# The Beauty and Joy of Computing[1]

*Lab Exercise 7: Send in the clones*

## Objectives

By completing this lab exercise, you should learn to

- Use the sprite cloning functionality of BYOB to create many sprites for complex animations;
- Access variables and scripts in other sprites; and
- Understand the basics of object-oriented programming, with object attributes and object scripts.

## Background (Pre-Lab Reading)

You should read this section before coming to the lab. It describes how various things work in BYOB and provides pictures to show what they look like in BYOB. You only need to read this material to get familiar with it. There is no need to actually do the actions described in this section, but you certainly can do them if it would help you follow along with the examples.

### Object-Oriented Programming Concepts and Terminology

In the previous labs, we have worked with a small number of sprites, and every sprite was specifically created and programmed through scripts that operated on that sprite. Scripts can create interesting and complex actions for individual sprites, but we often want to have many sprites. Think about an interesting game: there are often tens or hundreds or even thousands of characters that you must keep track of. In addition to characters, we might also have other game items such as weapons, clothing, buildings, that we also need to track. We obviously don't want to create a specific sprite for each entity in a game in which there are thousands of characters, and what saves us is the fact that many items share a similar structure and set of actions that they can perform. A style of program design that handles this very nicely is called **object-oriented programming**. While a full explanation of object-oriented design is beyond what we will do in this class, the basics are easy to understand.

Object-oriented programming is organized around **classes**, which are programming elements that share certain characteristics. Classes might represent things that you could potentially see, like characters in an animation, players or weapons in a game, or bigger program components such as backgrounds. However, classes can also represent more abstract things non-visible things like fractions or matrices or student lists. In a fighting game we might have a class for something as general as weapons, or for something slightly more specific like daggers which we might refer to as the "Dagger" class. **Objects** are specific instances of a class, so the dagger that your player is holding could be an instance of the Dagger class. If there is another player holding a different dagger, then that might be a different instance of the same general class. For another example, in BYOB consider that sprites are a class, and when you create a specific

---

[1] *Lab Exercises for "The Beauty and Joy of Computing"*

sprite (such as Alonzo) you have just created an instance of the sprite class. Note that you can create multiple sprites that look exactly alike - multiple Alonzos, for instance, but these are in fact separate objects.

It object-oriented programming, every object can contain attributes and methods:

- An **attribute** is a variable that is associated with one particular object. For example, an object from the “Dagger” class might have attributes such as weight, size, and sharpness. The collection of all attribute values of an object is referred to as the **state** of the object.
- A **method** is a function or a script that causes an object to perform some action. A method can access or change the attributes associate with this object, or it can perform some other action. For example, our Dagger class might have methods such as throw or stab, reflecting game play actions, or a draw method that draws it on the screen. A special kind of method called a **constructor** defines the actions required to initialize all of the attributes of an object - when an object is created, it creates variables for all of its attributes, and just like any variables these should be initialized!

BYOB sprites are good examples of objects: the attributes for a sprite object include things like its current position (x and y coordinates), direction, and whether it is visible, and the methods include blocks that operate on sprites such as the “move” and “glide” blocks. The correct way to think about these actions from an object-oriented standpoint is that they modify the state of the sprite object by changing the x and y coordinate attributes of the object.

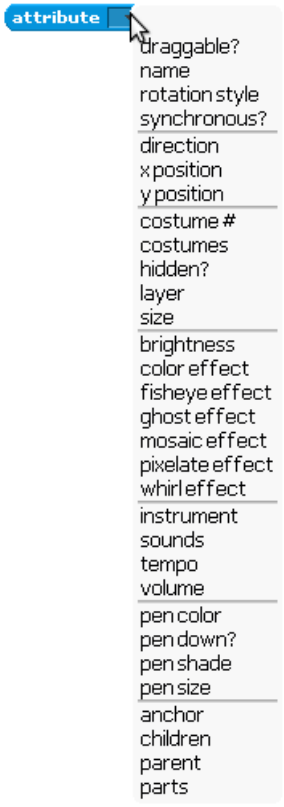
There are several different styles of object-oriented programming - for example, while Java and JavaScript are common programming languages with similar names, they support distinctly different styles of object-oriented programming. BYOB supports a particular style of object-oriented programming called **prototype-based programming** in which new instances of objects are created by cloning an existing object (called the **prototype**), and this is the style of object-oriented programming that we will focus on in this class. There’s obviously a lot more to object-oriented programming than has was just described - we’ll talk about a few other issues in the Post-Lab Reading, but for now let’s look at how the basics work by working through an example.

### Working with Objects

In this section, we will go through a complete example showing how to set up a class with both attributes and methods, and create objects that are instances of that class by cloning a prototype object. For this example, let’s make a silly animation: we want cars to drive around on the stage, bouncing against the edges. A new car is created every time a car runs over a magic hat that moves randomly around the stage, and there are two bombs on the stage that also move around randomly and any car that runs into a bomb is destroyed. Since the number of cars is controlled by luck (where the hat and bombs are randomly placed), we don’t know how many cars we need, so we can’t just create all the sprites manually at the beginning. Just because they are all cars and share some characteristics doesn’t mean they will all look the same - there are a few different “styles” of cars, and the style of a car is randomly chosen when it is created.

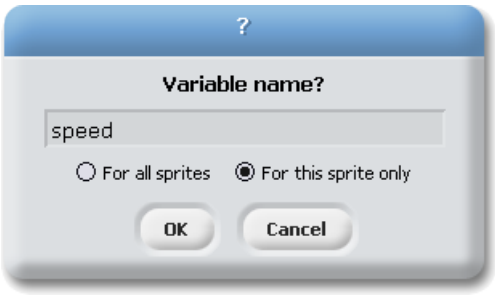
From an object-oriented programming standpoint, we want to define a class for cars, and then create a new car object when necessary. One car object is created when the animation starts, and a new car object is created every time an existing car hits the magic hat. Each car object will have all of the attributes that are common to all sprites (position, direction, etc.), and will have two attributes that are specific to our car definition: a speed and a style. As a first step, we create a new sprite with a car costume, and then import three other costumes in the costumes tab for the other styles. We'll change the name of this sprite to "CarProto" to indicate that it is the prototype for the car class. This will be our car prototype - we won't use the sprite at all other than to create new car objects. Since this particular sprite is never used in the animation, we click the "hide" button so that it's not shown on the stage.

Let's explore some BYOB operations that work on objects. In the "Sensing" category of the blocks palette, there is a reporter block at the bottom that is named "attribute". If you drag that out and click the drop down menu, you'll see this long list:




Those are all the "standard attributes" that are part of any sprite! You can see everything from the name of the sprite, to direction and x and y position coordinates, current costume number, and more. You've only seen some of these so far in this class - clearly there is still a lot to learn about how sprites work in BYOB!

Next, we create the attributes that are specific to the car prototype: Attributes are just variables, set to be "For this sprite only" as shown in this variable definition window:



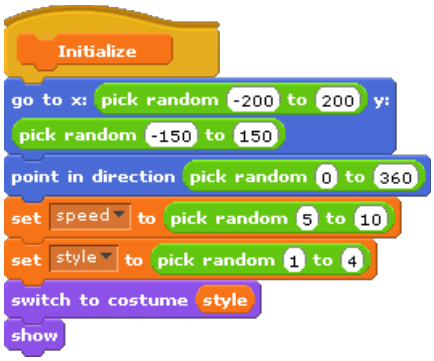

We define both “speed” and “style” in this way, and these variables show up in the blocks palette as follows:


Note that they are “under the line” since they are sprite local and not global variables.

Creating a new car object then involves two steps:

- First, the car prototype sprite runs a script that clones itself - this creates a new object that is a copy of the prototype, with its own copies of each attribute and sprite-specific scripts. For example, the prototype has a “speed” attribute, so the newly-created clone has its own “speed” attribute. The clone’s speed is initialized from the prototype’s speed, but once it is set up it is then independent: changing the prototype’s speed will not affect the clone’s speed and vice-versa. Understanding that each object within a class has its own independent set of variables (defining its state) is one of the most important concepts to understand when working with objects.
- Second, we need to initialize the newly-created car object. When we create a new object we generally don’t want all attributes to be exactly the same as its prototype, so we need to initialize them to make this object unique. This is the job of the constructor, the special method that was mentioned above. One very important thing to understand about the constructor is that we must run the constructor from the newly-created object. If you just called the constructor from the prototype and set attributes for “speed” or “style”, you would be setting the prototype’s attributes, not the new object’s attributes!

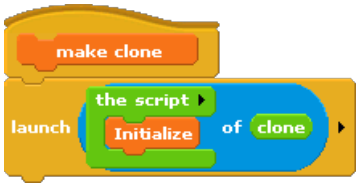
Let’s build the constructor first. For this application, we want to set up the car object’s attributes as follows: the constructor picks random values for the location coordinates, sets a random direction, picks a random speed between 5 and 10, picks a random style which we use to set the costume, and then “shows” the new sprite. Here’s what this block definition looks like - just like the attributes, this block should be defined “*For this sprite only*”:



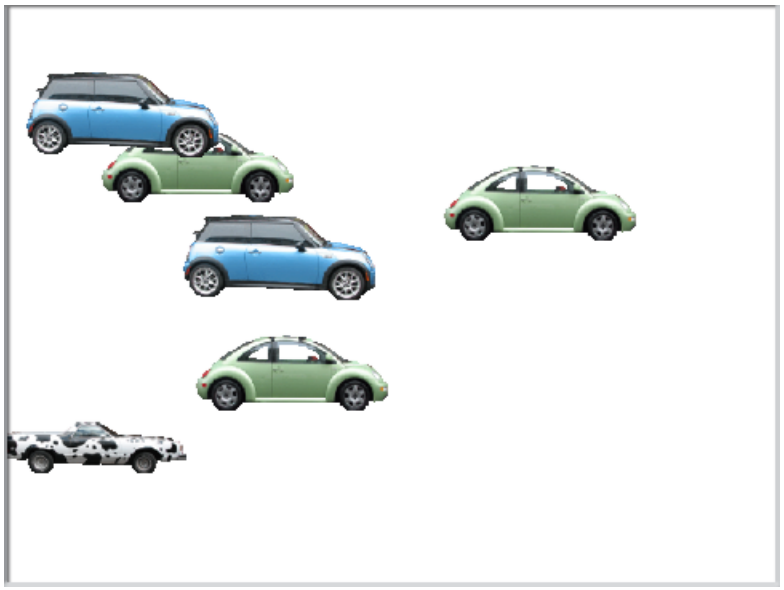

So now we have the constructor, which should be run in any new car object that is created, how do we actually create the new object and execute the constructor? At the bottom of the "Operators" category in the blocks palette is a very simple looking reporter block that's simply named "clone". Despite how plain this block looks, it is very powerful! It creates an complete copy of the current sprite, and then reports the object that was created so that it can be used in further operations. The original object/sprite is called the **parent object**, and the newly created object is called the **child object**. What do we want to do with the new object? At this point, all we want to do is run its constructor, but remember that we need to run the constructor *from the new sprite*, while we are currently executing in the context of the prototype sprite. This is a little tricky, but we'll go through the reasoning, and then show you the code. To refer to either an attribute or a script that is in another sprite, we use the block that looks like this (it's in the sensing category):


The first parameter gives what we want to access, and the second one specifies the object that we want to use. In this case we want to run the "Initialize" script (that's the constructor) in an object that we create using "clone" - we use the special "the script" block in "Operators" to specify the script we want to run, so we can create a clone and refer to the Initialize script in that clone as follows:

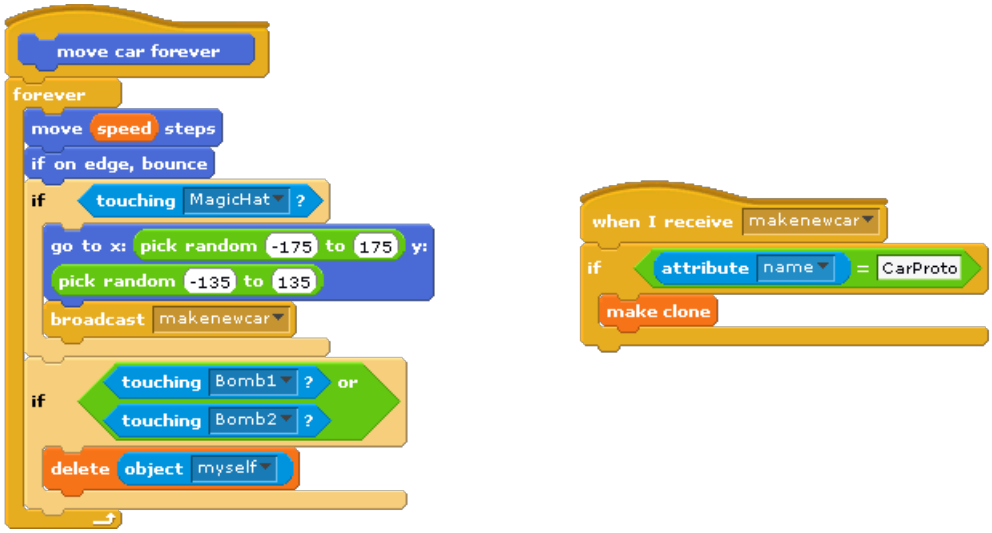

Now that we can refer to the script, we need to say what we want to do with this script, and the options are "launch" or "run", which are both in the "Control" category. The difference between these two options is whether we run the script and wait for it to complete (this is "run"), or whether we start the script in its own separate thread and let it run on its own, without waiting for it to finish (this is "launch"). This is a little messy, so let's put all these pieces together into a custom "make clone" block in the CarProto sprite that looks like this:




Once this is defined, we can drag the "make clone" block out in the "CarProto" sprite, and click it a few times - each click creates a new car object that is placed randomly on the stage, resulting in a stage that looks like this:



Finally, we need a car to start moving around when it is created. We first think about how movement works, and define a block (“For this sprite only”) named “move car forever” that does the correct movement based on the “speed” attribute of this sprite. It then checks for collisions - if it collides with the hat then it needs to create a new car, and if it collides with either of the bombs then it needs to delete itself. The first one of these is a little subtle - we want to make sure that it is the *prototype* object that clones itself (not the current sprite), and we need to ensure that only one car is created at a time. This is a little more intricate and complex than we need to go into for a lab that focuses on basic object-oriented design, but the solution is to broadcast a signal back to the CarProto sprite to tell it to create a new car. The resulting solution, showing both the “move car forever” definition and the event handler for CarProto, is shown below:

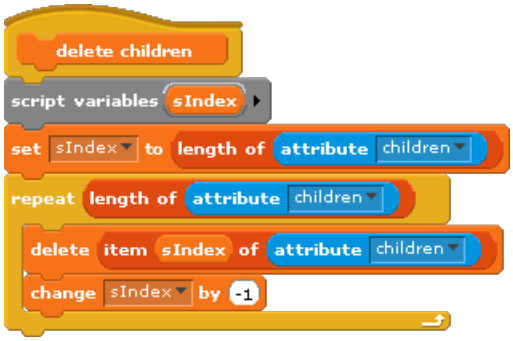



That is almost all we need! We need to launch the “move car forever” block from the car constructor, and then add some other game features such as random movement of the hat and bombs. These are fairly straightforward and don’t show anything new about object-oriented programming, so are not included here. You can download the complete solution from the class web site.

## Cleaning Up After Yourself

If you start writing and experimenting with scripts that create new objects, you will reach a point where you have created several new objects, and you want to get rid of your objects to start your code over. Or worse yet, you run a script that starts creating objects in an out-of-control way, and you end up with 1000 new objects that you don't want. How do you delete these objects? There's no standard answer for this, but I would recommend that you create a global script called "delete children" that you can call from any prototype object to delete every object that was created from that prototype. To understand how to do this, look back at the long list of attributes that are displayed above in the "attribute …" reporter block - see the one that is named "children"? That attribute is actually a list of all children that were created by that prototype object (every time you executed the "clone" block from this prototype).

What we want to do is to walk through the children list and delete each child object that is on that list. There is one subtle problem here: if you delete the first object in the "children" list, it will delete the first element from the list, changing the position of everything that follows. To make this even more confounding, this object does not disappear from the "children" list immediately - sometimes it will happen right away, and sometimes it will take a little time, so you don't actually know if it has been removed from the list when you go to the next iteration of the loop. This is frustrating, but fortunately there is an easy fix: we delete objects starting at the end of the children list and working forward! Think about this: deleting an item in the list can only affect other items that follow it in the list, so by going from the end of the list to the beginning we avoid problems of objects changing positions in the list. If you didn't understand that, take a chance to think it through and try to understand what the issue is. In the end, this is the block definition that we want:




Once this is defined, if you have accidentally (or on purpose!) created 10 car objects using the "make clone" script, if you click on the "delete children" block in the CarProto object you can watch all of the car objects (except the prototype!) in the sprites pane disappear one by one.

This Pre-Lab Reading has given you the basic background information on object-oriented programming, with examples of how this works in BYOB. The In-Lab activities will take this another step farther, looking at constructors that take parameters and other advanced features, and in the end you should have a cool memory game implemented!

## Self-Assessment Questions

Use these questions to test your understanding of the Background reading. Answers can be found at the end of the lab exercise.

1. Why is it important that attributes for an object be created “For this sprite only”? For example, what would happen if, in our animation, the “speed” attribute for a car object was created “For all sprites”?
2. Consider defining a “Fraction” class, where each object is a fraction with an integer numerator and integer denominator. What are some attributes and methods that this class might contain? Note that there’s no single right answer here - see what you can think of!
3. Describe a situation where a sprite is both a parent object and a child object.
4. Think back to the first lab with the “Alonzo and the Dragon” animation. Using the blocks introduced in the pre-lab reading, how could Alonzo make the Dragon disappear directly (“directly” means you can’t have Alonzo broadcast a signal that the Dragon catches and then makes itself disappear).
5. We used the “children” attribute in the “delete children” definition, and the “children” attribute turned out to be a list of sprites created by this sprite. What do you suppose the “parent” attribute is? Is it a list?
6. (Warning: this is actually a somewhat tricky question - you should definitely think it through and try to answer it, but the correct answer might surprise you!) What if we start up BYOB and define the following event-handler for Alonzo:

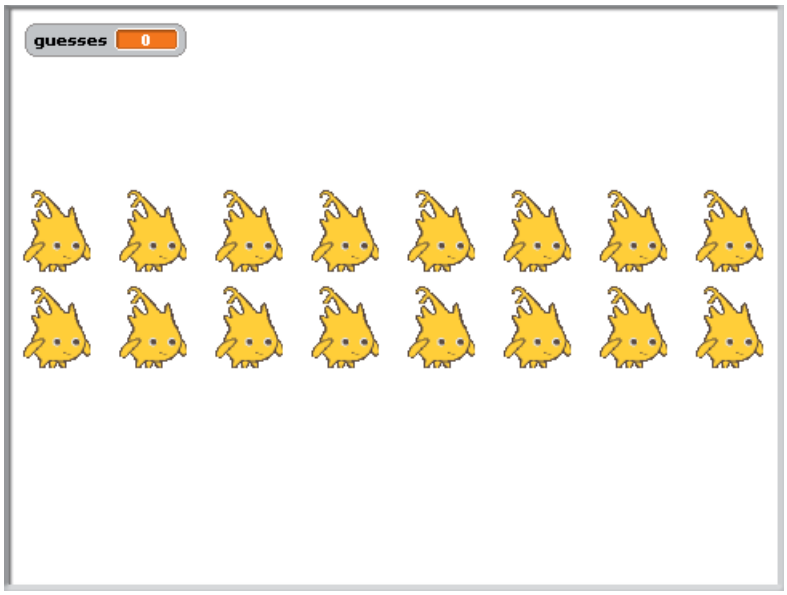
   
   

   Next, we press the space bar 4 times. How many Alonzo sprite are there now?

## Activities (In-Lab Work)

In this lab we are going to create a simple memory game. You’ve probably played this kind of game before - for example, some people play with half a deck of cards, shuffle them up and deal out the cards face down. Players then take turns turning over pairs of cards - if you match (two queens, for instance) you take the pair and go again. In our game, we will have a bunch of Alonzos, but each one has a secret identity! There are two Alonzos that are really dragons, two that are the the little blue character, etc. You will uncover secret identities by clicking on them.

Here’s the basic idea: We’ll start with a single Alonzo that sprite that will be a prototype for our game object. The prototype will have a total of 9 costumes, including the basic Alonzo costume and eight secret identity costumes. The basic outline of our game program is then as follows: Our game uses this prototype to create two rows of Alonzos, each row with eight Alonzos and a random assignment of secret identities. Since each of the new Alonzos is a clone of the parent, each one will have the 9 costumes of the original Alonzo. How do we keep track of the secret identity for each Alonzo? That’s precisely what we were describing above as an attribute, which is a value that can be set and tested or read separately for each clone. We will see exactly how this is done in Activity 1 below. When the Alonzos are laid out on the stage at the beginning of the game, each with a hidden secret identity, we want the stage to look like this:

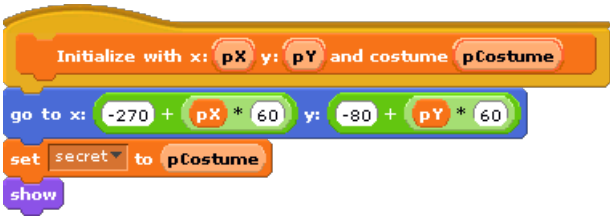



***Warning!!!*** *In this lab you will be creating sprites and deleting sprites dynamically. It is easy (way too easy!) to accidentally delete all your sprites and the scripts that go along with them - this is super frustrating when it happens (I speak from experience!), and you have to start over if you haven't saved your work. Because of this, it is extremely important that you save regularly. Just get in the habit of clicking the save icon every time you modify scripts so that you can recover them if they end up doing something disastrous.*

**Activity 1:** In this activity, you'll practice using cloning to create new objects. To start off, open BYOB with a new project, which should include an Alonzo sprite. Next create an attribute to hold a sprite's secret identity: Go to the variables category and click "Make a variable," Use "secret" for the name, and check "For this sprite only." While we're setting up this first Alonzo, drag out the "set size" block and set the size to 40%, click on it, and you'll have an appropriately-sized Alonzo for this exercise. Finally, you should add 8 "secret identity" costumes to the Alonzo sprite, using the "Import" button in the Costumes pane. You can pick whatever costumes you want, as long as they are roughly the same size as Alonzo. Here are the costumes I used:

Next, we need to create the constructor for secret-agent Alonzo, similar to the way we made a constructor in the pre-lab reading. In the constructor, we need to set the location for this sprite, set the secret identity, and since the parent sprite will be hidden we need to make sure this clone is visible. The only slightly tricky in this is the way we set the position. Unlike the Pre-Lab reading example, we want each new object to be place in a very specific location to form the two-row layout shown above, so we need to pass parameters to the constructor to indicate where the new sprite should be placed. We'll use two parameters to denote a row and column number with row values from 1 to 2 and column values from 1 to 8. To set the actual x and y placement we'll use some formulas: we'll allocate a 60x60 block for each Alonzo, shifted by an

amount so that we can fit two rows of eight that are roughly centered on the stage. Putting all these pieces together we get the following constructor, which you should build:




Next, we'll make the script that actually creates the clone, as we did with the "make clone" block in the Pre-Lab Reading, with the only difference here being the fact that we need to handle parameters for the position and costume. To send parameters to the constructor when we launch it, we can't just put the parameters inside the block indicated with "the script". It is actually the "launch" block that provides the inputs. Notice the arrow at the right edge of the "launch" block - if you click on that, you will be able to provide parameters (you can do this with the "run" block too). The resulting "make clone" block that you should build looks like this:


After you build this block, test it out to see if it works correctly. For example, drag out the "make clone" block, fill in the parameter and execute it, like this:


Click on this, and an Alonzo should appear on the screen, and an additional Alonzo sprite will appear in the sprites pane. Select this new sprite in the sprites pane, and check the "secret" variable in the variables category (you can either check "secret" so that it is a watch variable, or just click on the variable in the blocks palette to make it report its value). Does the secret have the value 2? (It should!) Next, select the original Alonzo sprite in the sprites pane, and try "make clone" again, but with different arguments (say x=2, y=2, and secret=4). You should see another Alonzo, at another position, and if you select that Alonzo in the sprites pane you should see the different secret value (4 in this case).

When you have completed this activity, and have experimented with this enough to understand how things work, save it as "Lab8-Activity1".

**Activity 2:** If you tested the "make clone" block from Activity 1 a few times, you should have several Alonzo sprites now. It would be nice if we had the block described in the Pre-Lab Reading (in the "Cleaning Up After Yourself" section) to get rid of the clones. For this activity, build this block and test it! ***Important:*** *Create this as a global script ("For all sprites"), and when you run it make sure you run it from the prototype Alonzo - that's the only one that should actually have children!*

When you have this script build and tested, save you work as Lab8-Activity2.

**Activity 3:** We want to set up two lines of Alonzo clones, with a random ordering of secret identities in each row. What we want to do is create a list of secret identities (numbers 1 through 8) in random order. This is actually a good use of the "insert" block with the "Any" argument, which puts a value in a random position. To create a randomly ordered list, we simply insert values at "Any" position, and if we use the index iterator pattern to count through values to insert. We will create a reporter block that builds a list and reports it, so that we can save the list in a variable and use it for secret identities. Here's the definition:


We can call this block, and iterate over the randomly-ordered identities, using those when we create a line of Alonzos. Examine the following script and understand what it does:


Notice the two loops - the outer loop iterates twice, once for each row of Alonzo's that we create. Within this first repeat loop, we create a random ordering of secret identities, and then create 8 clones of Alonzo with the secret identity taken from the random permutation (note that we add one, since the first costume is the basic Alonzo costume, not a secret identity). Build this script and test it out. When you've got it working correctly, save this as Lab8-Activity3.

**Activity 4:** Now you're on your own! This activity is entirely extra credit, but it will be good to see how far you can get in completing the memory game. The main task is to handle what happens when the player clicks on an Alonzo clone. We can uncover secret identities with a script that looks like this:


Of course, that's *not* what we want to do. We want to handle clicks in pairs, and hide or remove sprites when there is a match. This can be done with some fairly short scripts, but they're a little tricky. Experiment and see what you can make work!  Save your final version as Lab8-Activity4.

## Submission

In this lab, you should have saved the following files:  Lab8-Activity1, Lab8-Activity2, Lab8-Activity3. If your Lab8-Activity3 file contains all of the blocks created for previous activities, you can just submit that one file. You will also optionally have an extra credit file Lab8-Activity4, which should be turned in seprately.  Turn these in using whatever submission mechanism your school has set up for you.

## Discussion (Post-Lab Follow-up)

### More on Object-Oriented Programming

One of the key features of object-oriented programming, which we didn't address above or in the lab activities, are the notions of sub-classes and inheritance. Classes are often specializations of other classes: for example, a game might have an "Enemy" class, but then there are times when you might want to be more specific and have a class for specific types of enemies, such as an "Alien" class, a "Zombie" class, or a "Dragon" class. All enemies might share certain attributes (like a health value), but there are also some attributes that are specific to a type of enemy - of these choices, only a dragon would have a "wingspan" attribute (OK, I just made an assumption that an alien wouldn't have wings, but seriously, has anyone ever seen an alien with wings?).

Using object-oriented programming, a situation such as this would be handled by defining an "Enemy" class, and then defining the others as what are called **sub-classes**, so for example the "Dragon" class would be a sub-class of the "Enemy" class. Going the other way, we say that the more general class (like "Enemy") is a **super-class** of the more specific class (like "Dragon"). A sub-class **inherits** all of the attributes of its super-class, which means that the sub-class will have attributes for everything defined in the super-class, without having to define them again. If we defined an "Enemy" class that has attributes "health" and "location", and then defined a "Dragon" sub-class where the definition only has a "wingspan" attribute, every dragon would still have its own health and location simply because it is a sub-class of "Enemy".

The terminology that is used in object-oriented programming is to refer to an "**IS-A**" relationship. To see if one class should be a sub-class of another, ask yourself if you can put the phrase "IS-A" between the two class names. For example, we can say "A Dragon IS-A Enemy" (not grammatically correct, but give me a little break here), so the Dragon class should be a sub-class of the Enemy class. This makes sense because if one thing "is" another, then it should have all the characteristics (attributes) and can perform the same actions (method) as the

second thing. To consider another example, if you are creating an information system for a university, you might have classes for “Professor” and “Student”; however, since you can’t put “IS-A” between those two class (in either order) it doesn’t make sense for one to be a sub-class of the other. You could define a “Person” class and then both of these could be sub-classes of “Person” since a “Professor IS-A Person.” We could even define another level of class, such as “Freshman” - then a “Freshman IS-A Student” and a “Student IS-A Person.” This is another nice example of abstraction in computing: we can define a sequence of sub-classes that get more and more detailed and specific, and at each level we can (mostly) ignore the details that are hidden in the super-class.

We didn’t explore this in the lab activities since BYOB does not provide a good mechanism for defining sub-classes - you can “fake it” in BYOB, but it’s ugly and obscures what is really a very clean and elegant notion in more capable object-oriented languages. There is in fact one place where you see inheritance in BYOB: if you define your own class using a sprite, then it automatically includes all of the regular sprite attributes such as direction and position - these are really attributes that are inherited from the super-class “Sprite”. Unfortunately, this is a special case in BYOB, and doesn’t translate into a consistent method for defining sub-classes with inheritance.

## Terminology

The following new words and phrases were used in this lab:

- *attributes*: variables that are associated with a specific object or sprite
- *child object*: an object that is created by cloning a prototype object - in cloning, the original object is known as the “parent object” and the new object is a “child object”
- *class*: a general class of object – individual objects are instances of some class, which defines the common characteristics of the class (weapons, monsters, etc.)
- *constructor*: a script that is run to initialize various attributes in a new object
- *inherit*: the process by which a sub-class includes the attributes and methods of its super-class
- *IS-A*: the relationship between a sub-class and its super-class - for example, a “Dagger” IS-A “Weapon”
- *methods*: scripts that are specific to a particular object or sprite, performing actions on or for that object
- *object-oriented programming*: a style of program design that is oriented around defining programmatic objects in terms of the attributes and methods (note: object-oriented programming has many other aspects that are beyond the scope of this lab!)
- *parent object*: an object that creates another object by cloning - in cloning, the original object is known as the “parent object” and the new object is a “child object”
- *prototype*: an object that is defined so as to serve as a template or model of objects from a particular class
- *prototype-based programming*: a style of object-oriented programming in which new objects are created by cloning a prototype object
- *sub-class*: A class that is a specialization of another class - for example, a “Dagger” is a special type of “Weapon”, so a “Dagger” class might be a sub-class of a “Weapon” class

- *super-class*: The more general class of two classes related by an IS-A relationship - in the example given in the sub-class definition, “Weapon” is a super-class of “Dagger”

## Answers to Pre-Lab Self-Assessment Questions

1. When you set a variable as “For this sprite only”, every time you clone that sprite it clones the variable, creating a new “For this sprite only” variable for the new sprite/object. Therefore, if you create 5 new car objects, you have also created 5 new “speed” variables, one for each new car. If you didn’t do this, then cars could not keep track of their own speed separately from the other cars. If you had defined “speed” as a “For all sprites” variable, every new car object would share the *same* global speed variable, so changing the speed for one car would change the speed for *all* cars!

2. The two obvious attributes are the numerator and denominator of the fraction. There are lots of choices for methods: you could have a method that reduces a fraction to lowest terms, stamps out the fraction, multiplies a fraction by an integer, or adds two fractions together. Basically anything you might want to do with a fraction defines an action on that fraction, and any of these actions could be methods in the Fraction class.

3. Consider the following situation: you start off with a sprite named Sprite1, and this sprite executes “clone” to create a new sprite named “Sprite2”. Then this new sprite executes “clone” to create a new sprite named “Sprite3”. In this situation, Sprite2 is a child of Sprite1 (since it was created by Sprite1), and Sprite2 is also the parent of Sprite3. In other words, Sprite2 is both a parent sprite and a child sprite.

4. The way we launched the constructor of the new sprite from the parent sprite works for any two sprites - they don’t have to be parent and child! The following script, run by Alonzo, would make the Dragon disappear:


5. The parent attribute is actually just a single sprite, not a list. A sprite can have many children, because it can call “clone” many times, and so it needs to use a list for children. However, a sprite can have only one parent (the sprite that created it), so no list is needed and just a single sprite can be stored.

6. There would be 16 Alonzo sprites in the end. The key observation here is that when a sprite is cloned, everything is cloned, including scripts. Therefore, if the original Alonzo is named “Sprite1”, then when the spacebar is pressed it executes “clone” to create a new Alonzo named “Sprite2”. The new sprite has an event-handler for spacebar presses, just like the original Alonzo, so when the spacebar is pressed again both Sprite1 and Sprite2 handle this event and clone themselves, creating two new objects. Therefore, after 2 spacebar presses we have 4 Alonzo sprites. On the third spacebar press, all 4 Alonzos clone themselves, resulting in 8 Alonzos. And finally, on the 4th spacebar press these 8 Alonzos clone themselves, resulting in a total of 16 Alonzos. This is why you generally only want the prototype object to clone itself, not any of the child objects, which is why we included the “if” block in the “makenewcar” event handler in our example.