A. Sprites and Parallelism

Topics of Section {: .text-delta } 1. TOC {:toc}

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 spr/te 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:

photo

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:

photo

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 128. 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:

photo

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 or 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: 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 . In contrast, you can use the 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:

photo

In the 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." (But if you click the right arrow after the message name, the block becomes , and you can change it to to send the message just to one sprite.) 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. In Chapter VII,

"Object-Oriented Programming with Sprites," you'll see a more flexible way to send a message to a specific sprite using the tell and ask blocks. 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 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 if the text is displayed.

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.

Topics of Section {: .text-delta } 1. TOC {:toc}

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. The precise place where you let go of the mouse button will be the attachment point of the part on the anchor.

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

photo

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 the 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. Also, cloning the anchor (see Section VII. B) will also clone all its parts.

photo

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

Topics of Section {: .text-delta } 1. TOC {:toc}

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: . 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:

photo

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:

photo

Don't drop the input over a red halo:

photo

That's used for a purpose explained on page 68.

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

photo

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:

photo

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.)

photo

A reporter block with its inputs, maybe including other reporter blocks, such as , is called an expression.

title: D. Predicates and Conditional Evaluation layout: home photo: parent: Chapter 1 - Blocks, Scripts, and Sprites has_toc: true —

Topics of Section {: .text-delta } 1. TOC {:toc}

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

photo

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 , 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:

photo

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

photo

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

photo

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

photo

Sometimes you want to take the same action whether some condition is t

photo

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:

title: E. Variables layout: home photo: parent: Chapter 1 - Blocks, Scripts, and Sprites has_toc: true —

Topics of Section {: .text-delta } 1. TOC {:toc}

Try this script:

photo

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:

photo

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:

photo

"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:

photo

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

photo

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:

photo

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:

photo

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:

photo

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:

photo

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:

photo

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:

photo

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:

photo

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):

photo

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 temporarily 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:

photo

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.

Topics of Section {: .text-delta } 1. TOC {:toc}

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 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:

photo

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 25 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:

photo

(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

photo

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:

photo

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.

title: G. Etcetera layout: home photo: parent: Chapter 1 - Blocks, Scripts, and Sprites has_toc: true —

Topics of Section {: .text-delta } 1. TOC {:toc}

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:

reports a new costume consisting of everything that's drawn on the stage by any sprite. Rightclicking the block in the scripting area gives the option to change it to if vector logging is enabled. See page 116.

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. (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.

Takes a sprite as input. Like stamp except that the costume is stamped onto the selected sprite instead of onto the stage. (Does nothing if the current sprite doesn't overlap the chosen sprite.)

Takes a sprite as input. Erases from that sprite's costume the area that overlaps with the current sprite's costume. (Does not affect the costume in the chosen sprite's wardrobe, only the copy currently visible.)

See page 6.

See page 17.

Runs only this script until finished. In the Control palette even though it's gray

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.

Declare local variables in a script.

See page 91.

reports the value of a graphics effect.

Constant true or false value. See page 12.

Create a primitive using JavaScript. (This block is disabled by default; the user must check "Javascript extensions" in the setting menu each time a project is loaded.)

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 five items of the left menu let you examine the color

visible at the position. (The "RGBA" option reports a list.) 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). This is a hyperblock with respect to its second input.

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. "Le!er" 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 54. "Blocks" takes a script as the first input, reporting a list structure representing the structure of the script. See Chapter XI.

For lists, reports true only if its two input values are the very same list, so changing an item in one of them is visible in the other. (For =, lists that look the same are the same.) For text strings, uses case-sensitive comparison, unlike =, which is case-independent

For lists, reports true only if its two input values are the very same list, so changing an item in one of them is visible in the other. (For =, lists that look the same are the same.) For text strings, uses case-sensitive comparison, unlike =, which is case-independent.

These hidden blocks can be found with the relabel option of any dyadic arithmetic block. They're hidden partly because writing them in Snap! is a good, pretty easy programming exercise. Note: the two inputs to atan2 are Δx and Δy in that order, because we measure angles clockwise from north. Max and min are variadic; by clicking the arrowhead, you can provide additional inputs.

Similarly, these hidden predicates can be found by relabeling the relational predicates.

Metaprogramming (see Chapter XI. , page 101)

These blocks support metaprogramming, which means manipulating blocks and scripts as data. This is not the same as manipulating procedures (see Chapter VI.), which are what the blocks mean; in metaprogramming the actual blocks, what you see on the screen, are the data. This capability is new in version 8.0.

First class list blocks (see Chapter IV, page 46)

Numbers from will count up or down.

The script input to for each can refer to an item of the list with the item variable.

report the sprite or mouse position as a two-item vector (x,y).

First class procedure blocks (see Chapter VI, page 65)

photo

First class continuation blocks (see Chapter X, page 93)

photo

First class sprite, costume, and sound blocks (see Chapter VII, page 73)

photo

Object is a hyperblock

Scenes

The major new feature of version 7.0 is scenes: A project can include within it sub-projects, called scenes, each with its own stage, sprites, scripts, and so on. This block makes another scene active, replacing the current one. Nothing is automatically shared between scenes: no sprites, no blocks, no variables. But the old scene can send a message to the new one, to start it running, with optional payload as in broadcast (page 23).

In particular, you can say

photo

if the new scene expects to be started with a green flag signal.

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

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

photo

"Center" means the center of the stage, the point at (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. "Ray length" is the distance from the center of this sprite to the nearest point on the other sprite, in the current direction.

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.

he video block has a snap option that takes a snapshot and reports it as a costume. It is hyperized with respect to its second input.

The "neg" option is a monadic negation operator, equivalent to . "lg" is log2. "id" is the identity function, which reports its input. "sign" reports 1 for positive input, 0 for zero input, or -1 for negative input.

name changed to clarify that it's different from

photo

- and × are variadic: they take two or more inputs. If you drop a list on the arrowheads, the block name changes to sum or product.

photo

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.

photo

Extended broadcast: Click the right arrowhead to direct the message to a single sprite or the stage. Click again to add any value as a payload to the message.

photo

Extended when I receive: Click the right arrowhead to expose a script variable (click on it to change its name, like any script variable) that will be set to the data of a matching broadcast. If the first input is set to "any message," then the data variable will be set to the message, if no payload is included with the broadcast, or to a two-item list containing the message and the payload.

photo

If the input is set to "any key," then a right arrowhead appears:

photo

and if you click it, a script variable key is created whose value is the key that was pressed. (If the key is one that' represented in the input menu by a word or phrase, e.g., "enter" or "up arrow," then the value of key will be that word or phrase, except for the space character, which is represented as itself in key.)

photo

These ask features and more in the Menus library.

The of block has an extended menu of attributes of a sprite. Position reports an (x,y) vector. Size reports the percentage of normal size, as controlled by the set size block in the Looks category. Left, right, etc. report the stage coordinates of the corresponding edge of the sprite's bounding box. Variables reports a list of the names of all variables in scope (global, sprite-local, and script variables if the right input is a script).

Topics of Section {: .text-delta} 1. TOC {:toc}

{:title }

This chapter describes the Snap! features inherited from Scratch; experienced Scratch users can skip to Section 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:

photo

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:

photo

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 5, 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 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.

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.

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:

title: A. Local Storage layout: home photo: parent: Chapter 2 - Saving and Loading Projects and Media has_toc: true —

Topics of Section {: .text-delta } 1. TOC {:toc}

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.

title: B. Creating a Cloud Account layout: home photo: parent: Chapter 2 - Saving and Loading Projects and Media has_toc: true —

Topics of Section {: .text-delta } 1. TOC {:toc}

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 ( ) in the Tool Bar. Choose the "Signup..." option. This will show you a window that looks like the picture at the right.

photo

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.)

Topics of Section {: .text-delta} 1. TOC {:toc}

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:

photo

(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. # just-the-docs-template

This is a *bare-minimum* template to create a Jekyll site that:

- uses the Just the Docs theme;
- can be built and published on GitHub Pages;
- can be built and previewed locally, and published on other platforms.

More specifically, the created site:

- uses a gem-based approach, i.e. uses a `Gemfile` and loads the `just-the-docs` gem;
- uses the GitHub Pages / Actions workflow to build and publish the site on GitHub Pages.

To get started with creating a site, simply:

1. click "use this template" to create a GitHub repository
2. go to Settings > Pages > Build and deployment > Source, and select GitHub Actions

If you want to maintain your docs in the `docs` directory of an existing project repo, see Hosting your docs from an existing project repo.

After completing the creation of your new site on GitHub, update it as needed:

Replace the content of the template pages

Update the following files to your own content:

- `index.md` (your new home page)
- `README.md` (information for those who access your site repo on GitHub)

Changing the version of the theme and/or Jekyll

Simply edit the relevant line(s) in the `Gemfile`.

Adding a plugin

The Just the Docs theme automatically includes the `jekyll-seo-tag` plugin.

To add an extra plugin, you need to add it in the `Gemfile` *and* in `_config.yml`. For example, to add `jekyll-default-layout`:

- Add the following to your site's `Gemfile`:

  ```
  gem "jekyll-default-layout"
  ```

- And add the following to your site's `_config.yml`:

  ```
  plugins:
    - jekyll-default-layout
  ```

Note: If you are using a Jekyll version less than 3.5.0, use the `gems` key instead of `plugins`.

Publishing your site on GitHub Pages

1. If your created site is `YOUR-USERNAME/YOUR-SITE-NAME`, update `_config.yml` to:

   ```
   title: YOUR TITLE
   description: YOUR DESCRIPTION
   theme: just-the-docs

   url: https://YOUR-USERNAME.github.io/YOUR-SITE-NAME

   aux_links: # remove if you don't want this link to appear on your pages
     Template Repository: https://github.com/YOUR-USERNAME/YOUR-SITE-NAME
   ```

2. Push your updated `_config.yml` to your site on GitHub.
3. In your newly created repo on GitHub:
   - go to the `Settings` tab -> `Pages` -> `Build and deployment`, then select `Source`: `GitHub Actions`.
   - if there were any failed Actions, go to the `Actions` tab and click on `Re-run jobs`.

Building and previewing your site locally

Assuming Jekyll and Bundler are installed on your computer:

1. Change your working directory to the root directory of your site.
2. Run `bundle install`.
3. Run `bundle exec jekyll serve` to build your site and preview it at `localhost:4000`.

   The built site is stored in the directory `_site`.

Publishing your built site on a different platform

Just upload all the files in the directory `_site`.

Customization

You're free to customize sites that you create with this template, however you like!

Browse our documentation to learn more about how to use this theme.

Hosting your docs from an existing project repo

You might want to maintain your docs in an existing project repo. Instead of creating a new repo using the just-the-docs template, you can copy the template files into your existing repo and configure the template's Github Actions workflow to build from a `docs` directory. You can clone the template to your local machine or download the `.zip` file to access the files.

Copy the template files

1. Create a `.github/workflows` directory at your project root if your repo doesn't already have one. Copy the `pages.yml` file into this directory. GitHub Actions searches this directory for workflow files.
2. Create a `docs` directory at your project root and copy all remaining template files into this directory.

Modify the GitHub Actions workflow

The GitHub Actions workflow that builds and deploys your site to Github Pages is defined by the `pages.yml` file. You'll need to edit this file to that so that your build and deploy steps look to your `docs` directory, rather than the project root.

1. Set the default `working-directory` param for the build job.

```yaml
build:
  runs-on: ubuntu-latest
  defaults:
    run:
      working-directory: docs
```

2. Set the `working-directory` param for the Setup Ruby step.

```yaml
- name: Setup Ruby
    uses: ruby/setup-ruby@v1
    with:
      ruby-version: '3.1'
      bundler-cache: true
      cache-version: 0
      working-directory: '${{ github.workspace }}/docs'
```

3. Set the path param for the Upload artifact step:

```
- name: Upload artifact
    uses: actions/upload-pages-artifact@v1
    with:
      path: "docs/_site/"
```

4. Modify the trigger so that only changes within the `docs` directory start the workflow. Otherwise, every change to your project (even those that don't affect the docs) would trigger a new site build and deploy.

```
on:
  push:
    branches:
      - "main"
    paths:
      - "docs/**"
```

Licensing and Attribution

This repository is licensed under the MIT License. You are generally free to reuse or extend upon this code as you see fit; just include the original copy of the license (which is preserved when you "make a template"). While it's not necessary, we'd love to hear from you if you do use this template, and how we can improve it for future use!

The deployment GitHub Actions workflow is heavily based on GitHub's mixed-party starter workflows. A copy of their MIT License is available in actions/starter-workflows.

This is a *bare-minimum* template to create a Jekyll site that uses the Just the Docs theme. You can easily set the created site to be published on GitHub Pages – the README file explains how to do that, along with other details.

If Jekyll is installed on your computer, you can also build and preview the created site *locally*. This lets you test changes before committing them, and avoids waiting for GitHub Pages.[1] And you will be able to deploy your local build to a different platform than GitHub Pages.

More specifically, the created site:

- uses a gem-based approach, i.e. uses a `Gemfile` and loads the `just-the-docs` gem
- uses the GitHub Pages / Actions workflow to build and publish the site on GitHub Pages

Other than that, you're free to customize sites that you create with this template, however you like. You can easily change the versions of `just-the-docs` and

[1] It can take up to 10 minutes for changes to your site to publish after you push the changes to GitHub.

Jekyll it uses, as well as adding further plugins.

Browse our documentation to learn more about how to use this theme.

To get started with creating a site, simply:

1. click “use this template” to create a GitHub repository
2. go to Settings > Pages > Build and deployment > Source, and select GitHub Actions

If you want to maintain your docs in the `docs` directory of an existing project repo, see Hosting your docs from an existing project repo in the template README.
