more information go here:
<http://tinyurl.com/jens-data>

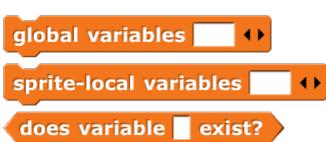
The world map library has these blocks:



Using any of the command blocks puts a map on the screen, in a layer in front of the stage's background but behind the pen trails layer (which is in turn behind all the sprites). The first block asks your browser for your current physical location, for which you may be asked to give permission. The next two blocks get and set the map's zoom amount; the default zoom of 10 fits from San Francisco not quite down to Palo Alto on the screen. A zoom of 1 fits almost the entire world. A zoom of 3 fits the United States; a zoom of 5 fits Germany. The zoom can be changed in half steps, i.e., 5.5 is different from 5, but 5.25 isn't.

The next five blocks concern the conversion between stage coordinates (pixels) and Earth coordinates (latitude and longitude). The **change by x: y:** block shifts the map relative to the stage. The **distance to** block measures the map distance (in meters) between two sprites. The three reporters with **current** in their names find *your* actual location, again supposing that geolocation is enabled on your device. **Update** redraws the map; **as costume** reports the visible section of the map as a costume. **Set style** allows things like satellite pictures.

The variables library has these blocks:



These blocks allow a program to perform the same operation as the **Make a variable** button, making either global or “for this sprite only” variables.

The JSON library includes these blocks:



Listify takes some text in JSON format (see page 39) and converts it to a structured list. **Value at key** looks up a key-value pair in a (listified) JSON dictionary.

The parallelization library contains these blocks:



The two **do in parallel** blocks take any number of scripts as inputs. Those scripts will be run in parallel, like ordinary independent scripts in the scripting area. The **and wait** version waits until all of those scripts have finished before continuing the script below the block.

The string processing library provides these blocks:



All of these could be written in **Snap!** itself, but these are implemented using the corresponding JavaScript library functions directly, so they run fast. They can be used, for example, in scraping data from a web site.

Back to the file menu: The **Costumes...** option opens a browser into the costume library:



You can import a single costume by clicking it and then clicking the Import button. Alternatively, you can import more than one costume by double-clicking each one, and then clicking Cancel when done. Notice that some costumes are tagged with “svg” in this picture; those are vector-format costumes that are not (yet, coming soon) editable within **Snap!**.

If you have the stage selected in the sprite corral, rather than a sprite, the **Costumes...** option changes to a **Backgrounds...** option, with different choices in the browser:



The costume and background libraries include both bitmap (go jagged if enlarged) and vector (enlarge smoothly) images. Thanks to Scratch 2.0/3.0 for most of these images! Some older browsers refuse to import a vector image, but instead convert it to bitmap.

The **Sounds...** option opens the third kind of media browser:



The Play buttons can be used to preview the sounds.

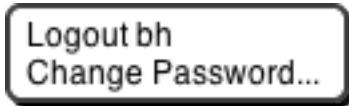
The Cloud Menu

The cloud icon shows a menu of options relating to your **Snap!** cloud account. If you are not logged in, you see this menu:



Choose **Login...** if you have a **Snap!** account and remember your password. Choose **Signup...** if you don't have an account. Choose **Reset Password...** if you've forgotten your password or just want to change it. You will then get an email, at the address you gave when you created your account, with a new temporary password. Use that password to log in, then you can choose your own password, as shown below.

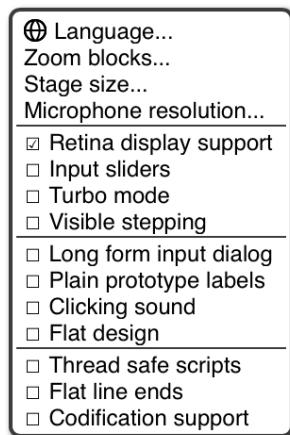
If you are already logged in, you'll see this menu:



Logout is obvious, but has the additional benefit of showing you who's logged in. **Change password...** will ask for your old password (the temporary one if you're resetting your password) and the new password you want, entered twice because it doesn't echo.

The Settings Menu

The settings icon shows a menu of **Snap!** options, either for the current project or for you permanently, depending on the option:



The **Language...** option lets you see the **Snap!** user interface (blocks and messages) in a language other than English. (Note: Translations have been provided by **Snap!** users. If your native language is missing, send us an email!)

The **Zoom blocks...** option lets you change the size of blocks, both in the palettes and in scripts. The standard size is 1.0 units. The main purpose of this option is to let you take very high-resolution pictures of scripts for use on posters. It can also be used to improve readability when projecting onto a screen while lecturing, but bear in mind that it doesn't make the palette or script areas any wider, so your computer's command-option-+ feature may be more practical. Note that a zoom of 2 is gigantic! Don't even try 10.

The **Stage size...** option lets you set the size of the *full-size* stage in pixels. If the stage is in half-size or double-size (presentation mode), the stage size values don't change; they always reflect the full-size stage.

The **Microphone resolution...** option sets the buffer size used by the **microphone** block in Settings. "Resolution" is an accurate name if you are getting frequency domain samples; the more samples, the narrower the range of frequencies in each sample. In the time domain, the buffer size determines the length of time over which samples are collected.

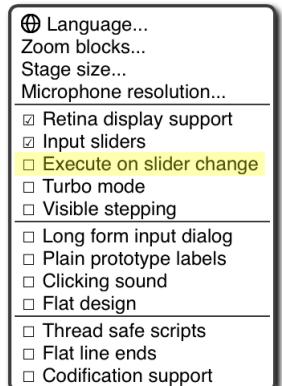
The remaining options let you turn various features on and off. There are three groups of checkboxes. The first is for temporary settings not saved in your project nor in your user preferences. **Retina display support** is

included only if your computer is attached to a retina (or “4x HD”) display, in which case it is on by default. Retina support makes text and blocks much sharper than if you have a retina display without retina support enabled, but every picture has four times as many pixels, which noticeably slows down displaying. (Also, if you take script or stage pictures, they will be twice as tall and twice as wide as you’re expecting, when viewed on non-retina displays.)

Input sliders provides an alternate way to put values in numeric input slots; if you click in such a slot, a slider appears that you can control with the mouse:



The range of the slider will be from 25 less than the input’s current value to 25 more than the current value. If you want to make a bigger change than that, you can slide the slider all the way to either end, then click on the input slot again, getting a new slider with a new center point. But you won’t want to use this technique to change the input value from 10 to 1000, and it doesn’t work at all for non-integer input ranges. This feature was implemented because software keyboard input on phones and tablets didn’t work at all in the beginning, and still doesn’t on Android devices, so sliders provide a workaround. It has since found another use in providing “lively” response to input changes; if **Input sliders** is checked, reopening the settings menu will show an additional option called **Execute on slider change**. If this option is also checked, then changing a slider in the scripting area automatically runs the script in which that input appears. The project **live-tree** in the **Examples** collection shows how this can be used; it features a fractal tree custom block with several inputs, and you can see how each input affects the picture by moving a slider.



Turbo mode makes many projects run much faster, at the cost of not keeping the stage display up to date. (**Snap!** ordinarily spends most of its time drawing sprites and updating variable watchers, rather than actually carrying out the instructions in your scripts.) So turbo mode isn’t a good idea for a project with **glide** blocks or one in which the user interacts with animated characters, but it’s great for drawing a complicated fractal, or computing the first million digits of π , so that you don’t need to see anything until the final result. While in turbo mode, the button that normally shows a green flag instead shows a green lightning bolt. (But **when green flag clicked** hat blocks still activate when the button is clicked.)

Visible stepping enables the slowed-down script evaluation described in Chapter I. Checking this option is equivalent to clicking the footprint button above the scripting area. You don’t want this on except when you’re actively debugging, because even the fastest setting of the slider is still slowed a lot.

The next group of four are user preference options, preserved when you load a new project. **Long form input dialog**, if checked, means that whenever a custom block input name is created or edited, you immediately see the version of the input name dialog that includes the type options, default value setting, etc., instead of the short form with just the name and the choice between input name and title text. The default (unchecked) setting is definitely best for beginners, but more experienced **Snap!** programmers may find it more convenient always to see the long form.

Plain prototype labels eliminates the plus signs between words in the Block Editor prototype block. This makes it harder to add an input to a custom block; you have to hover the mouse where the plus sign would have

been, until a single plus sign appears temporarily for you to click on. It's intended for people making pictures of scripts in the block editor for use in documentation, such as this manual. You probably won't need it otherwise.

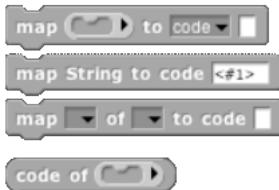
Clicking sound causes a really annoying sound effect whenever one block snaps next to another in a script. Certain very young children, and our colleague Dan Garcia, like this, but if you are such a child you should bear in mind that driving your parents or teachers crazy will result in you not being allowed to use **Snap!**. It might be useful for visually impaired users.

Flat design changes the “skin” of the **Snap!** window to a really hideous design with white and pale-grey background, rectangular rather than rounded buttons, and monochrome blocks (rather than the shaded, somewhat 3D-looking normal blocks). The monochrome blocks are the reason for the “flat” in the name of this option. The only thing to be said for this option is that, because of the white background, it may blend in better with the rest of a web page when a **Snap!** project is run in a frame in a larger page.

The final group of settings change the way **Snap!** interprets your program; they are saved with the project, so anyone who runs your project will experience the same behavior. **Thread safe scripts** changes the way **Snap!** responds when an event (clicking the green flag, say) starts a script, and then, while the script is still running, the same event happens again. Ordinarily, the running process stops where it is, ignoring the remaining commands in the script, and the entire script starts again from the top. This behavior is inherited from Scratch, and some converted Scratch projects depend on it; that's why it's the default. It's also sometimes the right thing, especially in projects that play music in response to mouse clicks or keystrokes. If a note is still playing but you ask for another one, you want the new one to start right then, not later after the old process finishes. But if your script makes several changes to a database and is interrupted in the middle, the result may be that the database is inconsistent. When you select **Thread safe scripts**, the same event happening again in the middle of running a script is simply ignored. (This is arguably still not the right thing; the event should be remembered and the script run again as soon as it finishes. We'll probably get around to adding that choice eventually.) Keyboard events (when key pressed) are always thread-safe.

Flat line ends affects the drawing of thick lines (large pen width). Usually the ends are rounded, which looks best when turning corners. With this option selected, the ends are flat. It's useful for drawing a brick wall or filling a rectangle.

Codification support enables an experimental feature that can translate a **Snap!** project to a text-based (rather than block-based) programming language. The feature doesn't know about any particular other language; instead, you can provide a translation for each primitive block using these special blocks:



Using these primitive blocks, you can build a block library to translate into any programming language. Watch for such libraries to be added to our library collection (or contribute one). To see some examples, open the project “**Codification**” in the **Examples** project list. Edit the blocks **map to Smalltalk**, **map to JavaScript**, etc., to see examples of how to provide translations for blocks.

Visible Stepping Controls

After the menu buttons you'll see the project name. After that comes the footprint  button used to turn on visible stepping and, when it's on, the slider to control the speed of stepping.

Stage Resizing Buttons

Still in the tool bar, but above the left edge of the stage, are two buttons that change the size of the stage. The first is the **shrink/grow** button. Normally it looks like this:  Clicking the button displays the stage at half-normal size horizontally and vertically (so it takes up $\frac{1}{4}$ of its usual area). When the stage is half size the button looks like this:  and clicking it returns the stage to normal size. The main reason you'd want a half size stage is during the development process, when you're assembling scripts with wide input expressions and the normal scripting area isn't wide enough to show the complete script. You'd typically then switch back to normal size to try out the project. The next **presentation mode** button normally looks like this:  Clicking the button makes the stage double size in both dimensions and eliminates most of the other user interface elements (the palette, the scripting area, the sprite corral, and most of the tool bar). When you open a shared project using a link someone has sent you, the project starts in presentation mode. While in presentation mode, the button looks like this:  Clicking it returns to normal (project development) mode.

Project Control Buttons

Above the right edge of the stage are three buttons that control the running of the project.

Technically, the **green flag**  is no more a project control than anything else that can trigger a hat block: typing on the keyboard or clicking on a sprite. But it's a convention that clicking the flag should start the action of the project from the beginning. It's only a convention; some projects have no flag-controlled scripts at all, but respond to keyboard controls instead. Clicking the green flag also deletes temporary clones. Shift-clicking the button enters Turbo mode, and the button then looks like a lightning bolt:  Shift-clicking again turns Turbo mode off.

Scripts can simulate clicking the green flag by **broadcasting** the special message **__shout__go__** (two underscores in each of the three positions shown).

The **pause** button  suspends running all scripts. If clicked while scripts are running, the button changes shape to become a **play** button:  Clicking it while in this form resumes the suspended scripts. There is also a **pause all** block in the Control palette that can be inserted in a script to suspend all scripts; this provides the essence of a breakpoint debugging capability. The use of the pause button is slightly different in visible stepping mode, described in Chapter I.

The **stop** button  stops all scripts, like the **stop all** block. It does *not* prevent a script from starting again in response to a click or keystroke; the user interface is always active. There is one exception: generic **when** blocks  will not fire until some non-generic event starts a script. The **stop** button also deletes all temporary clones.

B. The Palette Area

At the top of the palette area are the eight buttons that select which palette (which block category) is shown: Motion, Looks, Sound, Pen, Control, Sensing, Operators, and Variables (which also includes the List and Other blocks). There are no menus behind these buttons.

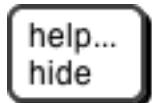
Buttons in the Palette

Under the eight palette selector buttons, at the top of the actual palette, are two semi-transparent buttons. The first is the *search*  button, which is equivalent to typing control-F: It replaces the palette with a search bar into which you can type part of the title text of the block you're trying to find. To leave this search mode, click one of the eight palette selectors, or type the Escape key.

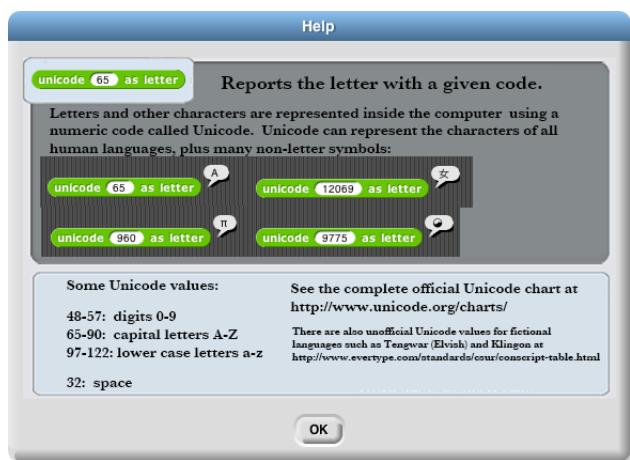
The other button  is equivalent to the “Make a block” button, except that the dialog window that it opens has the current palette (color) preselected.

Context Menus for Palette Blocks

Most elements of the **Snap!** display can be control-clicked/right-clicked to show a *context menu*, with items relevant to that element. If you control-click/right-click a *primitive* block in the palette, you see this menu:

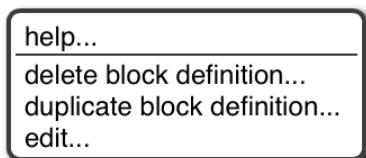


The **help...** option displays a box with documentation about the block. Here's an example:

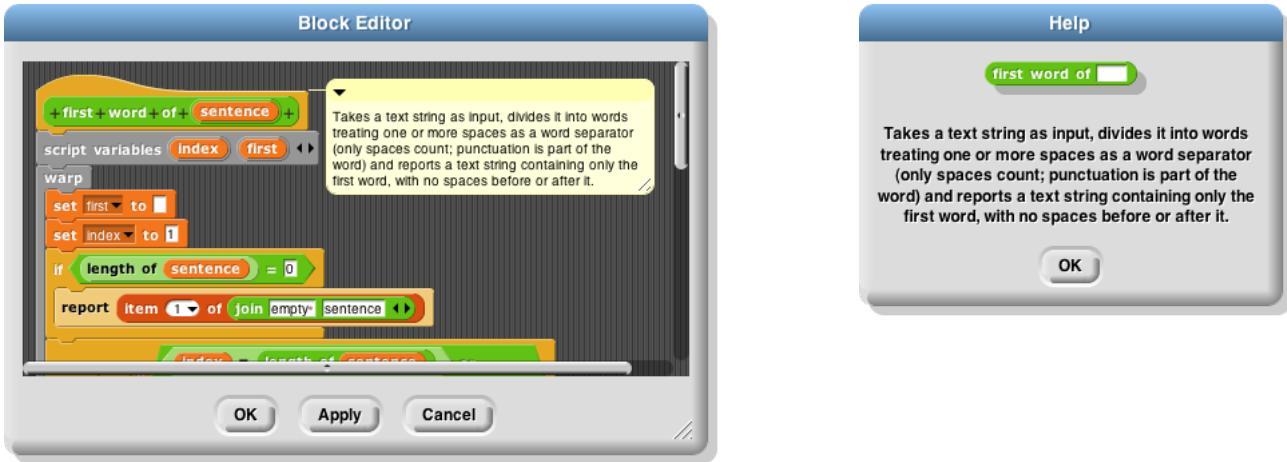


The **hide** option removes that block from the palette. (This option is available only when clicking the block in the palette itself, not in a script.) The purpose of the option is to allow teachers to present students with a simplified **Snap!** with some features effectively removed. The hiddenness of primitives is saved with each project, so students can load a shared project and see just the desired blocks.

If you control-click/right-click a *custom* (user-defined) block in the palette, you see this menu:



The **help...** option for a custom block displays the comment, if any, attached to the custom block's hat block in the Block Editor. Here is an example of a block with a comment and its help display:



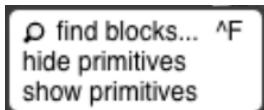
The **delete block definition...** option asks for confirmation, then deletes the custom block and removes it from any scripts in which it appears. (The result of this removal may not leave a sensible script; it's best to find and correct such scripts *before* deleting a block.) Note that there is no option to *hide* a custom block.

The **duplicate block definition...** option makes a *copy* of the block and opens that copy in the Block Editor. You must change the title text or input types of the new block in order to be able to exit from the Block Editor; you can't have two identical-looking blocks.

The **edit...** option opens a Block Editor with the definition of the custom block.

Context Menu for the Palette Background

Right-click/control-click on the grey *background* of the palette area shows this menu:



The **find blocks...** option does the same thing as the magnifying-glass button. The **hide primitives** option hides *all* of the primitives in that palette. The **show primitives** option, which is in the menu only if some primitives of this palette are hidden, unhides all of them.

Palette Resizing



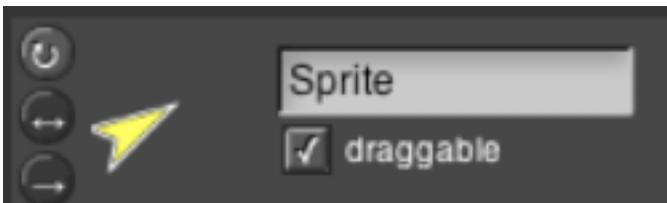
At the right end of the palette area, just to the left of the scripting area, is a resizing handle that can be dragged rightward to increase the width of the palette area. This is useful if you write custom blocks with very long names. You can't reduce the width of the palette below its standard value.

C. The Scripting Area

The scripting area is the middle vertical region of the **Snap!** window, containing scripts and also some controls for the appearance and behavior of a sprite. There is always a *current sprite*, whose scripts are shown in the scripting area. A dark grey rounded rectangle in the sprite corral shows which sprite (or the stage) is current. Note that it's only the visible *display* of the scripting area that is "current" for a sprite; all scripts of all sprites may be running at the same time. Clicking on a sprite thumbnail in the sprite corral makes it current. The stage itself can be selected as current, in which case the appearance is slightly different.

Sprite Appearance and Behavior Controls

At the top of the scripting area are a picture of the sprite and some controls for it:

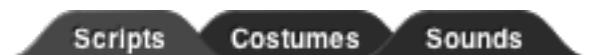


Note that the sprite picture reflects its rotation, if any. There are three things that can be controlled here:

1. The three circular buttons in a column at the left control the sprite's *rotation* behavior. Sprite costumes are designed to be right-side-up when the sprite is facing toward the right (direction = 90). If the topmost button is lit, the default as shown in the picture above, then the sprite's costume rotates as the sprite changes direction. If the middle button is selected, then the costume is reversed left-right when the sprite's direction is roughly leftward (direction between 180 and 359, or equivalently, between -180 and -1). If the bottom button is selected, the costume's orientation does not change regardless of the sprite's direction.
2. The sprite's *name* can be changed in the text box that, in this picture, says "Sprite."
3. Finally, if the **draggable** checkbox is checked, then the user can move the sprite on the stage by clicking and dragging it. The common use of this feature is in game projects, in which some sprites are meant to be under the player's control but others are not.

Scripting Area Tabs

Just below the sprite controls are three *tabs* that determine what is shown in the scripting area:



Scripts and Blocks Within Scripts

Most of what's described in this section also applies to blocks and scripts in a Block Editor.

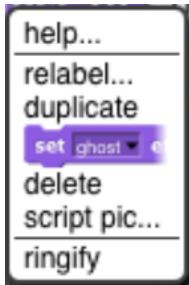
Clicking on a script (which includes a single unattached block) runs it. If the script starts with a hat block, clicking on the script runs it even if the event in the hat block doesn't happen. (This is a useful debugging technique when you have a dozen sprites and they each have five scripts with green-flag hat blocks, and you want to know what a single one of those scripts does.) The script will have a green "halo" around it while it's running. If the script is shared with clones, then while it has the green halo it will also have a count of how many instances of the script are running. Clicking a script with such a halo *stops* the script. (If the script includes a **warp** block, which might be inside a custom block used in the script, then **Snap!** may not respond immediately to clicks.)

If a script is shown with a *red* halo, that means that an error was caught in that script, such as using a list where a number was needed, or vice versa. Clicking the script will turn off the halo.

If any blocks have been dragged into the scripting area, then in its top right corner you'll see an *undo*  and/or *redo*  button that can be used to undo or redo block and script drops. When you undo a drop into an input slot, whatever used to be in the slot is restored. The redo button appears once you've used undo.

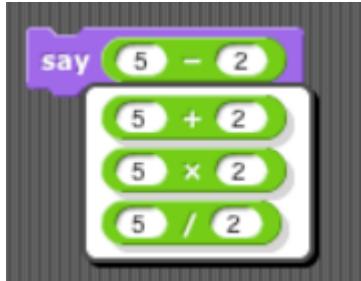
The third button  starts keyboard editing mode (Section D. , page 110).

Control-click/right-clicking a primitive block within a script shows a menu like this one:



The **help...** option shows the help screen for the block, just as in the palette. The other options appear only when a block is right-clicked/control-clicked in the scripting area.

Not every primitive block has a **relabel...** option. When present, it allows the block to be replaced by another, similar block, keeping the input expressions in place. For example, here's what happens when you choose **relabel...** for an arithmetic operator:



Note that the inputs to the existing – block are displayed in the menu of alternatives also. Click a block in the menu to choose it, or click outside the menu to keep the original block.

The **duplicate** option makes a copy of the *entire script* starting from the selected block. The copy is attached to the mouse, and you can drag it to another script (or even to another Block Editor window), even though you are no longer holding down the mouse button. Click the mouse to drop the script copy.

The block picture underneath the word **duplicate** is another duplication option, but it duplicates only the selected block, not everything under it in the script. Note that if the selected block is a C-shaped control block, the script inside its C-shaped slot is included. If the block is at the end of its script, this option does not appear.

The **delete** option deletes the selected block from the script.

The **script pic...** option opens a new browser tab containing a picture of the entire script, not just from the selected block to the end, or, in some browsers, saves the picture directly. In the first case, you can use the browser's Save feature to put the picture in a file. This is a super useful feature if you happen to be writing a *Snap!* manual! (If you have a Retina display, consider turning off Retina support before making script pictures; if not, they end up huge.)

If the script does *not* start with a hat block, or you clicked on a reporter, then there's one more option: **ringify** (and, if there is already a grey ring around the block or script, **unringify**). **Ringify** surrounds the block (reporter) or the entire script (command) with a grey ring, meaning that the block(s) inside the ring are themselves data, as an input to a higher order procedure, rather than something to be evaluated within the script. See Section VI, Procedures as Data.

Clicking a *custom* block in a script gives a similar but different menu:



The **relabel...** option for custom blocks shows a menu of other same-shape custom blocks with the same inputs. At present you can't relabel a custom block to a primitive block or vice versa. The two options at the bottom, for custom blocks only, are the same as in the palette.

Scripting Area Background Context Menu

Control-click/shift-click on the grey striped background of the scripting area gives this menu:



The **undrop** option is a sort of “undo” feature for the common case of dropping a block somewhere other than where you meant it to go. It remembers all the dragging and dropping you’ve done in this sprite’s scripting area (that is, other sprites have their own separate drop memory), and undoes the most recent, returning the block to its former position, and restoring the previous value in the relevant input slot, if any. Once you’ve undropped something, the **redrop** option appears, and allows you to repeat the operation you just undid. These menu options are equivalent to the and buttons described earlier.

The **clean up** option rearranges the position of scripts so that they are in a single column, with the same left margin, and with uniform spacing between scripts. This is a good idea if you can’t read your own project!

The **add comment** option puts a comment box, like the picture to the right, in the scripting area. It’s attached to the mouse, as with duplicating scripts, so you position the mouse where you want the comment and click to release it. You can then edit the text in the comment as desired.

You can drag the bottom right corner of the comment box to resize it. Clicking the arrowhead at the top left changes the box to a single-line compact form, as below, so that you can have a number of collapsed comments in the scripting area and just expand one of them when you want to read it in full.



If you drag a comment over a block in a script, the comment will be attached to the block with a yellow line:



Comments have their own context menu, with obvious meanings:



Back to the options in the menu for the background of the scripting area:



The **scripts pic...** option saves, or opens a new browser tab with, a picture of *all* scripts in the scripting area, just as they appear, but without the grey striped background. Note that “all scripts in the scripting area” means just the top-level scripts of the current sprite, not other sprites’ scripts or custom block definitions.

Finally, the **make a block...** option does the same thing as the “Make a block” button in the palettes. It’s a shortcut so that you don’t have to keep scrolling down the palette if you make a lot of blocks.

Controls in the Costumes Tab

If you click on the word “Costumes” under the sprite controls, you’ll see something like this:



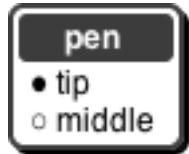
The **Turtle** costume is always present in every sprite; it is costume number 0. Other costumes can be painted within **Snap!** or imported from files or other browser tabs if your browser supports that. Clicking on a costume selects it; that is, the sprite will look like the selected costume. Clicking on the paint brush icon  opens the *Paint Editor*, in which you can create a new costume. Clicking on the camera icon  opens a window in

which you see what your computer's camera is seeing, and you can take a picture (which will be the full size of the stage unless you shrink it in the Paint Editor). This works only if you give **Snap!** permission to use the camera, and maybe only if you opened **Snap!** in secure (HTTPS) mode, and then only if your browser loves you.



Brian's bedroom when he's staying at Paul's house.

Control-clicking/right-clicking on the turtle picture gives this menu:



In this menu, you choose the turtle's *rotation point*, which is also the point from which the turtle draws lines. The two pictures below show what the stage looks like after drawing a square in each mode; **tip** (otherwise known as "Jens mode") is on the left in the pictures below, **middle** ("Brian mode") on the right:



As you see, "tip" means the front tip of the arrowhead; "middle" is not the middle of the shaded region, but actually the middle of the four vertices, the concave one. (If the shape were a simple isosceles triangle instead of a fancier arrowhead, it would mean the midpoint of the back edge.) The advantage of **tip** mode is that the sprite is less likely to obscure the drawing. The advantage of **middle** mode is that the rotation point of a sprite is rarely at a tip, and students are perhaps less likely to be confused about just what will happen if you ask the turtle to turn 90 degrees from the position shown. (It's also the traditional rotation point of the Logo turtle, which originated this style of drawing.)

Costumes other than the turtle have a different context menu:

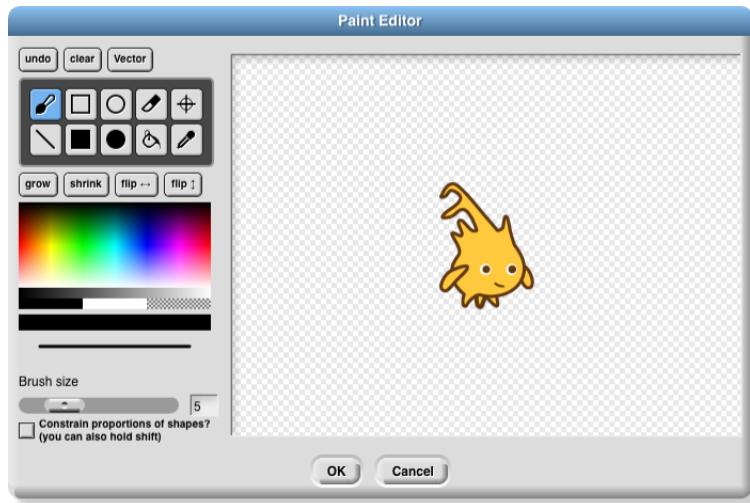


The **edit** option opens the Paint Editor on this costume. The **rename** option opens a dialog box in which you can rename the costume. (A costume's initial name comes from the file from which it was imported, if any, or is something like **costume5**.) **Duplicate** makes a copy of the costume, in the same sprite. (Presumably you'd do that because you intend to edit one of the copies.) **Delete** is obvious. The **export** option opens a new browser tab with a picture of the costume. You can then save it to a file, or select a different sprite in the **Snap!** tab, return to the picture tab, and drag the costume onto the **Snap!** tab to copy the costume to another sprite.

You can drag costumes up and down in the Costumes tab in order to renumber them, so that **next costume** will behave as you prefer.

The Paint Editor

Here is a picture of a Paint Editor window:



If you've used any painting program, most of this will be familiar to you. Currently, costumes you import can be edited only if they are in a bitmap format (png, jpeg, gif, etc.). There is a vector editor, but it works only for creating a costume, not editing an imported vector (svg) picture. Unlike the case of the Block Editor, only one Paint Editor window can be open at a time.

The ten square buttons in two rows of five near the top left of the window are the *tools*. The top row, from left to right, are the paintbrush tool, the outlined rectangle tool, the outlined ellipse tool, the eraser tool, and the rotation point tool. The bottom row tools are the line drawing tool, the solid rectangle tool, the solid ellipse tool, the floodfill tool, and the eyedropper tool. Below the tools is a row of four buttons that immediately change the picture. The first two change its overall size; the next two flip the picture around horizontally or vertically. Below these are a color palette, a greyscale tape, and larger buttons for black, white, and transparent paint. Below these is a solid bar displaying the currently selected color. Below that is a picture of a line showing the brush width for painting and drawing, and below that, you can set the width either with a slider or by typing a number (in pixels) into the text box. Finally, the checkbox constrains the line tool to draw horizontally or vertically, the rectangle tools to draw squares, and the ellipse tools to draw circles. You can get the same effect temporarily by holding down the shift key, which makes a check appear in the box as long as you hold it down. (But the Caps Lock key doesn't affect it.)

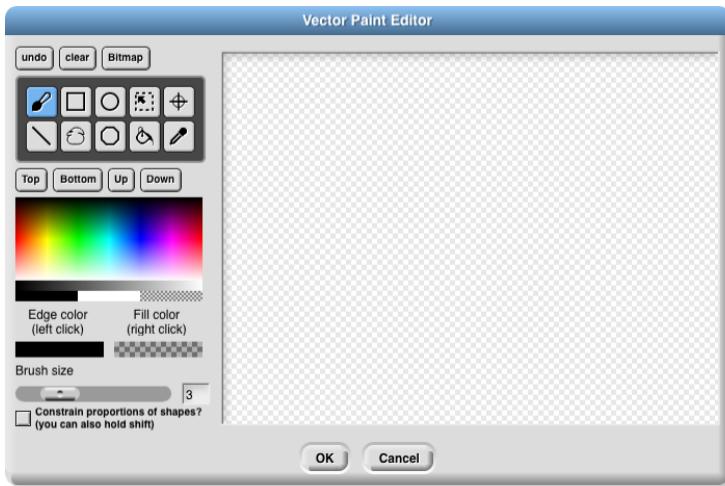
You can correct errors with the **undo** button, which removes the last thing you drew, or the **clear** button, which erases the entire picture. (Note, it does *not* revert to what the costume looked like before you started

editing it! If that's what you want, click the **Cancel** button at the bottom of the editor.) When you're finished editing, to keep your changes, click **OK**.

Note that the ellipse tools work more intuitively than ones in other software you may have used. Instead of dragging between opposite corners of the rectangle circumscribing the ellipse you want, so that the endpoints of your dragging have no obvious connection to the actual shape, in **Snap!** you start at the center of the ellipse you want and drag out to the edge. When you let go of the button, the mouse cursor will be on the curve. If you drag out from the center at 45 degrees to the axes, the resulting curve will be a circle; if you drag more horizontally or vertically, the ellipse will be more eccentric. (Of course if you want an exact circle you can hold down the shift key or check the checkbox.) The rectangle tools, though, work the way you expect: You start at one corner of the desired rectangle and drag to the opposite corner.

Using the eyedropper tool, you can click anywhere in the **Snap!** window, even outside the Paint Editor, and the tool will select the color at the mouse cursor for use in the Paint Editor. You can only do this once, because the Paint Editor automatically selects the paintbrush when you choose a color. (Of course you can click on the eyedropper tool button again.)

The only other non-obvious tool is the rotation point tool. It shows in the Paint Editor where the sprite's current rotation center is (the point around which it turns when you use a **turn** block); if you click or drag in the picture, the rotation point will move where you click. (You'd want to do this, for example, if you want a character to be able to wave its arm, so you use two sprites connected together. You want the rotation point of the arm sprite to be at the end where it joins the body, so it remains attached to the shoulder while waving.)



The vector editor's controls are much like those in the bitmap editor. One point of difference is that the bitmap editor has two buttons for solid and outline rectangles, and similarly for ellipses, but in the vector editor there is always an edge color and a fill color, even if the latter is “transparent paint,” and so only one button per shape is needed. Since each shape that you draw is a separate layer (like sprites on the stage), there are controls to move the selected shape up (forward) or down (backward) relative to other shapes. There is a selection tool to drag out a rectangular area and select all the shapes within that area.

Controls in the Sounds Tab

There is no Sound Editor in **Snap!**, and also no current sound the way there's a current costume for each sprite. (The sprite always has an appearance unless hidden, but it doesn't sing unless explicitly asked.) So the context menu for sounds has only **rename**, **delete**, and **export** options, and it has a clickable button labeled **Play** or **Stop** as appropriate. There is a sound *recorder*, which appears if you click the red record button ():



The first, round button starts recording. The second, square button stops recording. The third, triangular button plays back a recorded sound. If you don't like the result, click the round button again to re-record. When you're satisfied, push the **Save** button. If you need a sound editor, consider the free (both senses) <http://audacity.sourceforge.net>.

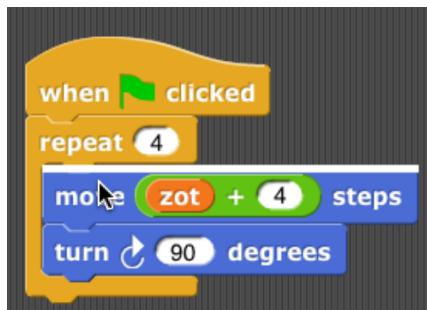
D. Keyboard Editing

An ongoing area of research is how to make visual programming languages usable by people with visual or motoric disabilities. As a first step in this direction, we provide a keyboard editor, so that you can create and edit scripts without tracking the mouse. So far, not every user interface element is controllable by keyboard, and we haven't even begun providing *output* support, such as interfacing with a speech synthesizer. This is an area in which we know we have a long way to go! But it's a start. The keyboard editor may also be useful to anyone who can type faster than they can drag blocks.

Starting and stopping the keyboard editor

There are three ways to start the keyboard editor. **Shift-clicking** anywhere in the scripting area will start the editor at that point: either editing an existing script or, if you shift-click on the background of the scripting area, editing a new script at the mouse position. Alternatively, typing **shift-enter** will start the editor on an existing script, and you can use the tab key to switch to another script. Or you can click the keyboard button at the top of the scripting area.

When the script editor is running, its position is represented by a blinking white bar:

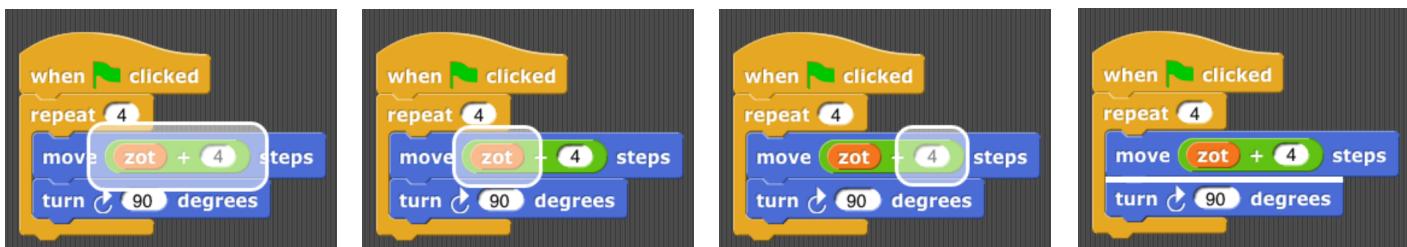


To leave the keyboard editor, type the **escape** key, or just click on the background of the scripting area.

Navigating in the keyboard editor

To move to a different script, type the **tab** key. **Shift-tab** to move through the scripts in reverse order.

A script is a vertical stack of command blocks. A command block may have input slots, and each input slot may have a reporter block in it; the reporter may itself have input slots that may have other reporters. You can navigate through a script quickly by using the **up arrow** and **down arrow** keys to move between command blocks. Once you find the command block that you want to edit, the **left and right arrow** keys move between editable items within that command. (Left and right arrow when there are no more editable items within the current command block will move up or down to another command block, respectively.) Here is a sequence of pictures showing the results of repeated right arrow keys starting from the position shown above:



You can rearrange scripts within the scripting area from the keyboard. Typing **shift-arrow** keys (left, right, up, or down) will move the current script. If you move it onto another script, the two won't snap together; the one you're moving will overlap the one already there. This means that you can move across another script to get to a free space.

Editing a script

Note that the keyboard editor *focus*, the point shown as a white bar or halo, is either *between* two command blocks or *on* an input slot. The editing keys do somewhat different things in each of those two cases.

The **backspace** key deletes a block. If the focus is between two commands, the one *before* (above) the blinking bar is deleted. If the focus is on an input slot, the reporter in that slot is deleted. (If that input slot has a default value, it will appear in the slot.) If the focus is on a *variadic* input (one that can change the number of inputs by clicking on arrowheads), then *one* input slot is deleted. (When you right-arrow into a variadic input, the focus first covers the entire thing, including the arrowheads; another right-arrow focuses on the first slot within that input group. The focus is “on the variadic input” when it covers the entire thing.)

The **enter** key does nothing if the focus is between commands, or on a reporter. If the focus is on a variadic input, the enter key adds one more input slot. If the focus is on a white input slot (one that doesn't have a reporter in it), then the enter key selects that input slot for *editing*; that is, you can type into it, just as if you'd clicked on the input slot. (Of course, if the focus is on an input slot containing a reporter, you can use the backspace key to delete that reporter, and then use the enter key to type a value into it.) When you finish typing the value, type the enter key again to accept it and return to navigation, or the escape key if you decide not to change the value already in the slot.

The **space** key is used to see a menu of possibilities for the input slot in focus. It does nothing unless the focus is on a single input slot. If the focus is on a slot with a pulldown menu of options, then the space key shows that menu. (If it's a block-colored slot, meaning that only the choices in the menu can be used, the enter key will do the same thing. But if it's a white slot with a menu, such as in the **turn** blocks, then enter lets you type a value, while space shows the menu.) Otherwise, the space key shows a menu of variables available at this point in the script. In either case, use the up and down arrow keys to navigate the menu, use the enter key to accept the highlighted entry, or use the escape key to leave the menu without choosing an option.

Typing **any other character** key (not special keys on fancy keyboards that do something other than generating a character) activates the *block search palette*. This palette, which is also accessible by typing control-F or command-F outside the keyboard editor, or by clicking the search button floating at the top of the palette, has a text entry field at the top, followed by blocks whose title text includes what you type. The character key you typed to start the block search palette is entered into the text field, so you start with a palette of blocks containing that character. Within the palette, blocks whose titles *start* with the text you type come first, then blocks in which *a word* of the title starts with the text you type, and finally blocks in which the text appears inside a word of the title.



Once you have typed enough text to see the block you want, use the arrow keys to navigate to that block in the palette, then enter to insert that block, or escape to leave the block search palette without inserting the block. (When not in the keyboard editor, instead of navigating with the arrow keys, you drag the block you want into the script, as you would from any other palette.)

If you type an **arithmetic operator** (+-*/[^]) or **comparison operator** (<=>) into the block search text box, you can type an arbitrarily complicated expression, and a collection of arithmetic operator blocks will be constructed to match:



As the example shows, you can also use **parentheses** for grouping, and non-numeric operands are treated as variables or primitive functions. (A variable name entered in this way may or may not already exist in the script. Only **round** and the ones in the pulldown menu of the **sqrt** block can be used as function names.)

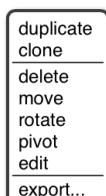
Running the selected script

Type **control-shift-enter** to run the script with the editor focus, like clicking the script.

E. Controls on the Stage

The stage is the area in the top right of the **Snap!** window in which sprites move.

Most sprites can be moved by clicking and dragging them. (If you have unchecked the **draggable** checkbox for a sprite, then dragging it has no effect.) Control-clicking/right-clicking a sprite shows this context menu:



The **duplicate** option makes another sprite with the same scripts, same costumes, etc., as this sprite. The new sprite starts at a randomly chosen position different from the original, so you can see quickly which is which. The new sprite is *selected*: It becomes the current sprite, the one shown in the scripting area. The **clone** option makes a permanent clone of this sprite and selects it.

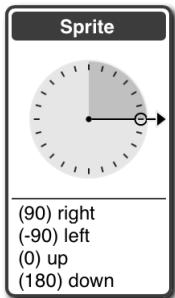
The **delete** option deletes the sprite. It's not just hidden; it's gone for good. The **edit** option selects the sprite. It doesn't actually change anything about the sprite, despite the name; it's just that making changes in the scripting area will change this sprite.

The **move** option shows a “move handle” inside the sprite (the diagonal striped square in the middle):



You can ordinarily just grab and move the sprite without this option, but there are two reasons you might need it: First, it works even if the “draggable” checkbox above the scripting area is unchecked. Second, it works for part sprites relative to their anchor; ordinarily, dragging a part moves the entire nested sprite.

The **rotate** option displays a rotation menu:



You can choose one of the four compass directions in the lower part (the same as in the **point in direction** block) or use the mouse to rotate the handle on the dial in 15° increments.

The **pivot** option shows a crosshair inside the sprite:



You can click and drag the crosshair anywhere onstage to set the costume’s pivot point. (If you move it outside the sprite, then turning the sprite will revolve as well as rotate it around the pivot.) When done, click on the stage not on the crosshair. Note that, unlike moving the pivot point in the Paint Editor, this technique does not visibly move the sprite on the stage. Instead, the values of **x position** and **y position** will change.

The **edit** option makes this the selected sprite, highlighting it in the sprite corral and showing its scripting area. If the sprite was a temporary clone, it becomes permanent.

The **export...** option saves, or opens a new browser tab containing, the XML text representation of the sprite. (Not just its costume, but all of its costumes, scripts, local variables and blocks, and other properties.) You can save this tab into a file on your computer, and later import the sprite into another project. (In some browsers, the sprite is directly saved into a file.)

Control-clicking/right-clicking on the stage background (that is, anywhere on the stage except on a sprite) shows the stage’s own context menu:



The stage’s **edit** option selects the stage, so the stage’s scripts and backgrounds are seen in the scripting area. Note that when the stage is selected, some blocks, especially the Motion ones, are not in the palette area because the stage can’t move.

The **show all** option makes all sprites visible, both in the sense of the **show** block and by bringing the sprite onstage if it has moved past the edge of the stage.

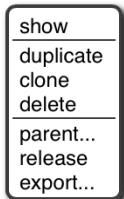
The **pic...** option saves, or opens a browser tab with, a picture of everything on the stage: its background, lines drawn with the pen, and any visible sprites. What you see is what you get. (If you want a picture of just the background, select the stage, open its **costumes** tab, control-click/right-click on a background, and export it.)

The **pen trails** option creates a new costume for the currently selected sprite consisting of all lines drawn on the stage by the pen of any sprite.

F. The Sprite Corral and Sprite Creation Buttons

Between the stage and the sprite corral at the bottom right of the **Snap!** window is a dark grey bar containing three buttons. These are used to create a new sprite. The first button ➤ makes a sprite with just the turtle costume, with a randomly chosen position and pen color. (If you hold down the Shift key while clicking, the new sprite's direction will also be random.) The second button 🖌 makes a sprite and opens the Paint Editor so that you can make your own costume for it. (Of course you could click the first button and then click the paint button in its **costumes** tab; this paint button is a shortcut for all that.) Similarly, the third button 📸 uses your camera, if possible, to make a costume for the new sprite.

In the sprite corral, you click on a sprite's "thumbnail" picture to select that sprite (to make it the one whose scripts, costumes, etc. are shown in the scripting area). You can drag sprite thumbnails (but not the stage one) to reorder them; this has no special effect on your project, but lets you put related ones next to each other, for example. You can right-click/control-click a sprite's thumbnail to get this context menu:



The **show** option makes the sprite visible, if it was hidden, and also brings it onto the stage, if it had moved past the stage boundary. The next three options are the same as in the context menu of the actual sprite on the stage, discussed above.

The **parent...** option displays a menu of all other sprites, showing which if any is this sprite's parent, and allowing you to choose another sprite (replacing any existing parent). The **release** option, which is shown only if this sprite is a (permanent, or it wouldn't be in the sprite corral) clone; it changes the sprite to a temporary clone. (The name is supposed to mean that the sprite is released from the corral.) The **export...** option exports the sprite, like the same option on the stage.

The context menu for the stage thumbnail has only one option, **pic...**, which takes a picture of everything on the stage, just like the same option in the context menu of the stage background.

G. Preloading a Project when Starting Snap!

There are several ways to include a pointer to a project in the URL when starting **Snap!** in order to load a project automatically. You can think of such a URL as just running the project rather than as running **Snap!**, especially if the URL says to start in presentation mode and click the green flag. The general form is

`http://snap.berkeley.edu/run#verb:project&flag&flag...`

The “verb” above can be any of `open`, `run`, `cloud`, `present`, or `dl`. The last three are for shared projects in the **Snap!** cloud; the first two are for projects that have been exported and made available anywhere on the Internet.

Here’s an example that loads a project stored at the **Snap!** web site (not the **Snap!** cloud!):

```
http://snap.berkeley.edu/run#open:http://snap.berkeley.edu/snapsource/Examples/vee.xml
```

The project file will be opened, and **Snap!** will start in edit mode (with the program visible). Using `#run`: instead of `#open`: will start in presentation mode (with only the stage visible) and will “start” the project by clicking the green flag. (“Start” is in quotation marks because there is no guarantee that the project includes any scripts triggered by the green flag. Some projects are started by typing on the keyboard or by clicking a sprite.)

If the verb is `run`, then you can also use any subset of the following flags:

<code>&editMode</code>	Start in edit mode, not presentation mode.
<code>&noRun</code>	Don’t click the green flag.
<code>&hideControls</code>	Don’t show the row of buttons above the stage (edit mode, green flag, pause, stop).
<code>&lang=fr</code>	Set language to (in this example) French.
<code>&noExitWarning</code>	When closing the window or loading a different URL, don’t show the browser “are you sure you want to leave this page” message.

The last of these flags is intended for use on a web page in which a **Snap!** window is embedded.

Here’s an example that loads a shared (public) project from the **Snap!** cloud:

```
http://snap.berkeley.edu/run#present:Username=jens&ProjectName=tree%20animation
```

(Note that “`Username`” and “`ProjectName`” are TitleCased, even though the flags such as “`noRun`” are camelCased. Note also that a space in the project name must be represented in Unicode as `%20`.) The verb `present` behaves like `run`: it ordinarily starts the project in presentation mode, but its behavior can be modified with the same four flags as for `run`. The verb `cloud` (yes, we know it’s not a verb in its ordinary use) behaves like `open` except that it loads from the **Snap!** cloud rather than from the Internet in general. The verb `dl` (short for “download”) does not start **Snap!** but just downloads a cloud-saved project to your computer as an `.xml` file. This is useful for debugging; sometimes a defective project that **Snap!** won’t run can be downloaded, edited, and then re-saved to the cloud.

H. Mirror Sites

If the site `snap.berkeley.edu` is ever unavailable, you can load **Snap!** at any of the following mirror sites:

- <http://bjc.edc.org/snapsource/snap.html>
- <http://media.mit.edu/~harveyb/snap>
- <http://cs10.org/snap>

Index

! block · 90
variable · 87
#1 · 50

A

a new clone of block · 58
Abelson, Hal · 3
About option · 82
add comment option · 105
Advanced Placement Computer Science Principles · 86
AGPL · 82
all but blocks · 88
all but first of block · 34
all but first of stream · 87
all of block · 87
Alonzo · 8
anchor · 9
anchor (in my block) · 59
animate block · 92
animation · 11
animation library · 92
anonymous list · 31
Any (unevaluated) type · 53
any of block · 87
Any type · 41
append block · 87
Arduino · 73
arithmetic · 10
array, dynamic · 34
arrow, upward-pointing · 44
arrowheads · 31, 44
ask block · 67
assoc block · 87
association list · 69
associative function · 36
at block · 19
attribute · 57
attributes, list of · 59

B

background blocks · 21
Backgrounds... option · 96
backspace key (keyboard editor) · 111
Ball, Michael · 3
base case · 29
bigger data library · 93
BIGNUMS block · 90
binary tree · 32

bitmap · 96
bjc.edc.org · 115
block · 5; command · 5; C-shaped · 6; hat · 5; predicate · 11; reporter · 9; sprite-local · 56
Block Editor · 26, 27, 40
block library · 30, 85
block picture option · 104
block shapes · 25, 41
block variable · 28
block with no name · 90
blocks, color of · 25
Boole, George · 11
Boolean · 11
Boolean (unevaluated) type · 53
Boolean constant · 11
break command · 80
breakpoint · 16, 100
broadcast and wait block · 8
broadcast block · 54
brown dot · 8
Build Your Own Blocks · 25
button: pause · 16; recover · 24; visible stepping · 17

C

C programming language · 49
call block · 46, 49
call w/continuation block · 78
camera icon · 106
Cancel button · 109
carriage return character · 19
cascade blocks · 86
case-independent comparisons block · 94
cases block · 88
catch block · 80
center of the stage · 20
center x (in my block) · 59
center y (in my block) · 59
Chandra, Kartik · 3
change crayon by block · 91
Change password... option · 97
child class · 68
children (in my block) · 59
Church, Alonzo · 8
class · 66
class/instance · 58
clean up option · 105
clear button · 108
clicking on a script · 103
Clicking sound option · 99
clone of block · 70
clones: permanent · 55; temporary · 55
clones (in my block) · 59
cloud (startup option) · 115
Cloud button · 22, 83

cloud icon · 96
cloud storage · 22
codification support option · 99
color of blocks · 25
color palette · 108
combine block · 35
command block · 5
comma-separated values · 39
comment box · 105
compose block · 86
Computer Science Principles · 86
cond in Lisp · 88
conditional library: multiple-branch · 88
constant functions · 53
constructors · 32
context menu · 101
context menu for the palette background · 102
context menus for palette blocks · 101
continuation · 74
continuation passing style · 75
Control palette · 6
controls in the Costumes tab · 106
controls in the Sounds tab · 109
controls on the stage · 112
control-shift-enter (keyboard editor) · 112
CORS · 73
cors proxies · 73
costume · 5, 7
costumes (in my block) · 59
Costumes tab · 8, 106
costumes, first class · 60
Costumes... option · 95
counter class · 66
CPS · 77
crayon library · 91
Cross-Origin Resource Sharing · 73
crossproduct · 51
cs10.org · 115
C-shaped block · 6, 48
C-shaped slot · 53
CSV · 39
csv format · 19
current block · 73
current date or time · 73
current location block · 88
current sprite · 103
custom block in a script · 105

D

dangling rotation · 9
dangling? (in my block) · 59
data hiding · 54
data structure · 32
data table · 69
data type · 19
data types · 40
date · 73
Dave, Achal · 3

Debugging · 16, 100
default value · 44
delegation · 68
Delete a variable · 13
delete block definition... option · 102
delete option · 104, 108, 112
design principle · 31, 58
devices · 72, 73
dialog, input name · 27
Dinsmore, Nathan · 3
direction to block · 20
dispatch procedure · 66, 67, 69
distance to block · 20
dl (startup option) · 115
do in parallel block · 94
does variable exist block · 94
down arrow (keyboard editor) · 110
Download source option · 83
draggable checkbox · 103, 112
duplicate option · 104, 108, 112
dynamic array · 34

E

easing block · 92
easing function · 92
edge color · 109
edit option · 108, 112, 113
edit... option · 102
editMode (startup option) · 115
ellipse tool · 108, 109
ellipsis · 44
else block · 88
else if block · 88
empty input slots, filling · 47, 49, 51
empty? block · 87
enter key (keyboard editor) · 111
eraser tool · 108
error block · 89
error catching library · 89
escape key (keyboard editor) · 110
Examples button · 83
Execute on slider change option · 98
Export blocks... option · 85
export option · 108, 113
Export project... option · 85, 86
export... option · 114
expression · 10
eyedropper tool · 108, 109

F

factorial · 29, 52
factorial function · 90
false block · 18
file icon menu · 83
fill color · 109

Finch · 73
find blocks... option · 102
find first · 35
first blocks · 88
first class data type · 31
first class procedures · 46
first class sprites · 54
flag, green · 5
Flat design option · 99
flat line ends option · 99
floodfill tool, · 108
focus (keyboard editor) · 111
footprint button · 100
for block · 12, 18, 44, 46
for block · 87
For this sprite only · 14
formal parameters · 50
frequency distribution analysis library · 93
function, associative · 36
function, higher order · 35
functional programming style · 33

G

generic hat block · 5
getter · 57
getter/setter library · 89
glide block · 98
global variable · 13, 14
global variables block · 94
go to block · 20
green flag · 5
green flag button · 100
green halo · 103

H

halo · 10, 103; red · 50
hat block · 5, 26; generic · 5
help... option · 101, 104
help... option for custom block · 102
hexagonal blocks · 26, 41
hexagonal shape · 11
hide and show primitives · 16
hide option · 101
hide primitives option · 102
hide variable block · 16
hideControls (startup option) · 115
higher order function · 35, 51
higher order procedure · 47
hint · 45
histogram · 93
Hotchkiss, Kyle · 3
HSV block · 89
HTML (HyperText Markup Language) · 72
HTTP · 73
HTTPS · 73, 107

Hudson, Connor · 3
Hummingbird · 73
Hz for block · 93

I

ice cream · 84
icons in title text · 45
id block · 53
identity function · 53
if block · 11
if else block · 52
if else reporter block · 18
imperative programming style · 33
import... option · 85
in front of block · 34
in front of stream · 87
index variable · 18
infinite precision integer library · 90
Ingalls, Dan · 3
inherit block · 59
inheritance · 54, 68
input · 5
input list · 49
input name · 50
input name dialog · 27, 40
input sliders option · 98
input-type shapes · 40
instance · 66
interaction · 14
internal variable · 44
is_a_? block · 19
item 1 of block · 34
item 1 of stream · 87
iteration library · 86

J

Java programming language · 49
JavaScript · 18
jigsaw-piece blocks · 25, 41
json format · 19
JSON library · 94
jukebox · 8

K

Kay, Alan · 3
keep from stream · 87
key:value: block · 88
keyboard: piano · 20
keyboard editing button · 104
keyboard editor · 110
keyboard shortcuts · 83
key-value pair · 69

L

label block · 18
lambda · 48
lang= (startup option) · 115
Language... option · 97
last blocks · 88
layout, window · 4
Leap Motion · 73
LEAP Motion library · 88
left arrow (keyboard editor) · 110
Lego NXT · 73
let block · 89
lexical scope · 66
Libraries... option · 86
library: block · 30
license · 82
Lifelong Kindergarten Group · 3
line break in block · 45
line drawing tool · 108
linked list · 34
list block · 31
list library · 87
list of procedures · 52
List type · 41
list view · 36
list, linked · 34
listify block · 94
lists of lists · 32
little people · 77
little person · 29
loading saved projects · 23
local state · 54
location-pin · 14
Login... option · 97
Logout option · 97
Long form input dialog option · 98
long input name dialog · 40

message passing · 54, 67
method · 54, 56, 67
methods table · 69
middle option · 107
mirror sites · 115
MIT Artificial Intelligence Lab · 3
MIT Media Lab · 3
monadic negation operator · 21
Morphic · 3
Motyashov, Ivan · 3
move option · 112
multiline block · 89
multi-line library · 89
multiple input · 44
Multiple inputs · 44
multiple-branch conditional library · 88
mutation · 33
mutators · 32
my block · 54, 57

N

name: input · 50
name (in **my** block) · 59
name box · 103
negation operator · 21
neighbors (in **my** block) · 59
nested calls · 51
Nesting Sprites · 9
new line character · 45
New option · 83
new sprite button · 7
newline character · 19
Nintendo · 73
noExitWarning (startup option) · 115
nonlocal exit · 80
noRun (startup option) · 115
Number type · 41

M

Make a block · 25
Make a block button · 101
make a block... option · 106
Make a list · 31
Make a variable · 13
make internal variable visible · 44
Maloney, John · 3
map block · 35, 46
map over stream · 87
map to code block · 99
map-pin symbol · 56
Massachusetts Institute of Technology · 3
McCarthy, John · 3
Media Lab · 3
media.mit.edu · 115
memory · 15
message · 54

object block · 54
object oriented programming · 54, 66
Object type · 41
objects, building explicitly · 66
of block · 21
open (startup option) · 115
Open... option · 83
orange oval · 12
other clones (in **my** block) · 59
other sprites (in **my** block) · 59
outlined ellipse tool · 108
outlined rectangle tool · 108
oval blocks · 25, 41

P

paint brush icon · 106
Paint Editor · 106
Paint Editor window · 108
paintbrush tool · 108
palette · 5
palette area · 101
palette background · 102
Parallax S2 · 73
parallelism · 7, 33
parallelization library · 94
parent (in **my** block) · 59
parent attribute · 58
parent class · 68
parent... option · 114
partial compilation · 93
parts (in **my** block) · 59
parts (of nested sprite) · 9
pause all block · 16, 100
pause button · 16, 100
pen color library · 89
pen trails block · 18
pen trails option · 114
permanent clone · 114
permanent clones · 55
physical devices · 72
piano keyboard · 20
pic... option · 114
pivot option · 113
pixel, screen · 19
pixels library · 92
Plain prototype labels option · 98
play block · 93
play sound block · 8
playing sounds · 8
plot sound block · 93
point towards block · 20
points as inputs · 20
polymorphism · 56
position block · 94
Predicate block · 11
preloading a project · 114
present (startup option) · 115
presentation mode button · 100
primitive block within a script · 104
procedure · 11, 47
Procedure type · 53
procedures as data · 8
product block · 87
project control buttons · 100
Project notes option · 83
prototype · 26
prototype hint · 45
prototyping · 57, 58, 69
pulldown input · 42

R

read-only pulldown input · 42
recover button · 24
rectangle tool · 108
recursion · 28
recursive call · 49
recursive operator · 52
red halo · 49, 50, 104
redo button · 104
redrop option · 105
Reference manual option · 83
relabel... option · 104, 105
release option · 114
remove duplicates from block · 87
rename option · 108
renaming variables · 14
repeat block · 6, 48
repeat blocks · 86
repeat until block · 11
report block · 29
Reporter block · 9
reporter **if** block · 11
reporter **if else** block · 18
reporters, recursive · 29
Reset Password... option · 97
Retina display support option · 97
reverse block · 87
Reynolds, Ian · 3
RGB block · 89
right arrow (keyboard editor) · 110
ring · 47, 49
ring, gray · 34
ringify · 47
ringify option · 105
Roberts, Eric · 29
robots · 72, 73
rotation · 103
rotation buttons · 103
rotation point tool · 108, 109
rotation x (in **my** block) · 59
rotation y (in **my** block) · 59
run (startup option) · 115
run block · 46, 49
run w/continuation · 80
run w/continuation block · 79

S

safely try block · 89
Save as... option · 85
Save option · 85
save your project in the cloud · 22
Scheme · 3
Scheme number block · 90
scope: lexical · 66
Scratch · 4, 8, 25, 31, 32, 33, 40

Scratch Team · 3
 screen pixel · 19
 script · 4
 script pic · 28
`script pic...` option · 104
script variables block · 14, 67
 scripting area · 5, 103
 scripting area background context menu · 105
`scripts pic...` option · 106
 search bar · 84
 search button · 101
 secrets · 82
 selectors · 32
`self (in my block)` · 59
 sensors · 72
 sentence library · 88
 separator: menu · 43
 Servilla, Dylan · 3
set block · 14
set flag block · 89
set pen color blocks · 89
set pen to crayon block · 91
set value block · 89
 setter · 57
setting block · 89
 settings icon · 97
 shapes of blocks · 25
 shift-arrow keys (keyboard editor) · 111
 Shift-click (keyboard editor) · 110
 shift-clicking · 82
 shift-enter (keyboard editor) · 110
 Shift-tab (keyboard editor) · 110
 shortcut · 106, 114
 shortcuts: keyboard · 83
show all option · 113
show option · 114
show primitives option · 102
show stream · 87
show variable block · 16
shrink/grow button · 100
sieve · 87
Signup... option · 97
 simulation · 54
 Single input · 44
 single stepping · 17
 slider: stepping speed · 17
 Snap!logo menu · 82
 Snap!manual · 104
 Snap! program · 4
Snap! website option · 83
 snap.berkeley.edu · 83
 solid ellipse tool · 108
 solid rectangle tool · 108
 sophistication · 53
sort block · 87
 sound manipulation library · 93
sounds (in my block) · 59
 sounds, first class · 60
Sounds... option · 96
 source files for Snap! · 83
 space key (keyboard editor) · 111
 special form · 53
split block · 19, 72
 sprite · 5
 sprite appearance and behavior controls · 103
 sprite corral · 7, 114
 sprite creation buttons · 114
 sprite nesting · 9
 sprite-local block · 56
 sprite-local variable · 13, 14
sprite-local variables block · 94
 sprites · 54
 spiral · 12
 stack of blocks · 5
 stage · 5, 54
stage (in my block) · 59
 stage blocks · 21
 Stage resizing buttons · 100
Stage size... option · 97
 Stanford Artificial Intelligence Lab · 3
 starting Snap! · 114
 Steele, Guy · 3
stop all block · 100
stop block · 20
stop block block · 29
stop button · 100
stop script block · 29
 stop sign · 7
Stream · 87
 stream library · 87
 string processing library · 94
Structure and Interpretation of Computer Programs · 3
 submenu · 43
substring block · 94
sum block · 87
 Super-Awesome Sylvia · 73
 Sussman, Gerald J. · 3
 Sussman, Julie · 3
switch in C · 88
 symbols in title text · 45
 synchronous rotation · 9
 system getter/setter library · 89

T

tab character · 19
 tab key (keyboard editor) · 110
 table view · 36
 temporary clone · 113
 temporary clones · 55
 Terms of Service · 23
 text input · 8
 Text type · 41
 text-based language · 99
Thinking Recursively · 29
 thread · 81
thread block · 81

Thread safe scripts option · 99

thumbnail · 103

time · 73

tip option · 107

title text · 27

to block · 20

tool bar · 5

tool bar features · 82

touching block · 20

transient variable · 15

translation · 97

translations · 28

transparent paint · 109

true block · 18

Turbo mode option · 98

Turtle costume · 8, 106

turtle's rotation point · 107

two-item (x,y) lists · 20

type · 19

U

undo button · 104, 108

undrop option · 105

unevaluated procedure types · 42

unevaluated type · 53

Uniform Resource Locator · 72

unringify · 47, 67

unringify option · 105

Unused blocks... option · 85

up arrow (keyboard editor) · 110

upvar · 44

upward-pointing arrow · 44

url block · 72, 88

USE BIGNUMS block · 90

use case-independent comparisons block · 94

user interface elements · 82

user name · 22

V

value at key block · 94

variable · 12, 57; block · 28; global · 13; renaming · 14; script-local ·

14; sprite-local · 13, 14; transient · 15

variable watcher · 13

variable-input slot · 49

variables in ring slots · 47

variables library · 94

variadic input · 31, 44

variadic library · 87

vector · 96

vector editor · 109

visible stepping · 100

visible stepping button · 17

visible stepping option · 98

W

wardrobe · 8

warp block · 103

watcher · 14

Water Color Bot · 73

web services library · 88

when I am block · 18

whitespace · 19

Wiimote · 73

window layout · 4

with inputs · 47

word and sentence library · 88

World Wide Web · 72

writeable pulldown inputs · 42

X

X position · 10

Xerox PARC · 3

Y

Y position · 10

yield block · 81

Yuan, Yuan · 3

Z

zebra coloring · 10

Zoom blocks... option · 97
