

# Maine LearnToMod Project

## Example Curriculum

# Table of Contents

<b>How can I teach coding if I don't know how to code?!?</b>	<b>4</b>
<b>The Curriculum</b>	<b>5</b>
<b>LearnToMod Day 1: Suggested Introductory Lesson.</b>	<b>7</b>
Getting Settled	7
Introducing Mods	7
Why coding can be challenging	8
Tour LearnToMod	8
Wrapping Up	8
<b>Functions: Chapter 1, Lesson 1</b>	<b>9</b>
Goal	9
Definition	9
Code	9
Final Mod	12
Common Questions and Errors	12
<b>Events: Chapter 1, Lesson 2</b>	<b>14</b>
Prerequisite: Functions Lesson	14
Goal	14
Definition	14
Code	15
Final Mod	16
Common Questions and Errors	16
<b>Variables: Chapter 1, Lesson 3</b>	<b>18</b>
Prerequisite: Events Lesson	18
Goals	18
Definition	18
Code	18
Final Mod	21

Common Questions and Errors	21
<b>Drones: Chapter 1, Lesson 4</b>	<b>22</b>
Prerequisite: Functions	22
Goal	22
Definition	22
Code	22
Final Mod	24
Common Questions and Errors	24
<b>Locations: Chapter 1, Lesson 5</b>	<b>26</b>
Prerequisite: Drones	26
Goal	26
Definition	26
Code	26
Final Mod	27
Common Questions and Errors	28
<b>Loops: Chapter 2, Lesson 1</b>	<b>29</b>
Prerequisites: Functions, Drones	29
Goal	29
Definition	29
Code	29
Final Mod	30
Common Questions and Errors	31
<b>Logical Statements: Chapter 2, Lesson 2</b>	<b>32</b>
Goal	32
Definition	32
Code	32
Final Mod	36
Common Questions and Errors	37
<b>Inner Loops: Chapter 2, Lesson 3</b>	<b>38</b>

Prereq: Loops Lesson	38
Goal	38
Code	38
Final Mod	39
Common Questions and Errors	40
<b>Functions with Parameters: Chapter 3, Lesson 1</b>	<b>41</b>
Goal	41
Definition	41
Code	41
Final Mod	47
Common Questions and Errors	48
<b>Events with Parameters: Chapter 3, Lesson 2</b>	<b>49</b>
Goal	49
Definition	49
Code	49
Final Mod	53
Common Questions and Errors	54
<b>Events with Player's Location: Chapter 3, Lesson 3</b>	<b>56</b>
Goal	56
Definition	56
Code	56
Final Mod	64
Common Questions and Errors	64

## ***How can I teach coding if I don't know how to code?!?***

Technology has many conveniences, however the breakneck pace of it's development has created a uniquely difficult problem for today's teachers. Coding is unquestioningly an important skill for modern students, and will only become more important in the future, but many teachers (including the author of this curriculum) received little to no formal education in coding.

While programming may be daunting at first, we implore you to always remember the first and most important rule of coding, concisely summarized here by Science Fiction author Douglas Adams:

*"Don't Panic."*

We, here at LearnToMod, have been hard at work creating resources, lessons and guides for teachers and students alike. With these resources will have no trouble providing your students with quality coding lessons lessons, regardless of your prior experience.

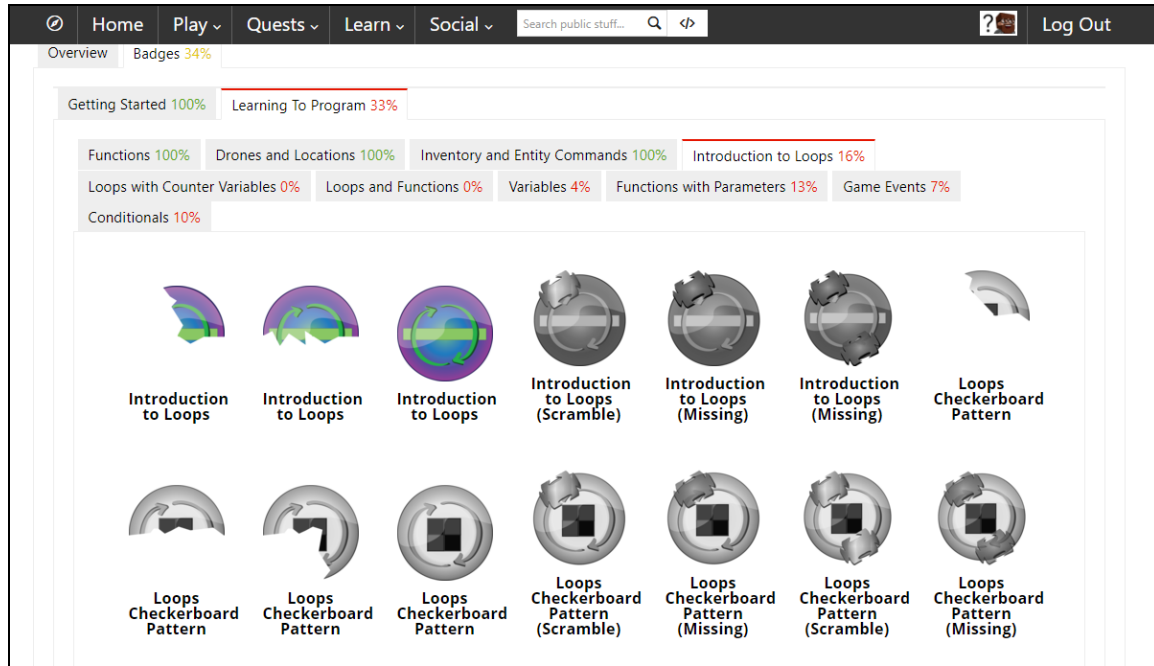
We must emphasize that this curriculum is by no means the only way to use LearnToMod in a classroom. The most important educational tool in any classroom is the teacher, so we encourage you to tweak, change, or throw away as much of the curriculum as you like to provide the best possible experience for your students.

Let's get started!

# The Curriculum

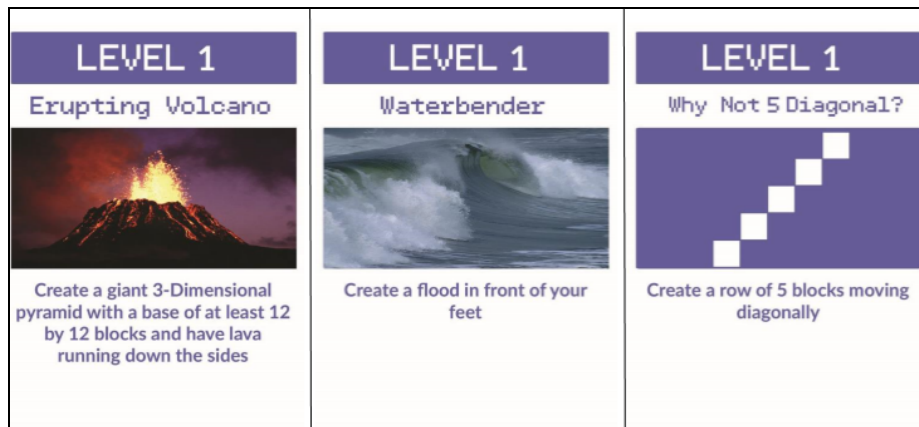
After the introductory lesson, each lesson in the Example LearnToMod curriculum is divided into two halves -- “Level Up” time and the “Mod of the Day.”

During Level Up time, students will utilize LearnToMod’s built-in instructional tools and badge-based lessons to explore their own interests, at their own pace. In this period, the role of the instructor is to act as a facilitator for the students, guiding them through difficulties while providing advice and encouragement.



A sample of LearnToMod’s available badges, found in “Skills and Drills.”.

If students are feeling confident in their coding abilities, they may choose to attempt a “Challenge Card.” Challenge Cards contain ideas for mods and task students with discovering how to code them.



A sample of beginner challenge cards.

Students will track their badges and challenge cards on a LearnToMod “Journey Sheet.” After completing a set number of badges and challenge cards, the student will test their knowledge by designing and creating their own personalized mod. Upon the successful creation of this mod, the student will “level up” and get a fresh Journey Sheet. Level up’s provides the teacher with an easy way to track students’ progress, provide the students with a sense of progression and accomplishment, and encourage students to challenge themselves with a variety of activities.

During the “Mod of The Day”, the instructor will lead the class in the creation of a new mod. Each mod is outlined in this curriculum and gradually introduces students to essential concepts in computer science, such as variables, loops, and logic. Students are encouraged to use any remaining class time to play with the code and add their own creative elements to it.

As the curriculum progresses, the concepts covered in the Mod of the Day increase in complexity. Some of the most complex mods may require multiple class sessions to complete. The instructor is encouraged to spend as much, or as little, time as needed doing activities in each lesson. Instructors are also encouraged to alter the order of the lessons to best suit their students’ interests. Keep in mind however, that some activities require certain concepts to be covered prior to running the activity. The prerequisites for each lesson will be listed under the lesson title.

Here is an example of how program held for eight 1-hour lessons might be run:

	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8
30 min	First Day Intro	Level up time	Level up time	Level up time	Level up time	Level up time	Level up time	Level up time
30 min		CH 1 Activity	CH1 Activity	CH 1 Activity	CH 2 Activity	CH 2 Activity	CH 2 Activity	Share Projects

# LearnToMod Day 1: Suggested Introductory Lesson

While many of the lessons in this curriculum can be done in any order, we recommend spending your first session doing this introductory lesson, or a variation of it. It is based on a 1-hour class.

## Getting Settled

5 minutes:

- Help the students get seated
- Introduce instructors and mentors
- Take attendance

## Introducing Mods

5 minutes:

Good afternoon everyone! My name is \_\_\_\_\_. How are you all doing today?

*Get student responses.*

I have a question for you all. Who here likes Minecraft? Does anybody here play Minecraft?

*Get student responses.*

So, those of you who have played Minecraft, I'm sure you've done things dug a mine, used the materials to build a nice house, and then had that house get blown up by creepers right? Has anybody here ever played around with Minecraft mods before?

*Get student responses.*

What is a f?

*Get student responses.*

“Mod” is short for modification, or change. Basically, mods are used when people change the way the game works to make cool stuff happen. Has anybody seen any cools mods?

*Get student responses.*

Those are some awesome mods! In this class, we are going to learn how to make our very own mods! To make mods, we first need to understand how to talk to the computer and tell it what we want it to do. This can be kind of tricky, so to make it easier, computer scientists have made special languages to talk to computers. We call this “coding.”





## Why coding can be challenging

**20 minutes:**

The hardest part about coding is understanding that what may seem obvious to us is not always obvious to a computer. Let me show you:

Lead the class in an activity demonstrating the difficulties of explaining the task to something that has no knowledge of the task. Explain to the students that you are a computer and the students must program you to perform a specific task. Pick a task the students are familiar with (making a sandwich, putting on shoes, etc.), and have the students give you step-by-step instructions to complete the task. In each step of their instructions, you must deliberately misinterpret everything they recommend. For example, if the students say, “Put the shoe on”, place the shoe on a shelf or table. If they correct themselves and say, “Put the shoe on your foot” place the shoe on top of your foot, rather than put your foot inside of the shoe.)

All right! So what went wrong?

*Get student responses.*

Computers don’t understand the world like we do, so we need to be very specific and very clear with the instructions we give them. Code helps us to do that. We are going to be using a special coding language that has been made specifically to make mods for Minecraft. Who wants to get started?

## Tour LearnToMod

**20 minutes:**

- If possible, show students the LearnToMod website on a projector and briefly walk through how to sign in. We recommend passing out student name tents, and Take Home Sheets, so students can easily keep track of their login information.
- Once signed in, you can take a moment to give the students a tour of the LearnToMod website, or you can play our video tour. The video tour can be found at [https://youtu.be/3RIZ9T\\_u1Mo](https://youtu.be/3RIZ9T_u1Mo).
- Students should spend any remaining class time completing lessons in the “Skills and Drills” badge collection, found on the LearnToMod site under menu item: Learn > Badges.
- It is a good habit for students to test their mods after each step, to make sure they are on track. While this may not be feasible for every lesson, students should definitely test their final mods.

## Wrapping Up

**10 minutes until class end**

- Give 5 minute warning!
- Remind them to take their credentials sheets home.



# Functions: Chapter 1, Lesson 1

## Goal

- Make a mod with multiple functions that
  - Sends the player a message
  - Gives the player a sword
- Work through the Functions badges

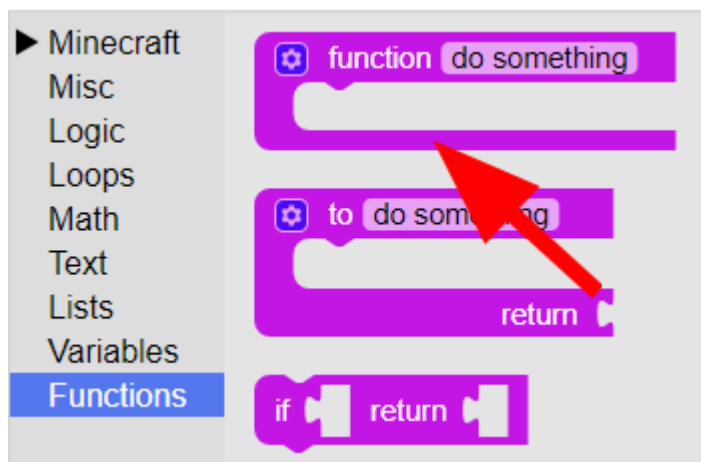
## Definition

The *function* is the most basic building block of a program. Functions act like instruction books for us to put all of our code inside of. When a mod first starts, the computer will look for a function called *main* (in all lowercase). As the first step in any mod, if a *main function* is not present, the mod will not proceed.

## Code

In each mod, we always need ONE and ONLY ONE main function. As the starting location for our program, without a main function containing instructions, the computer will be unable to navigate our code and we will get errors when trying to run our mod. Mods are not limited to a single function; quite the opposite! Multiple functions are a great way to organize code.

To *create a new function*, we need to *define* it. To the left side of the LearnToMod *code editor* (the area of LearnToMod where users can create code), click **Functions** and click the purple block with a mouth, **function 'do something'**; this is called a *function definition*.



Drag this block into your workspace. To rename it, click on **‘do something’** and type `main`. When we run our program, our computer will start at the top of the `main` function and do every command in the order they appear.

Let's put a command in this function. Click and dropdown the **Minecraft** menu and click **Players**. Find and click the red block, **Send message \_\_\_ to \_\_\_**.



Drag this block into the “mouth” of the `main` function. Now click **Text** and click on the aqua “\_” (an empty *text block*) and the red **me** block from **Players**. You will find that these blocks fit right into the **Send message...** block. Click on and type something into the “\_” text block. (In the example, the programmer typed `HELLO WORLD`. Your mod should now look something like this:



Demonstrate this mod in action or, if students are logged onto their accounts, have them test their code.

Now, let's create a function to give the player weapons. We can call it whatever we want, but I suggest you give it a meaningful name. For instance, if you have a function that gives your player weapons, you can give it a name like `give_weapons` instead of `cool_function`. In large programs, you may have dozens of functions to keep track of, so proper (logical) naming helps a lot to organize them.

**NOTE:** When naming *functions*, the *first* letter of every function must be lowercase.

**NOTE:** In this example, the “naming convention” of functions uses lowercase words connected by the underscore character, such as `give_weapons`. However, in some mods, programmers use slightly different conventions, such as `giveWeapons` where all words are connected and all words after the first word are capitalized. As mentioned above, the first character of a function must be lowercase.

```

function main
  Send message "HELLO WORLD!" to me

function give_weapon

```

Let's make this new function give the player a diamond sword. Under the **Item** tab, click on the command **Give \_\_\_ of item type \_\_\_ to player \_\_\_**. Fill in the blanks with a **0 number block** from the **Math** tab, a **DIAMOND\_SWORD** block from **Materials**, section **[D-G]**, and a **me** block from the **Players** tab. Plug these blocks into the **Give item...** block and put the whole thing into our new function. Change the number block from a **0** to a **1**.

```

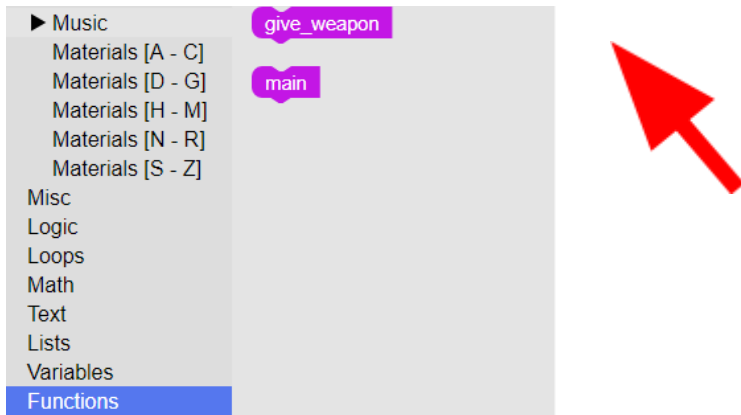
function main
  Send message "HELLO WORLD!" to me

function give_weapon
  Give 1 of item type DIAMOND_SWORD to player me

```

Now we have two functions! We aren't quite done yet though. Our `main` function is the only function that automatically runs when we activate our program. If we want to run other functions, we need to tell the computer to do so. We can do this with a *function call*. Let's add one to our main function.

Under **Functions**, you should see a block that shares a name with your function. Get this block and place it inside of your `main` function.



## Final Mod

You should now have a mod that looks something like this:



This mod will first send a message and then call our “give\_weapon” function, which will give our character a diamond sword.

Test your mod by clicking the “MOD” button at the top of the screen and running your mod in Minecraft.

## Common Questions and Errors

- Students may recognize that the **Give ... item ...** command could simply be placed in the `main` function, eliminating the need for a function call altogether. This is 100% correct, and these students should be praised for finding a more efficient solution. Reiterate to students that function calls are very helpful if the same code must be run multiple times, or if the code must only be run when specific criteria are met. Functions can also be a useful tool for organizing large chunks of related code.

- If code is not running properly, students should check that they have properly created a main function and that they are calling the second function from it. Check that the number **0** is changed to **1**. Computers are very picky about syntax!!
- A good way to troubleshoot code is by testing with parts of the code that seems to work and gradually adding other parts into the action. By right-clicking on a block in the editor, it may be disabled (click **Disable block**), which will grey out the block and any of its contents. Remember to **Mod** before trying your code again in Minecraft.

# Events: Chapter 1, Lesson 2

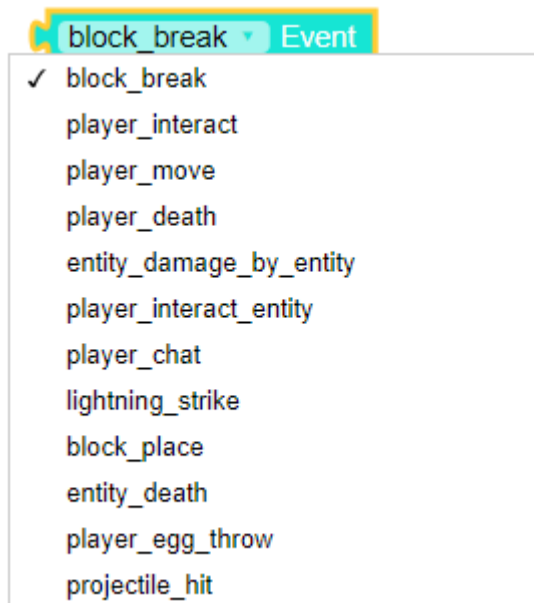
## Prerequisite: Functions Lesson

### Goal

- Make a mod that strikes lightning and spawns an entity, every time the player breaks a block
- Work through the Events badges

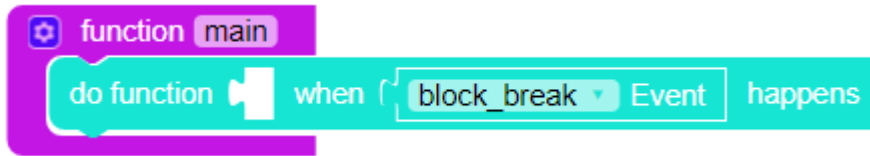
### Definition

An *event* is any occurrence of a specific set of circumstances (it can also be called a *trigger*). When an event occurs, the game will run specific code in response. In Minecraft, there are probably a couple of hundred different events happening at any time and we might not be aware of them. Some of these events are basic or frequent: **block\_break**; **player\_chat**; **entity\_death**. Others are specific, such **creeper\_power**; **entity\_combust\_by\_block**; or **player\_bucket\_fill**. The LearnToMod software already has some of these events *hard coded* (programmed) into the **Event** block dropdown menu:

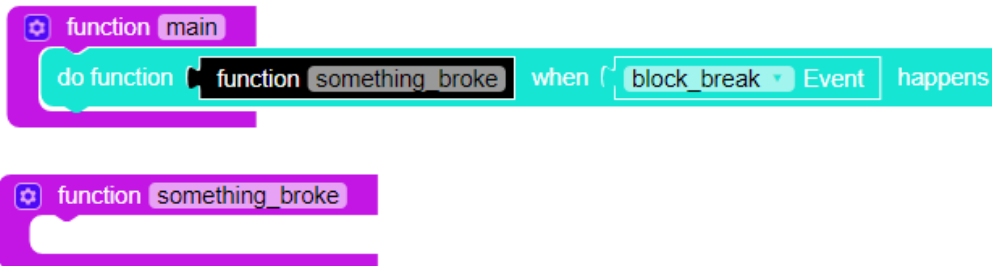


## Code

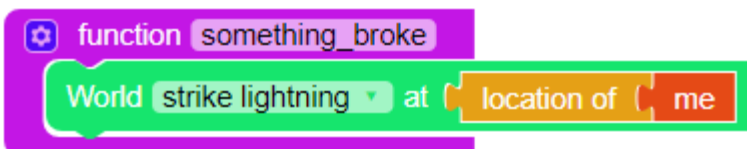
Let's start by using a **block\_break** block and a **do function \_\_ when \_\_ happens** from **Events**. Plug the **block\_break Event** block into the second slot of the **Do function...** block. Now create a `main` function and put the new event code inside of it.



The **do function ...** block says that when a specified event is triggered, it can run a specific function. Create a new function called `something_broke` or a similar name. Under **Misc**, grab a black **function 'function name'** block. This black block should fit into the other space in our **do function** event block. The black block is a *function reference* and it refers (directs) the code to run that function whenever the defined event happens. Change the text in the black function block from **'function name'** to the name of the new function; in the example it is `something_broke`.



Now we have an event! As long as our mod is active, any time the player breaks a block, our mod will run our `something_broke` function! Now just need to put some code inside of it. We want this function to strike lightning and summon a zombie. To strike lightning, click on **World strike lightning at \_\_** from the **World** tab. Plug it into our new function. The blank space in this command must be filled with a location. Get a **me** block from **Players**. The **me** block refers to the player, however the player has lots of pieces of information -- values, characteristics, etc. -- associated with them. The computer doesn't know what information we want to use. We need to specifically request the *location of the player*. Under the **Entities** tab, get a **location of** block. Plug this into the space in the **World strike lightning ...** block. Then, plug the **me** block into the slot left by the **location of** block.



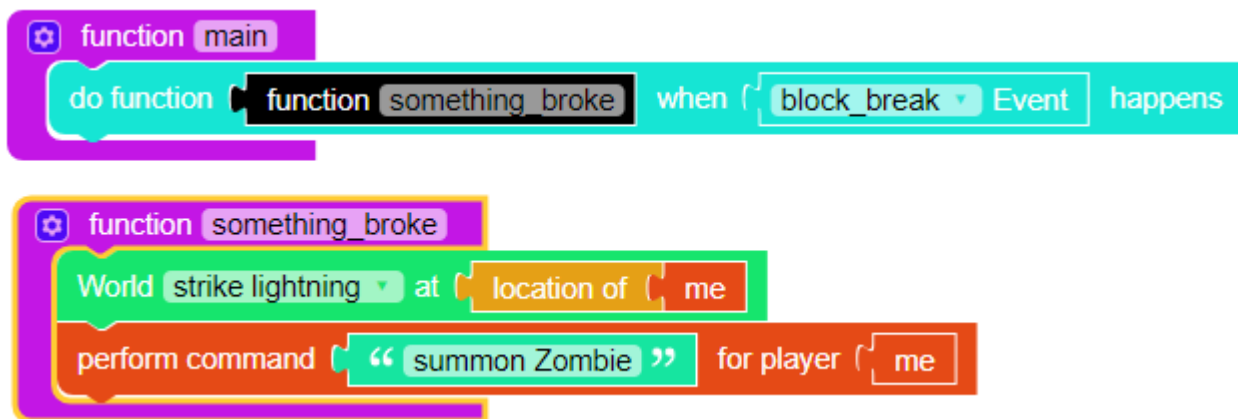


Now let's *summon* (or *spawn*) a zombie. To do this, we will use the **Perform command \_\_\_ for player \_\_\_** block, from the **Players** tab. Fill in the two spaces in this block with an **” Text** block and a **me** block, respectively. Minecraft has numerous commands that can be executed in-game. The **Perform command ...** block lets us use some of those. Some students may have used some of these commands, either through Minecraft's in-game *Command Line*, or the *Minecraft Command Blocks*. Tell these students that any commands they know will also work inside of LearnToMod. In this case, we want to summon a zombie, so type `summon Zombie` into the text block.

**NOTE:** Commands are case sensitive. Minecraft commands (in LearnToMod) use a specific naming convention, including lowercase first word, spaces between each word, and an uppercase letter to begin second and subsequent words.

## Final Mod

Your code should look something like this:



```
function main
do function function something_broke when block_break Event happens

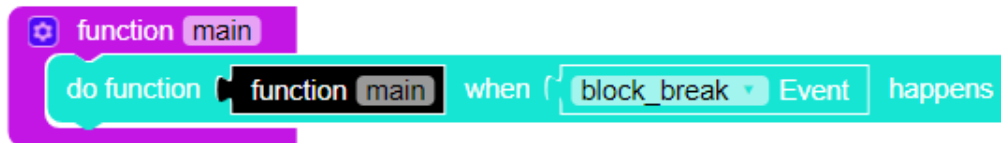
function something_broke
World strike lightning at location of me
perform command "summon Zombie" for player me
```

In this code, the aqua **do function ...** block is telling the computer to run the function `something_broke` whenever a block is broken. According to this code, when we run the mod, the computer will write a note in its memory that says, "Whenever a block breaks, I will run the `something_broke` function." The `something_broke` function will cause the program to strike lightning and spawn a zombie.

Test your mod by clicking the "MOD" button at the top of the screen and running your mod in Minecraft.

## Common Questions and Errors

Some students may discover they can create an event in which a function runs itself! While this practice *may* have uses in some cases, generally, it is highly discouraged.



```
function main
do function function main when (block_break Event) happens
```

If improperly designed, a function that runs itself can lead to an error called a *memory leak*. In short, when the event occurs, the function duplicates itself to create two identical functions. When the event occurs again, these two copies duplicate themselves to create four identical functions. *This duplication continues exponentially, at every instance of the event, slowly using up all the available memory of the computer.* If the leak continues, all the working memory of the computer will eventually be used up, resulting in the server crashing. If this does occur, simply restart the student’s server.

It is also very important that the “summon Zombie” command be formatted exactly as it is written above. Code with errors in spelling, capitalization, or punctuation will not work properly. Explain to students that computers don’t have the same reasoning that we do and cannot infer what we are telling it. While a mistake in punctuation may be small to us, it’s big to a computer.

# Variables: Chapter 1, Lesson 3

## Prerequisite: Events Lesson

### Goals

- Create a counter to count the number of blocks the player has broken
- Work through the Variables badges

### Definition

In programming, *a variable is a value that can change depending on conditions or information passed to the program*. A variable can represent something as simple as a single number, or as complex as an entire function. Each variable has two *attributes*: its *name* and its *content*. The programmer can choose to use either or both of these, depending on what the mod needs.

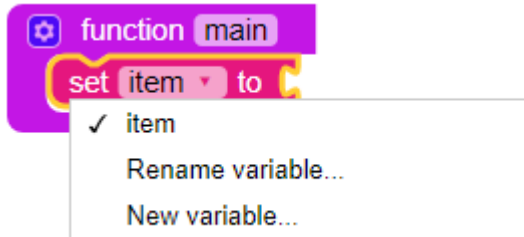
We can have as many variables as we want. Variables come in all kinds of different types, to hold different contents. For example, variables can represent numbers, “drones”, “booleans” (true or false), “strings” (text), lists, or any other value you want to give it: Here are some variables that have been *defined* in this LearnToMod code:

```
function main
  set number to 3
  set drone1 to new Drone
  set isFull to true
  set name to "Bill"
  set inventory to (create list with (DIAMOND_HELMET
  (DIAMOND_CHESTPLATE
  (DIAMOND_BOOTS
```

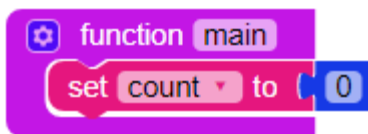
### Code

When we are programming, using a variable requires that we first *declare* it. When we *declare a variable*, we give it a name and tell the computer what it represents (the type of content). *While variables can be defined anywhere in your code, a variable cannot be used until the computer reads the definition*. As such, it is good practice to put variables at the very beginning of your code, so they are the first thing the computer reads.

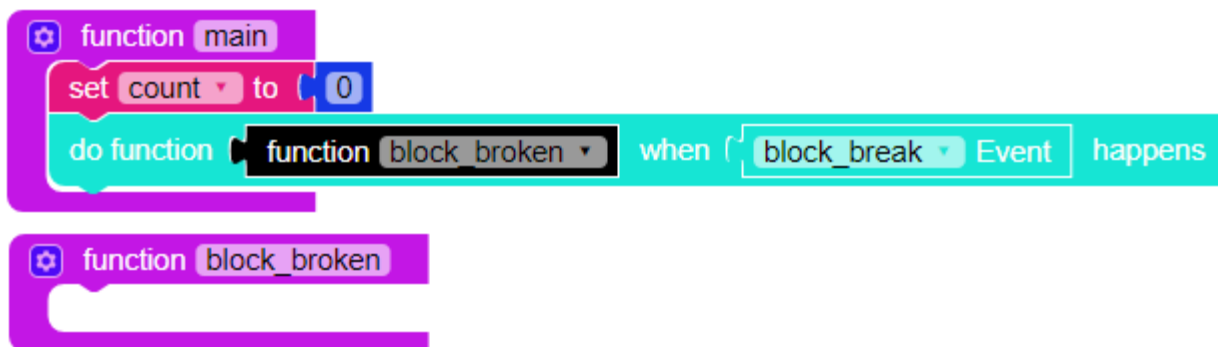
You can create variables by selecting **Variables** in the menu. Choose the **set 'item' to** block. To change a variable's name, click the dropdown arrow next to **'item'**, click **New variable...**, and type in your variable's name. Remember to make the name meaningful. This makes it easier for you to keep track of them and easier for other people to read your code.



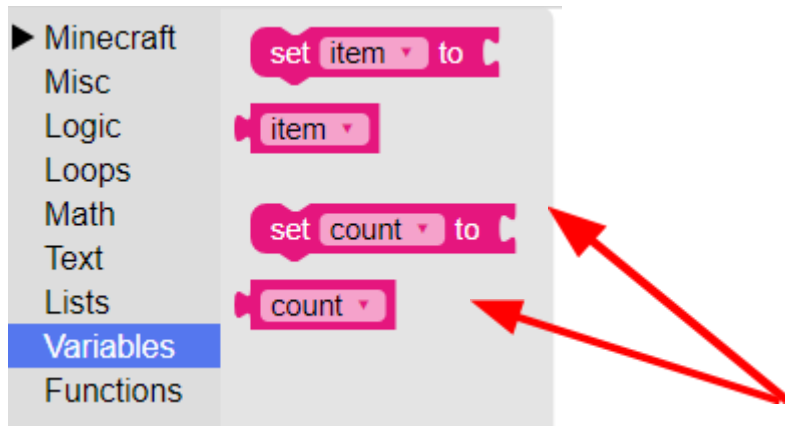
This only takes care of the first *attribute*, the name. Now, we have to tell the computer what the variable is going to contain (content). Let's make a program that will count the number of blocks we break. If we want to count something, we'll need a blue number block, **0**. Go ahead and get one from the **Math** tab.



Next, we should create a **block\_break** event and another function that will run when the event occurs. You will find needed blocks in **Functions**, **Events**, and **Misc**.

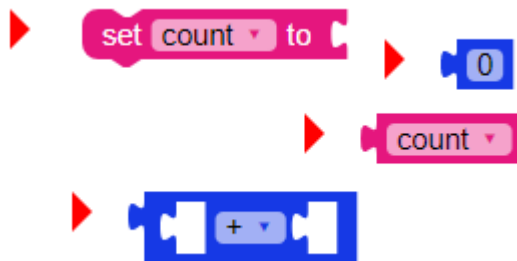


Every time we break a block, we want the computer to add one to our count and send us a message with the count. If you open up the **Variables** tab, you'll see that we now have *two* different blocks that represent our variable attributes.

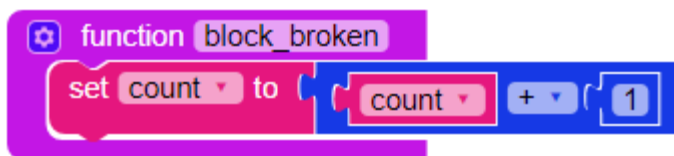


These blocks both refer to our variable, but they do slightly different things. The first *sets our variable to a new value*. The second will *give us (gets) the value that is currently stored inside of the variable*.

To add to our count, we'll need to use *both* of these `count` blocks, as well as a **0** number block, and a `++` block. Both of these blocks can be found in the **Math** tab.



Every time we break a block, we want to set our variable, `count`, to the previous value + 1. To accomplish this, arrange your blocks as follows:



Be sure to change the number block from **0** to **1**. If we don't change this value, we will add 0 to our variable every time the function is run and the variable will never increase!

To see the value (content) of the variable, we'll send the player a message. Get a **Send message...** block from **Players**. Put a copy of our variable in the first slot, and a **me** block in the second slot (**Variables** and **Players**).

## Final Mod

Our final mod contains an event, which when triggered by a player breaking a block, will add 1 to our count and send us a message containing the value of our count. Your code should look something like this:

```
function main
  set count to 0
  do function function block_broken when block_break Event happens

function block_broken
  set count to count + 1
  Send message count to me
```

If students are motivated, challenge them to add code that will subtract from their counter, if they place a block.

Test your mod by clicking the “MOD” button at the top of the screen and running your mod in Minecraft.

## Common Questions and Errors

- Understanding the function reference block; why not just call function directly? When to use it?
- Syntax is critical to programming. If students are getting errors when they test their mods, have them check each letter’s case and all punctuation.

# Drones: Chapter 1, Lesson 4

## Prerequisite: Functions

### Goal

- Create a mod that will place a block for us
- Work through the Drones badges

### Definition

A *drone* is like an invisible robot that helps you, in Minecraft. This drone accepts step-by-step instructions, which tell it how to move. They are basically location-based, making them very useful when building something, or when spawning mobs (creatures), in specific places.

You can program a drone to move in six directions (up, down, forward, backwards, left, or right) by any amount of distance. The metric for distance in Minecraft is a block, therefore, if you give a drone a command, such as...



Move Drone `d` in direction `left` distance `6`

...the drone will move 6 blocks to the left.

### Code

Before using a drone in our mod, we have to declare it within a variable. To do this, we go to the **Variables** menu and get the generic variable declaration block, **set 'item' to**. Select **New variable...**



A small window will open in our browser and it will ask us to type in a name. The default name for the drone is simply `d`, but you can name it whatever you want. Keep in mind that if you name your drone something other than `d`, you will need to change the default name in the blocks we use to command

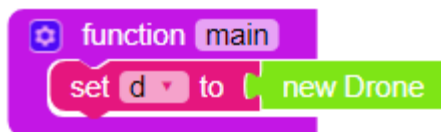
the drone as well. Once we select a name, we must get a **new Drone** block from the **Drone** menu (under Minecraft menu) and attach it inside our variable:



```
set item to new Drone
```

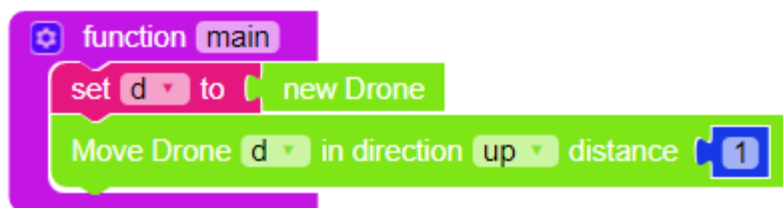
This basically tells the program that whenever we refer to `d`, we are talking about our drone. Explain to students that in this case, *our variable is acting as a name tag*. In some complex projects, you may use several drones simultaneously, so each drone must have a unique name to differentiate them. Now, we can start using our drone in our program.

As always, we should start our mod with a `main` function. Put your drone definition inside of it. Remind students that although variables can be defined anywhere in your code, the computer has to be aware of it before a variable can be used. Best practice is to set variables at the beginning of your code, so they are the first thing the computer reads.



```
function main
  set d to new Drone
```

Now that our drone is defined, we need to give it some commands. Under the **Drone** tab, get the block **Move Drone 'd' in direction 'up' distance '1'**. Students may notice there are two separate blocks that both say the same thing. The difference between these blocks is the slot for the number *block*, rather than a number space. These blocks would work identically in this lesson, however using the number block is more versatile, as we will see in future lessons. The number block may be replaced with a variable, allowing the drone command to be modified with code.



```
function main
  set d to new Drone
  Move Drone d in direction up distance 1
```

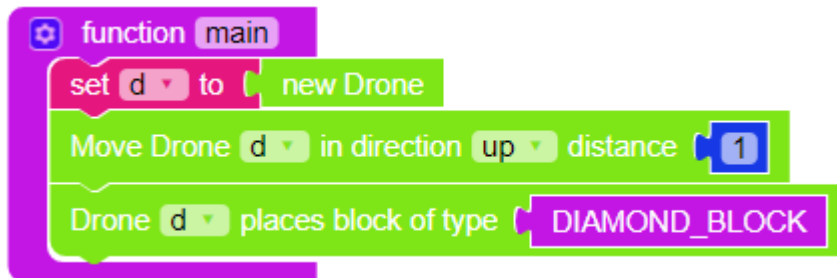
Now that we have a movement command, let's tell the drone to place a block. Find a **Drone 'd' places block of type \_\_\_\_** command, in the **Drone** tab. Fill the empty slot with a **DIAMOND\_BLOCK** block from the **Materials** tab, [D-G].

**NOTE:** This mod will not work if the empty space is filled with a **DIAMOND** block, rather than a **DIAMOND\_BLOCK** block. This is because **DIAMOND** simply refers to the raw material, whereas **DIAMOND\_BLOCK** refers to an actual object that may be placed in the world.



## Final Mod

Our mod will now create a drone, move it up, and command it to place a diamond block. Your code should look something like this:



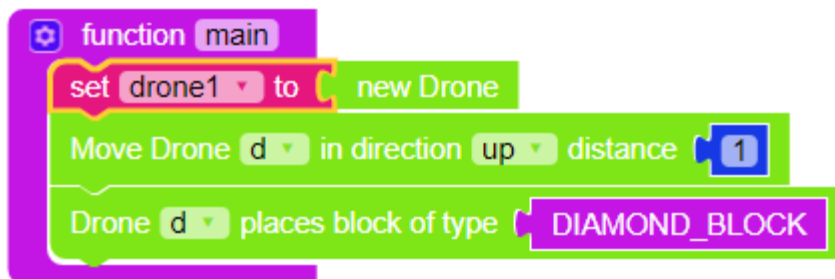
```
function main
  set d to new Drone
  Move Drone d in direction up distance 1
  Drone d places block of type DIAMOND_BLOCK
```

Students can repeat these commands to make the drone place multiple blocks and build simple structures. If students have already learned to use loops, challenge them to use a loop to build a tower, with this code.

Test your mods by clicking the “MOD” button at the top of the screen and running your mod in Minecraft.

## Common Questions and Errors

- The most common mistake for students to make a new variable (for example, `drone1`), but then forget to change the default variable, ‘`d`’, in other blocks of code. For example...



```
function main
  set drone1 to new Drone
  Move Drone d in direction up distance 1
  Drone d places block of type DIAMOND_BLOCK
```

This will produce an error. The variable `drone1` is defined, but never used in the function. The second and third blocks call for a drone named ‘`d`’ that has not been defined in the function. This error can be easily solved by either changing ‘`d`’ in the second and third blocks to `drone1`, or by renaming `drone1` to `d`.

```
function main
  set drone1 to new Drone
  Move Drone drone1 in direction up distance 1
  Drone drone1 places block of type DIAMOND_BLOCK
```

- As noted above, some students may also tell their drone to place blocks of type **DIAMOND**, rather than **DIAMOND\_BLOCK**. This will not work, as **DIAMOND** simply refers to the material, whereas **DIAMOND\_BLOCK** refers to an actual object that may be placed in the world. Explain that while this seems like a small difference to us, it's a pretty big one to the computer!

# Locations: Chapter 1, Lesson 5

## Prerequisite: Drones

### Goal

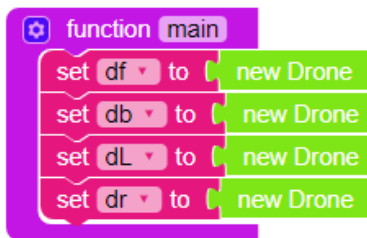
- Strike lightning at the location of a four different drones
- Work through the Drones and Locations badges

### Definition

So far we've been using a single drone to build structures. However, we can control multiple drones at the same time. This will allow us to make things happen simultaneously, in multiple locations.

### Code

In this lesson, we are going to create a mod that will strike lightning 5 blocks in front of, behind, to the left, and to the right of the player. First, we need to create 4 drones. We can name them whatever we like, but each name must be unique, so we can control them separately. In this example, I've named them `df`, `db`, `dL`, and `dr`, for drone forward, drone back, drone left, and drone right.



```
function main
  set df to new Drone
  set db to new Drone
  set dL to new Drone
  set dr to new Drone
```

Now let's move the drones! Get four 'Move Drone...' blocks. Change the distance moved for each block to 5. Now, we need to select the direction for each drone. To specify which drone you are moving, click the arrow next to 'd' and choose the appropriate drone. Change the direction by clicking the arrow next to 'up'. Make sure you are moving the right drones in the right directions! This can make it possible to keep track of drones, within complicated mods.

**NOTE:** Blocks may be copied and pasted (**Ctrl C** and **Ctrl V**) or duplicated, by right-clicking on block and selecting **Duplicate**.

```

function main
  set df to new Drone
  set db to new Drone
  set dL to new Drone
  set dr to new Drone
  Move Drone df in direction forward distance 5
  Move Drone db in direction backward distance 5
  Move Drone dL in direction left distance 5
  Move Drone dr in direction right distance 5

```

Our drones should be set to move to the correct locations. All we need to do is strike lightning at the location of the drones! Use four **World strike lightning at \_\_\_** blocks from the **World** tab. To specify the location to strike lightning (our drones' locations), attach the **location of** block from **Entities** to each of our drone variables.

## Final Mod

Your code should look like something like this:

```

function main
  set df to new Drone
  set db to new Drone
  set dL to new Drone
  set dr to new Drone
  Move Drone df in direction forward distance 5
  Move Drone db in direction backward distance 5
  Move Drone dL in direction left distance 5
  Move Drone dr in direction right distance 5
  World strike lightning at location of df
  World strike lightning at location of db
  World strike lightning at location of dL
  World strike lightning at location of dr

```

We now have a mod that will strike lightning all around the player! We created 4 drones, moved them, and told the mod to strike lightning at each of their locations. We can use **location of** blocks to specify the location of many different things in Minecraft. The only limitation is that we must be very clear with

our references. For instance, we cannot say **location of 'EntityType' 'pig'**, as Minecraft has hundreds of pigs, and the mod will not understand which pig to use.

Test your mod by clicking the "MOD" button at the top of the screen and running your mod in Minecraft.

## **Common Questions and Errors**

Since we are working with multiple drones, it is easy to mix up their names. If students are having difficulty, double check the drone names in the commands to make sure each drone is used. Using graph paper to chart out directions can also be helpful, when drone movement is confusing.

# Loops: Chapter 2, Lesson 1

## Prerequisites: Functions, Drones

### Goal

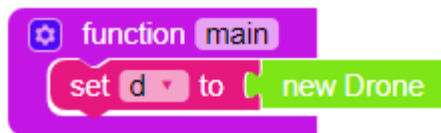
- Create a Mod that uses loops and a drone, to build a 30 cube tall tower
- Work through the Loops badges

### Definition

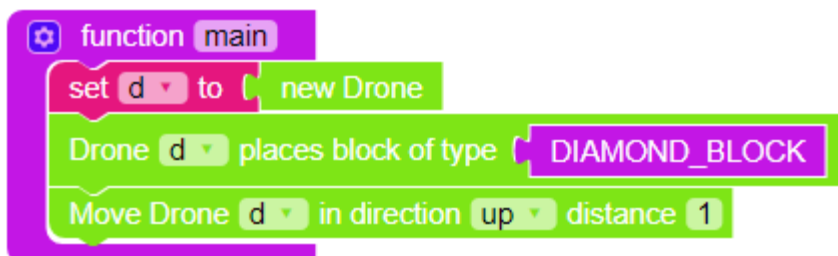
Sometimes (especially when making buildings) we will want to create a mod that will do the same task over and over again. Coding this by hand can be a pain and take a while. However, computers are very good at repeating the same thing over and over. In many cases, we can give the majority of our codework to the computer by using *loops*.

### Code

In this mod, we are going to use a loop to command a drone to build a tower. Create a `main` function and a new drone.

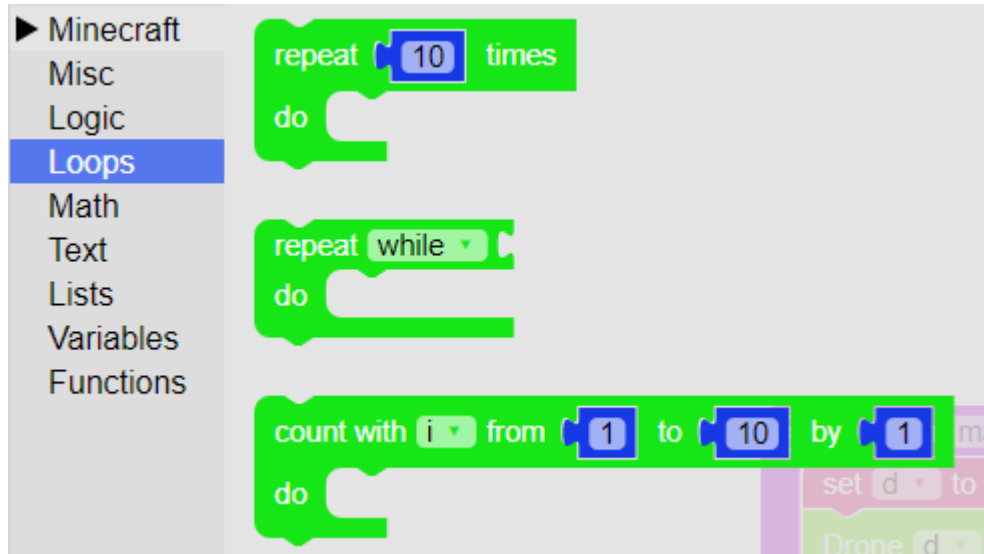


To make a tower, we need the drone to place a block, move up a space, and repeat the pattern until a tower has been created. Get a **Drone 'd' places block of type \_\_\_\_**, from the **Drone** tab, and fill the space in it with **DIAMOND\_BLOCK** from **Materials**. Now get a **Move Drone ... distance '1'** block from **Drone**.

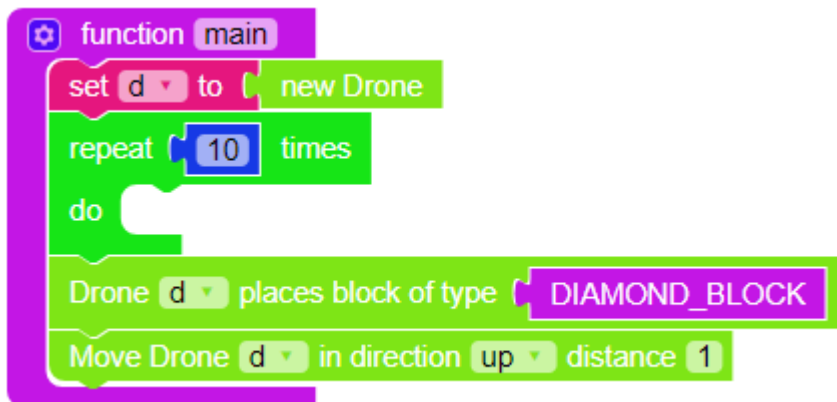


Here's the problem; if we want to create a tall tower, we will need to repeat these two commands over and over, telling the drone to repeatedly place a block and move up. This is boring and inefficient. We can make our lives easier by using a loop.

You'll find multiple options for loops, inside of the **Loops** tab.



There are lots of different variations of loops. All of them are useful, but to begin with, we'll just use the **repeat '10' times** loop. Use this loop inside of your function, right below the new drone.



Now, move the **Drone 'd' places block of type \_\_\_\_\_** and **Move Drone 'd' in direction 'up' distance '1'** blocks into the **do** segment of the loop.

## Final Mod

Your final mod should look something like this:

```
function main
  set d to new Drone
  repeat 10 times
    do
      Drone d places block of type DIAMOND_BLOCK
      Move Drone d in direction up distance 1
```

A loop is a simple and easy way of telling the computer to do the same thing over and over again. In this case, we are telling our drone to place a block and move up. Then, the computer will move back to the start of our loop, and repeat the code. Currently it will repeat this loop 10x, but we can change this by adjusting the number block at the top of the loop, allowing us to easily adjust our mod.

Test your mod by clicking the “MOD” button at the top of the screen and running your mod in Minecraft.

## Common Questions and Errors

The most common mistake for students first using loops is usually simply placing the loop in the wrong location of our code. For example, many students will try to place the **set ‘d’ to new Drone** command inside of their loop.

```
function main
  repeat 10 times
    do
      set d to new Drone
      Drone d places block of type DIAMOND_BLOCK
      Move Drone d in direction up distance 1
```

This will not function correctly. Drones always spawn at the location (block) at which the player is looking, when the mod begins. When the code pictured above starts, it will begin the loop and create a new drone. The drone will move up and place a block. The function will then loop back and replace this drone with a new drone, sending it back to the starting location. Since the drone is being reset at the start of every loop, it will never move above 1 block in height.



## Logical Statements: Chapter 2, Lesson 2

### Goal

Make a mod that strikes lightning after the player has walked a set distance

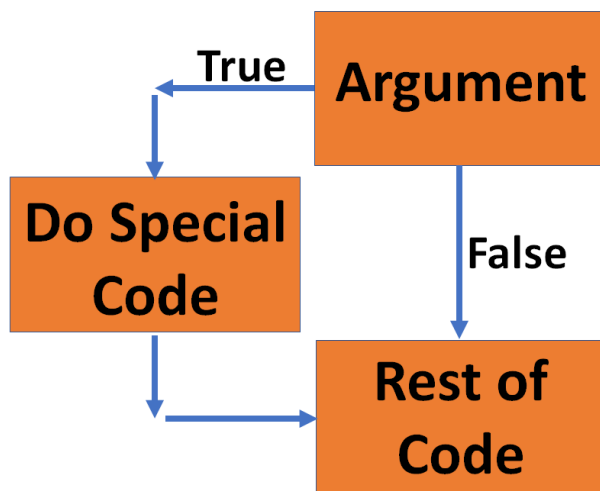
Continue to work through the Loops badges

### Definition

Up until this lesson, we have focused mostly on telling the computer what to do. With *logical statements*, we teach the computer how to think. While there are many different types of logical statements, one of the simplest, but most useful, is the *If statement*.

In the first part of an **If** statement, we give the computer a true/false statement to evaluate. Examples might include determining if: the player is at full health; the player is holding a sword; there are three or more enemies nearby. This part of the **if** statement is called an *argument*. If this argument is true, the computer will *execute* (perform) a specific piece of code. If the argument is false, the computer will skip this code and continue on with the rest of the program.

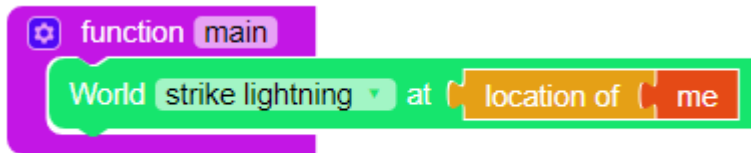
In short, the basic structure is...



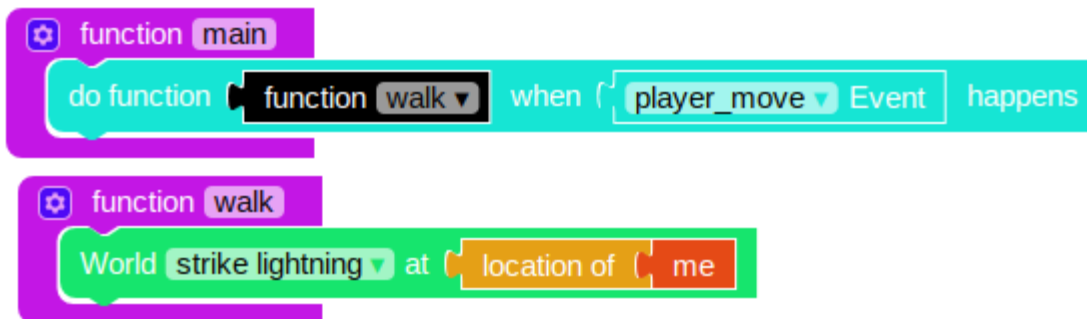
Simple as it seems, *If statements* are at the core of even the most complex programs.

## Code

We are going to create a mod that will count the steps the player has taken and strike lightning every 10 steps. To begin, let's command a lightning strike -- found in the **World** tab. We want the lightning to strike at our location, so we need a **location of** block from the **Entities** tab and a **me** block from the **Players** tab. Put these blocks together and plug them into the remaining blank space. Put the whole command into your `main` function.



Now our mod will strike lightning whenever we run it! To have lightning strike when our character moves around, create and name a new function and move the **World strike lightning** command into it. To add an event, or trigger, use a **do function \_\_\_ when \_\_\_ happens** block (**Events**) in the `main` function. Now fill the second slot with a '**block\_break**' **Event** block. Use the dropdown arrow to the right of **block\_break** to change the type of event. Switch the event type to **player\_move**. Fill the first slot with the black **function 'function'** call block, from the **Misc** tab. Make sure you change the function block to reference your new function. Your mod should now look similar to this:



Our mod will strike lightning everytime we move (walk). We need to add in some logic. In the Variables lesson, we created a mod that would count how many blocks we broke. This mod uses very similar code. Create a new variable in the `main` function named `count`. In our `walk` function, use **Math** blocks to add **1** to this variable.

```

function main
  set count to 0
  do function function walk when player_move Event happens

function walk
  set count to count + 1
  World strike lightning at location of me

```

We need an **if** statement. You'll find **if** statements under **Logic**. Place this block inside of the `walk` function.

```

function main
  set count to 0
  do function function walk when player_move Event happens

function walk
  set count to count + 1
  World strike lightning at location of me
  if
  do

```

IF the player has walked '10' steps, we want the mod to strike lightning. Go ahead and move the **World strike lightning** command into the **do** segment of the **if** statement. Your code should look like this:

```

function main
  set count to 0
  do function function walk when player_move Event happens

function walk
  set count to count + 1
  if
  do World strike lightning at location of me

```

You will see that there is a blank space directly to the right of the word **if**. This blank space is where we will place our argument (the condition part of the logic statement.) To make an argument, we need the **Logic** block: `___ = ___`. Plug this into the blank space.

```

function main
  set count to 0
  do function function walk when player_move Event happens

function walk
  set count to count + 1
  if ( = )
  do World strike lightning at location of me

```

The **if** statement will now check to see if two values are equal. We need to specify the values we want it to compare. We want the function to strike lightning every 10 steps, so we will compare the variable `count` to the number **10**. Fill the two blanks with the `count` variable and a number block from **Math**. Set the number block to 10.

```

function main
  set count to 0
  do function function walk when player_move Event happens

function walk
  set count to count + 1
  if count = 10
    do World strike lightning at location of me
  
```

Last, but not least, we need to reset our count variable after the lightning strike. Add a **set count to** block right after the lightning strike line and connect a 0 number block to it.

## Final Mod

Your code should look similar to this:

```

function main
  set count to 0
  do function function walk when player_move Event happens

function walk
  set count to count + 1
  if count = 10
    do World strike lightning at location of me
    set count to 0
  
```

With this mod, the computer will run some code every time the player takes a step. First, it will increase the step counter by 1, and then it will check if the counter is equal to 10. If `count = 10`, the mod will strike lightning and reset the counter. If the counter does not equal 10, it won't do anything, until the next time the player moves.

Test your mod by clicking the “MOD” button at the top of the screen and running your mod in Minecraft.

## **Common Questions and Errors**

This mod is fairly simple, but has lots of small pieces that can easily get mixed up. If students are having difficulty getting their mod to work, check to make sure their number blocks are set to the correct values. Also, make sure the counter is defined at the beginning of the mod and reset during each **do** function.

## Inner Loops: Chapter 2, Lesson 3

### Prereq: Loops Lesson

### Goal

- Create a Mod that uses loops and drones to build a wall
- Continue to work through Loops badges

### Code

Back in our first loops lesson, we made a mod that would command a drone to build a tower. Our final mod looked something like this:

```
function main
  set d to new Drone
  repeat 10 times
    do
      Drone d places block of type DIAMOND_BLOCK
      Move Drone d in direction up distance 1
```

Now, what if we wanted to make a wall? A wall is pretty straightforward to make; it is basically just a bunch of towers placed side by side. All we need to do is tell the drone to move back down to the ground, move a bit to the left, and then build a new tower!

```

function main
  set d to new Drone
  repeat 10 times
  do
    Move Drone d in direction up distance 1
    Drone d places block of type DIAMOND_BLOCK
  Move Drone d in direction down distance 10
  Move Drone d in direction left distance 1
  repeat 10 times
  do
    Move Drone d in direction up distance 1
    Drone d places block of type DIAMOND_BLOCK

```

Uh oh, this might be a problem. Our wall is only two blocks wide and our code is already getting pretty big. Conveniently, there is an easy solution! We can use *nested loops* -- loops inside of other loops! Let's get rid of the code for the second tower and put the second loop around the code for the first tower. The **set 'd' to** block should not be put into the loops!

```

function main
  set d to new Drone
  repeat 10 times
  do
    repeat 10 times
    do
      Move Drone d in direction up distance 1
      Drone d places block of type DIAMOND_BLOCK
    Move Drone d in direction down distance 10
    Move Drone d in direction left distance 1

```

Since this mod is self contained, it is generally good practice to put it in a function of its own and call the new function from the `main` function.

## Final Mod

Just like that, we have a new mod! Your code should look like this:



```

function main
  wall

function wall
  set d to new Drone
  repeat 10 times
  do
    repeat 10 times
    do
      Move Drone d in direction up distance 1
      Drone d places block of type DIAMOND_BLOCK
    Move Drone d in direction down distance 10
    Move Drone d in direction left distance 1
  
```

When the mod is run, it will create a drone and begin an inner loop. The drone moves up and places a block and repeats this 10 times, to create a tower. When the first tower is done, the drone moves down 10 blocks and moves left 1 block, before looping back to the top and starting over. The nested loop directs the drone to build a 10 x 10 block wall!

Separating code into multiple functions can make organization much easier in large and complex mods. It also minimizes the need for writing the same code over and over. If students want to build walls at multiple places in their mod, it is much easier to simply call the function again.

If students want an extra challenge, encourage them to use a third loop to turn their wall into a giant cube.

Test your mod by clicking the “MOD” button at the top of the screen and running your mod in Minecraft.

## Common Questions and Errors

Sometimes students will confuse the inner loops and add code into it that should stay out. The logic may not be intuitive, at first.

# Functions with Parameters: Chapter 3, Lesson 1

## Goal

- Create a generic function that takes input and performs a specific task using that information
- Work through the Functions with Parameters badges

## Definition

A function can be given (passed) extra values when it gets called; these values are called *parameters* and they are stored in variables. Parameters are a very efficient way of creating functions that are flexible and can be used in multiple, similar operations.

## Code

In our second loops lesson, we created a wall by repeatedly looping our tower function. This is great, unless we wanted to change our tower in any way. For example, what if we wanted our wall to be made of an alternating pattern of diamond block towers and slightly higher gold block towers?



We could accomplish this by creating a second tower function and repeating them one after the other. However, this is inefficient. For example:

```

function main
  set d to new Drone
  repeat 10 times
    do
      repeat 10 times
        do
          Move Drone d in direction up distance 1
          Drone d places block of type DIAMOND_BLOCK
        do
          Move Drone d in direction down distance 10
          Move Drone d in direction left distance 1
      repeat 13 times
        do
          Move Drone d in direction up distance 1
          Drone d places block of type GOLD_BLOCK
        do
          Move Drone d in direction down distance 13
          Move Drone d in direction left distance 1
    do

```

With a few simple changes however, we can make our wall with half the code.

We will create a new tower function that very similar to the mod we created in the first loop lesson -- with some small changes. Instead of embedding the specific number of blocks and block materials into the individual commands, we are going to use parameters to input the desired number and material for each tower, *as the code progresses*. (Think about switching out a peripheral with “plug and play.”) This way, we can change the pattern for height and block type of the towers, without rewriting each step over and over.

Create a `main` function and a drone variable. Create another function (example uses `tower`) for the towers’ code.

```

function main
  set d to new Drone

function tower
  repeat 10 times
  do
    Move Drone d in direction up distance 1
    Drone d places block of type
  Move Drone d in direction down distance 10
  Move Drone d in direction left distance 1

```

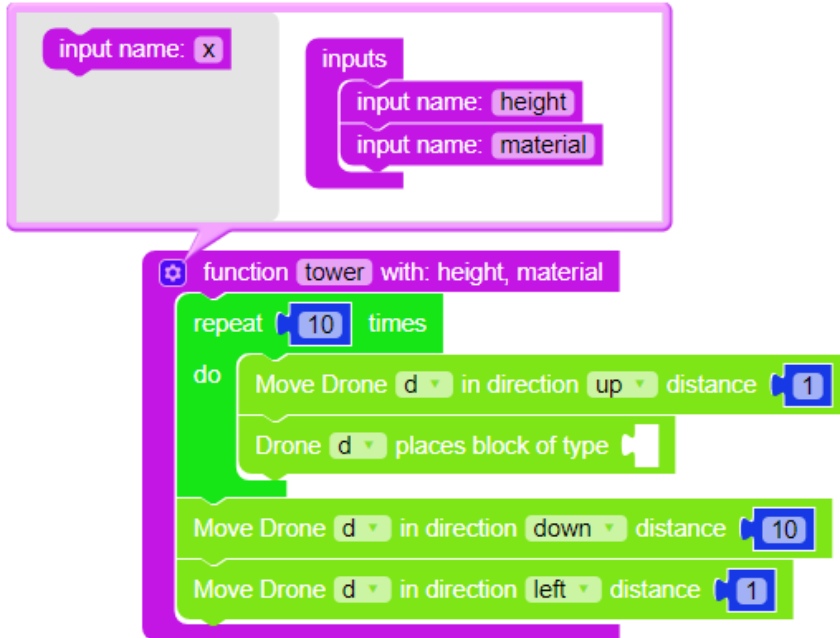
For drone movement, use the **Move** blocks with the blue number variable block in them.



We use these **Move...** blocks because we can replace the number blocks with variables. These variables are going to be a little bit different from our previous variables, however. These are going to be *parameter variables*. To make a parameter, click the blue **gear** icon on the **function's** top left corner.



The **gear** icon opens an input window to create parameters. Drag two **input name: x** blocks into the **inputs** section, and name them `height` and `material`. Click the **gear** again to close the window.



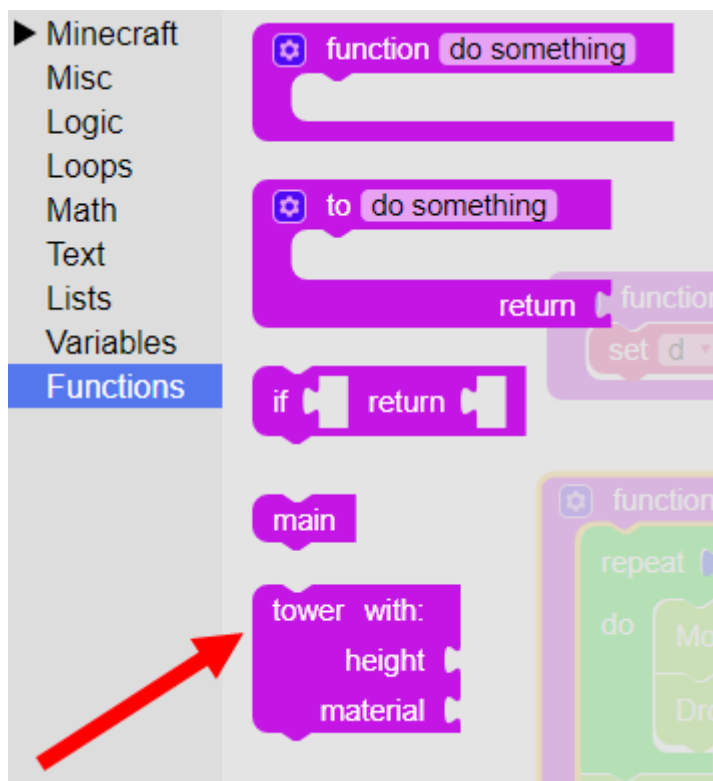
It may not look like much has changed, but it has! Check out the **Variables**. You'll notice that there are now variables for `material` and `height`. Also, at the top of the function, **with: height, material** appears. Plug the `material` variable into the **drone 'd' places block of type \_\_\_** command, and place a `height` variable into the **repeat \_\_\_ times** slot of our loop and in the **Move Drone 'd'...distance \_\_\_** slot. Your code should now look like this:

```

function tower with: height, material
  repeat height times
  do
    Move Drone d in direction up distance 1
    Drone d places block of type material
  Move Drone d in direction down distance height
  Move Drone d in direction left distance 1

```

When we run this function, the computer will check to find the values for `height` and `material` and use those values inside of our code. We just need to tell the computer what these variables contain. Click **Function** to call your second function from the `main` function.



As you can see, our function call now contains two extra slots; one for each of our parameters! Click on the function call, and place it in your `main` function. Let's plug some blocks into these slots. Since `height` is a number, we'll want to put a number block (**Math**) into this slot and set it to 10. `Material` will represent what we want our tower to be made of, so put **DIAMOND\_BLOCK** block into this slot. Your `main` function should look like this:

```

function main
  set d to new Drone
  tower with:
    height 10
    material DIAMOND_BLOCK

```

When we call our `main` function, we specify that we want to build a tower with a height of 10, made of diamond blocks. We can change these parameters, every time we call the function, if we want. Add another `tower` function call (or the name of your function), but this time, set `height` to 13 and set `material` to **GOLD\_BLOCK**.

```

function main
  set d to new Drone
  tower with:
    height 10
    material DIAMOND_BLOCK
  tower with:
    height 13
    material GOLD_BLOCK

```

```

function tower with: height, material
  repeat height times
    do
      Move Drone d in direction up distance 1
      Drone d places block of type material
  Move Drone d in direction down distance height
  Move Drone d in direction left distance 1

```

Now, our drone will build a diamond block tower that is 10 blocks high and then build a tower that is 13 blocks high and made of gold. If we set a loop around our function calls, we'll build a wall of alternating height and material.

## Final Mod

```
function main
  set d to (new Drone)
  repeat 10 times
    do
      tower with:
        height 10
        material (DIAMOND_BLOCK)
      tower with:
        height 13
        material (GOLD_BLOCK)

function tower with: height, material
  repeat height times
    do
      Move Drone d in direction up distance 1
      Drone d places block of type material
  Move Drone d in direction down distance height
  Move Drone d in direction left distance 1
```

This mod begins by creating a drone that builds a tower, with 10 specified for the height parameter and **DIAMOND\_BLOCK** specified for the material parameter. Next, the drone creates another tower with 13 specified as the height, and **GOLD\_BLOCK** specified as the material. We repeat this 10 times to create a wall of alternating materials and heights!

Under the appropriate circumstances, parameters are incredibly useful. It is possible to create mods that achieve similar results, with a creative use of variables and loops, but parameters help to streamline the process. Additionally, more advanced ways of using parameters allow us to access details (attributes, characteristics) of our Minecraft world not contained in any of the LearnToMod blocks.

Test your mod by clicking the “MOD” button at the top of the screen and running your mod in Minecraft.



## Common Questions and Errors

While programming with variables, it is very important that the correct variables are placed in the correct places. Some students may mix up their `material` and `height` variables. Others may try to replace all number blocks with `height` variables, which prevents the `tower` function from getting the necessary `height` information from `main`. If students are having difficulty keeping their variables straight, encourage them to get some graph paper and draw step by step what their drone is doing at every step of the mod.

Parameters may confuse some students, because they require using variables to abstractly represent different values at different times. Practicing with the Parameters badges, in LearnToMod 'Skills and Drills', will help with this.

## Events with Parameters: Chapter 3, Lesson 2

### Goal

- Create a function that will retrieve information from the player chat
- Work through the Game Events badges

### Definition

When some events occur, we want to retrieve and use specific information related to an event. For example, in this mod, we are going to retrieve the message a player sent during a **player\_chat** event. To retrieve this information, we will need parameters. *Parameters are a type of variable that can be passed to a function when it runs (or is triggered by an event).*

### Code

We are going to create a mod that will dress the player in a full suit of armor, when the player types 'on' in the chat, allowing them to quickly prepare for battle!

**NOTE:** To trigger a **player\_chat** event, the player simply type `T` to comment in the chat.

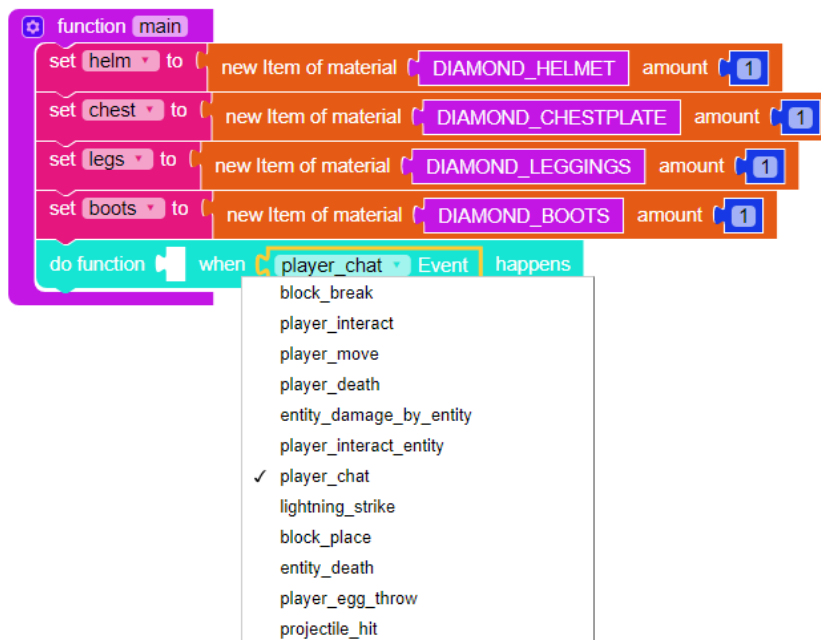
First, we need to prepare some variables. Create four variables: `helm`; `chest`; `legs`; and `boots`. (It is suggested that students use the variable and function names used in the example, to help to keep the code clear.) We need to set these variables to represent different pieces of armor. To do this, get four **new Item of material \_\_\_ amount \_\_\_** block from the **Players** tab and connect one to each of our variables. Set a number block from the **Math** tab to `1` and place it in the second slot of each block. Now, get one each of the **DIAMOND\_HELMET**, **DIAMOND\_CHESTPLATE**, **DIAMOND\_LEGGINGS**, and **DIAMOND\_BOOTS** blocks out from **Materials** [D-G]. Place these blocks into the remaining slots of their respective variables.



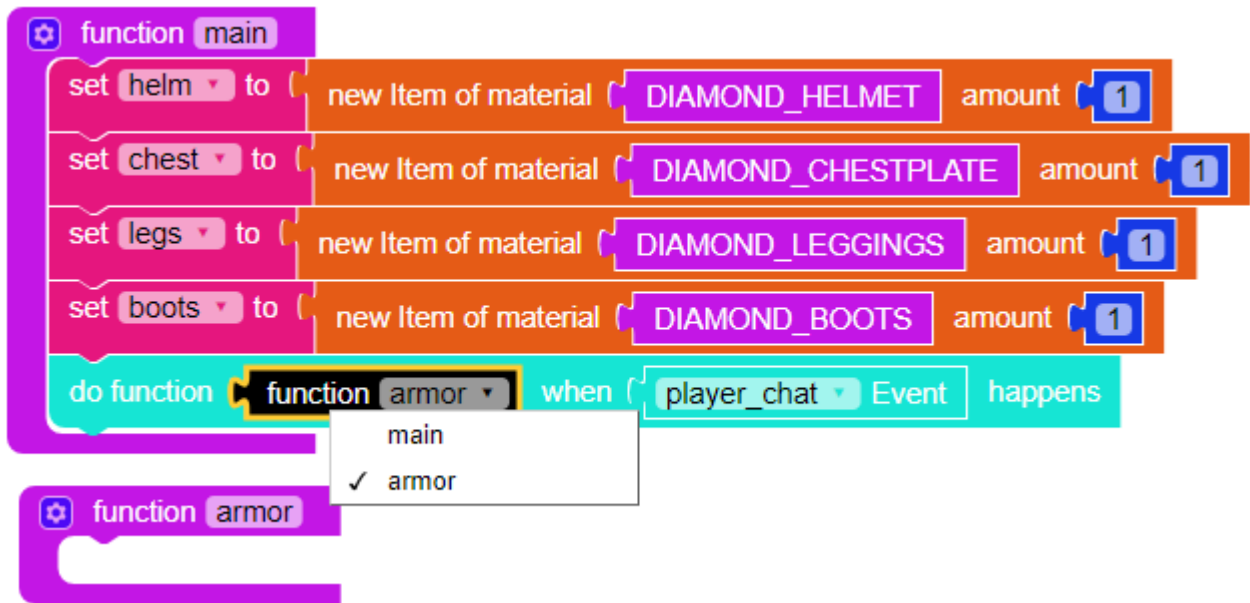
**NOTE:** Some students may try to simply connect the armor blocks (DIAMOND\_HELMET, DIAMOND\_CHESTPLATE, etc.) directly to the variable. While this seems logical, it will not work.

The armor blocks represent a *type of item, rather than a specific instance of it*. Just as I know what a shirt is, but I can't give you one unless I have one, the computer knows what a DIAMOND\_HELMET is, but it does not have one for us to wear, unless we tell it to make one. By using the **new Item of material...** command, we can tell the computer to build (an instance of) an item for us.

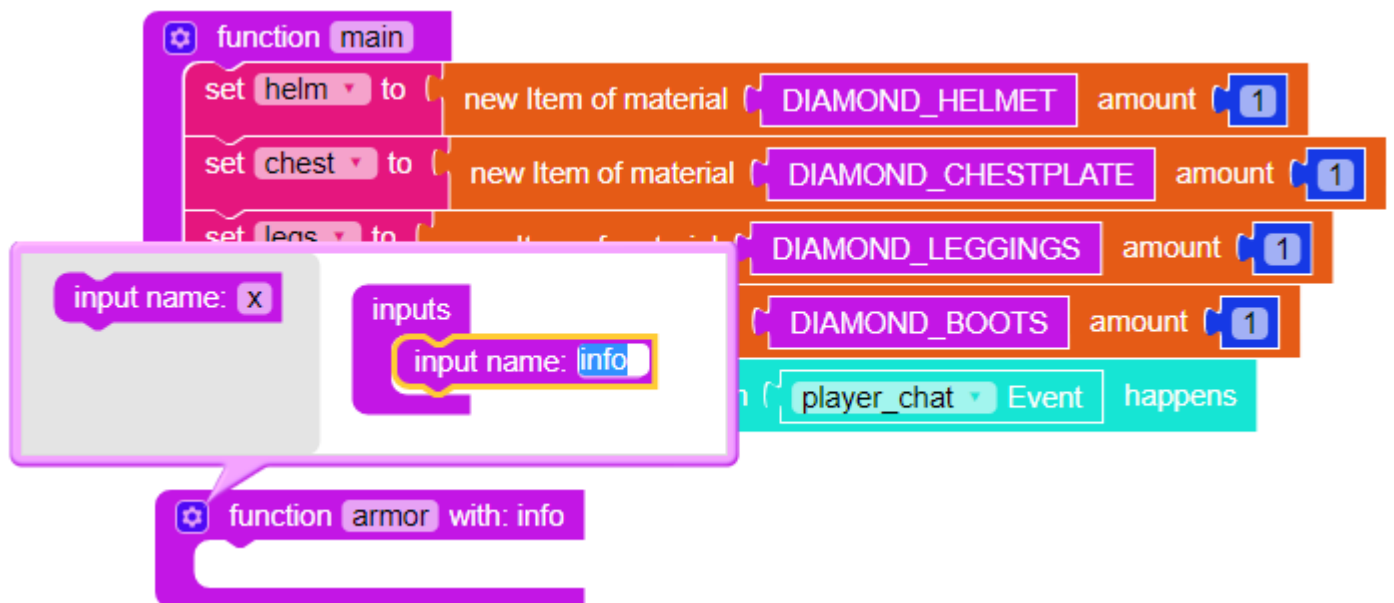
Next, we need to create an event. We want our mod to be controlled by commands the player puts into the chat. To do this we need a '**player\_chat**' event. Click **Events** and the **do function \_\_\_ when \_\_\_ happens**. Get the '**block\_break**' Event block and place it into the second blank space. Change the type of event to **player\_chat** by clicking the dropdown arrow to the right of **block\_break**.



We'll need a function to run as well. Create a second function and name it **armor**. Under **Misc**, use a black **function 'main'** block and place it into the first space in the **do function \_\_\_...** block. Now, use the dropdown arrow to switch the function from **main** to our new **armor** function.



We are going to do something a little bit different with this function. We want this function to retrieve some information from the event that started it. In this case, we want the function to get the *message* the player typed into their chat. We will need to add an **input** to our function. Click the **gear** icon on the *armor* function to pull up the input window. Drag an **input name: 'x'** block into the **inputs** block. Now, type *info* in where the *x* is. When you click on the **gear** icon to close the input window, you will see that the second function says: **function 'armor' with: 'info'** .



**Info** contains a lot of, well, info. We need to isolate which piece we want (in this case, the *players message*), in a variable. Create a new variable and name it `player_message`. Now, under **Misc**, click

the 'item' 's default' block. Connect this to the `player_message` variable. Replace 'item' with `info` and replace 'default' with 'message'.

NOTE: **Message** is one of many pieces of information that is stored inside **info**.

```
function main
  set helm to (new Item of material (DIAMOND_HELMET) amount (1))
  set chest to (new Item of material (DIAMOND_CHESTPLATE) amount (1))
  set legs to (new Item of material (DIAMOND_LEGGINGS) amount (1))
  set boots to (new Item of material (DIAMOND_BOOTS) amount (1))
  do function (function armor) when (player_chat Event) happens

function armor with: info
  set player_message to (info's message)
```

The **info's message** block will search through all of the available information inside of the parameter, and retrieve a variable called 'message.' Think of **info** as a *chart or array of variables*, where, in each line of the table, one column *identifies the variables by name* ('message') and the other column contains the content/value of that variable. The variable 'message' contains the *text the player typed* into `player_chat`. We are saving this text in our own `player_message` variable, so we can easily use it.

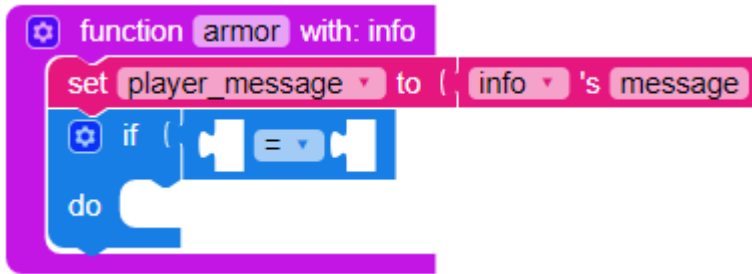
What the parameter variable code -- `set player_message to 'info's message'` -- is saying:

- Create a function with the **info** parameter;
- Define a new variable called `player_message`;
- Set up our `player_message` variable to receive the message (text) typed by the player -- stored in the **info** array (table) under the variable name **message**.

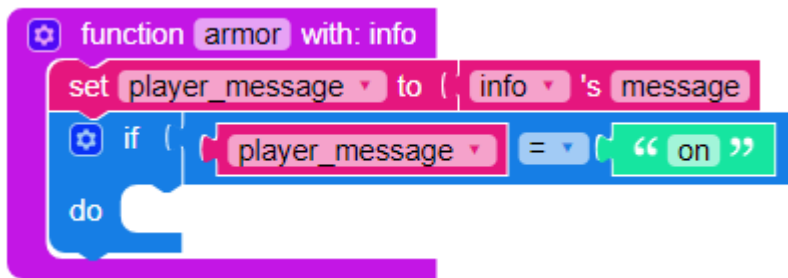
We want our function to check the message the player sent. If the player sent the message "on" we want the mod to give us a full set of armor.

Open up the **Logic** tab, and grab an **If do** block. Place this underneath our new variable.

You will see that there is a blank space directly to the right of the word **if**. This blank space is where we will place our argument. To make an argument, we need another logic block. Open **Logic** and get a **\_\_\_ = \_\_\_** block. Plug this into the blank space.



Now the **if** statement will check to see if one thing is equal to another. We want the mod to dress the player in armor when they type “on” into the chat, so let’s check to see if the `player_message = “on”`. Get the `player_message` variable from the **Variables** tab and place it into the first blank slot. Then put a blank text block from the **Text** tab and place it in the second slot. Now, type `on` into the text block.



Our function will now check the player message. If the player has typed “on” into the chat, the function will perform the code in the **do** section of the **if** block. Let's put some code in there!

Under the “**Players**” tab, look for the block **Change armour piece helmet to item \_\_\_ for player \_\_\_**. Place four of these blocks into the **do** section of our **if** function. These blocks will change the player armor to the variables’ values we specified in our `main` function. The grey dropdown space in each **Change armour piece \_\_\_...** block should correspond to a different armour piece. Put the matching variable into the first blank slot (grey ‘**helmet**’ to `helm`, and so on). Finally, fill the second blank slot of each block with a **me** block from the **Players** tab.

## Final Mod

Your finished mod should look like this:

```

function main
  set helm to new Item of material DIAMOND_HELMET amount 1
  set chest to new Item of material DIAMOND_CHESTPLATE amount 1
  set legs to new Item of material DIAMOND_LEGGINGS amount 1
  set boots to new Item of material DIAMOND_BOOTS amount 1
  do function function armor when player_chat Event happens

```

```

function armor with: info
  set player_message to (info's message)
  if (player_message = "on")
  do
    Change armour piece helmet to item helm for player me
    Change armour piece chestplate to item chest for player me
    Change armour piece legs to item legs for player me
    Change armour piece boots to item boots for player me

```

Whew, that was a long one! Our final mod is pretty cool, though! Have your students enter Minecraft and run the mod. To test it, press `T` to open the chat window, and then type in `on` (case sensitive!!) You should find yourself fully donned in diamond armor! You can check by pressing `F5` to look at your character. If you are playing in Survival mode, you can also check by opening your inventory, or looking for the armor icon above your health.

This is just one example of the many parameters that can be retrieved from events. For a full list of available parameters, check out our “Functions with Info” guide.

Be sure to test your mod by clicking the “MOD” button at the top of the screen and running your mod in Minecraft.

## Common Questions and Errors

As noted above, connecting armor blocks directly to the variable will not work. It is the difference of the computer being able to identify what the armor block is rather than producing one for the player to wear, in response to code.

Some students may have difficulty getting their mod to work. In these cases, check the spelling, capitalization and punctuation of “`on`” in their code and when they type it in the chat. Remind them

that while “On”, “on”, “ON”, and “on.”, all mean the same thing to us, they are very different for computers. The computer will only do our function if our command is written exactly as we have defined it in our code.



# Events with Player's Location: Chapter 3, Lesson 3

## Goal

- Create a Mod that checks the player's location and strikes lightning when they are standing on a specific block
- Work through Game Events badges

## Definition

We have worked a lot with location, but what if we want to make a mod that will execute *only* when the player is at a specific point in their world? This is actually pretty easy to do!

In Minecraft, the location of any object is defined with XYZ coordinates, just like a graph! This means that any location in the game can be defined with just three simple numbers! To find coordinates in game, press F3. This will bring up all kinds of information about the game, including the location of your player.



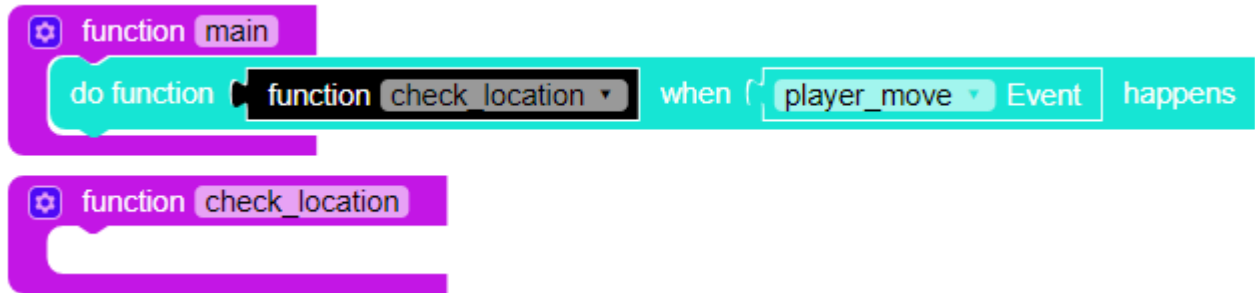
Above: XYZ coordinates are marked with red line around them.

**NOTE:** Some students may not find using the coordinates easy, at first. Remind them that x and y are the directions that they can use on a flat piece of graph paper, while z would require depth, or a 3D model.

## Code

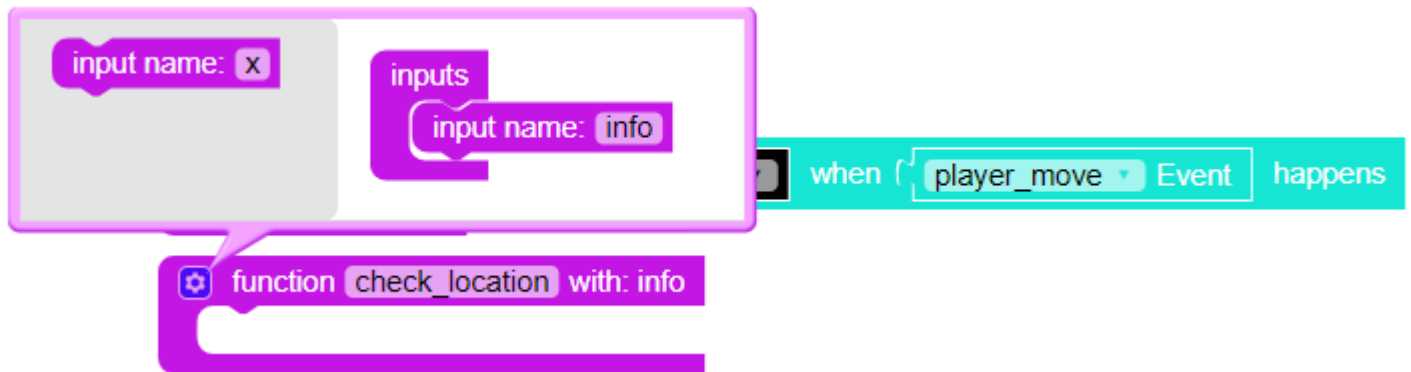
As always, we will start with a `main` function. We will need a **do function** `__` **when** `__` **happens** inside `main`. Fill the second slot with **'block\_break'** **Event** and change the event type to

**player\_move**. Now create a second function (for example, `check_location`) and fill the first slot with the **function** 'function' block from the **Misc** tab.

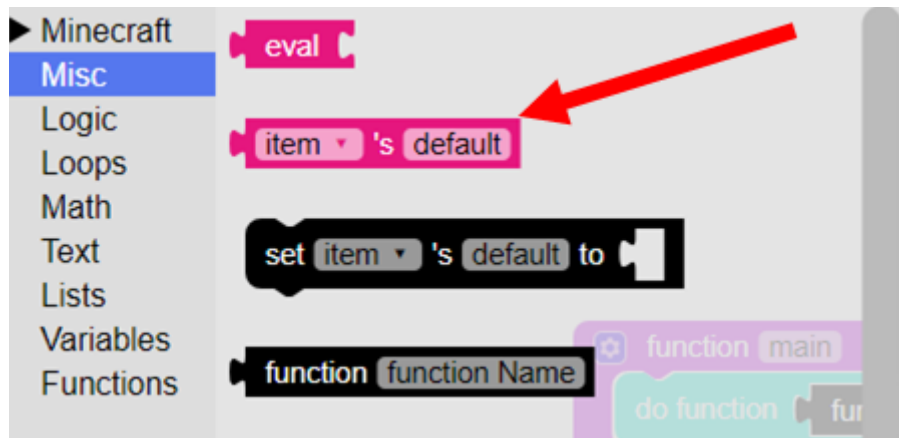


Make sure your **function** block is set to the function you just created!

Now, we are going to add some parameters to our function. Click the **gear**, and add a new **input**. Name this input `info`.



`Info` is a unique parameter. *Instead of setting our parameters when we call a function, `info` will automatically take information from the event that called the function.* In this case, we want to pull the location of the player from `info`. We will need a couple of steps to get to this information. First, get the **'item' 's default'** block from **Misc**.



*Info is not a single variable; it acts as a container for a bunch of different variables. We want to access the variable that refers to the player. To do this, change **item** to `info` and change **default** to **player**. Now, create a new variable called `me` and set it to 'info' 's 'player'.*

```
function main
do function function check_location when player_move Event happens

function check_location with: info
set me to (info 's player)
```

The variable `me` will now refer to the player who activated the `player_move` event.

`me`, like `info`, has a lot of variables stored inside of it. We have narrowed things down significantly; all of the variables contained within `me` refer to the player. We are interested in our location, so create another variable and call it `coordinates`. Now, using another 'item' 's 'default' block, change its values to 'me' and 'location'. This code will retrieve a variable called `location` from the pile of variables associated with our character (`me`), and save it as a variable we can easily access.

```
function main
do function function check_location when player_move Event happens

function check_location with: info
set me to (info 's player)
set coordinates to (me 's location)
```

Almost there! As I said at the beginning, `location` is made up of three numbers, an `x`, a `y` and a `z` coordinate. We are going to create three new variables and set each to one of these numbers. Create three new variables called `x_coordinate`, `y_coordinate`, and `z_coordinate`. Set each of these variables to `coordinates 's x`, `coordinates 's y` and `coordinates 's z`, respectively.

```

function main
do function function check_location when (player_move Event happens

function check_location with: info
set me to (info's player)
set coordinates to (me's location)
set x_coordinate to (coordinates's x)
set y_coordinate to (coordinates's y)
set z_coordinate to (coordinates's z)

```

And just like that, we have three variables that will tell us the player's x, y and z coordinates. We can use these to determine where the player is!

**NOTE:** We can also determine the player's location without using parameters with the following code:

```

function main
do function function check_location when (player_move Event happens

function check_location
set player_loc to (location of (me))
set xLoc to (player_loc's x)
set yLoc to (player_loc's y)
set zLoc to (player_loc's z)

```

Truthfully, this method of obtaining location is likely more efficient for identifying the player. However, using we feel that using parameters is an important enough skill that it deserves reiteration here.

Now, we need to define where we want the player to be, for our mod to activate. There are a few ways to do this, but the easiest is probably to simply walk there and check!



In my case, I'm going to use **210, 29, 112** as my **x, y, and z** coordinates. You will likely need different coordinates for your mod, however, to make sure your mod activates near your current location. To make the mod perform certain actions when the player reaches a location, we need to use some logic blocks.

Get an **if do** block from the **Logic** tab. We want to check the player's **location**, in relation to the location where we want our event to occur. Use a **\_\_\_ = \_\_\_** block, and then plug in your **x\_coordinate** variable, and a number block containing the desired corresponding coordinate.

```
function main
do function function check_location when (player_move Event) happens

function check_location
set me to (info's player)
set coordinates to (me's location)
set x_coordinate to (coordinates's x)
set y_coordinate to (coordinates's y)
set z_coordinat to (coordinates's z)
if (x_coordinate = 210)
do
```

As you can see, we have a bit of a problem. We only have space for one argument in our **if** block, but we need three arguments to define our coordinates. We can solve this problem by using an **and** block. This block is also inside of the **Logic** tab. You can plug multiple arguments into an **and** block. With the **and** block, the **if** block will check to make sure that *both* of our criteria are true. If either are false, the **if** block will not execute the code in the **do** section. Detach and save the `x_coordinate=___`. Create another `___=___` block and fill this one with your `y_coordinate` values. Put the `x_coordinate` and `y_coordinate` argument blocks into the **and** block, and attach it to the **if** block.

```

function main
do function function check_location when player_move Event happens

```

```

function check_location
set me to info's player
set coordinates to me's location
set x_coordinate to coordinates's x
set y_coordinate to coordinates's y
set z_coordinate to coordinates's z
if (x_coordinate = 210 and y_coordinate = 65)
do

```

That takes care of two of our coordinates. For our z coordinate, let's use another `___ = ___` block to create a similar `z_coordinate` condition and then take our first **and** block and place it into the left slot of yet another **and** block. The `z_coordinate` block will fit into the right slot of the **and** statement.

```

function main
do function function check_location when player_move Event happens

```

```

function check_location
set me to info's player
set coordinates to me's location
set x_coordinate to coordinates's x
set y_coordinate to coordinates's y
set z_coordinate to coordinates's z
if (x_coordinate = 210 and y_coordinate = 65 and z_coordinate = 92)
do

```

**NOTE:** Minecraft does not measure location in whole numbers. Instead it calculates decimals, down to the thousandths place (or in the case of height, the millionths place). This provides a bit of a problem for defining a location. Currently, our mod is checking to see if our location is equal to equal to EXACTLY 210.000, 65.00000, 92.000. This point is incredibly small, and it is unlikely the player will ever enter these exact coordinates.

The easiest way to solve this problem is to define our location in terms of an area, rather than as a specific point.

First, change all of the `=` symbols to `≥` symbols by clicking the triangle next to them.

```

function main
do function function check_location when player_move Event happens

function check_location
set me to info's player
set coordinates to me's location
set x_coordinate to coordinates's x
set y_coordinate to coordinates's y
set z_coordinate to coordinates's z

if (round(x_coordinate) ≥ 210 and round(y_coordinate) ≥ 65 and round(z_coordinate) ≥ 92)
do

```

Now the mod will activate when the player's location is greater than or equal to our coordinates.

Now we need to add some further restrictions, so the mod does not activate in ALL locations above our desired coordinate. Copy the **and** blocks we have created so far, and change all of the  $\geq$  symbols to  $<$ . Also change each number block to 1 greater than its current value.

```

(round(x_coordinate) < 211 and round(y_coordinate) < 66 and round(z_coordinate) < 93)

```

Now, we need to plug all of these arguments together! We can simply create another **and** block to connect these new arguments to our old ones. This works great, and the only downside of it is that it creates a really big block that can be hard to view all at once. For ease of viewing in this guide, I am instead going to create a second **if** block inside of the **do** function of our first **if** block, and plug our new arguments into it. In this particular case, these two methods are functionally identical, it simply adjusts the order that the arguments are evaluated in.

```

function main
do function function check_location when player_move Event happens

function check_location
set me to info's player
set coordinates to me's location
set x_coordinate to coordinates's x
set y_coordinate to coordinates's y
set z_coordinate to coordinates's z

if (round(x_coordinate) ≥ 210 and round(y_coordinate) ≥ 65 and round(z_coordinate) ≥ 92)
do
  if (round(x_coordinate) < 211 and round(y_coordinate) < 66 and round(z_coordinate) < 93)
  do

```

Now, our mod will check to see if they players coordinates are equal to, or above, our desired values (first **if** block). If they are, the mod will use the second **if** block to make sure their coordinates are not greater than 1 block away from the desired values. If this is also true, the mod will continue to the **do** section of the second **if** block. To finish off, let's put some code in this section! Grab a **World strike**



**lightning at** \_\_\_\_ block from the **World** tab. For our location, let's use the player's location, since we conveniently already have this saved in our `coordinates` variable. Just plug this variable into the empty space and we are all done!

## Final Mod

```
function main
do function function check_location when player_move Event happens

function check_location
set me to info's player
set coordinates to me's location
set x_coordinate to coordinates's x
set y_coordinate to coordinates's y
set z_coordinate to coordinates's z

if (round(x_coordinate) >= 210 and round(y_coordinate) >= 65 and round(z_coordinate) >= 92)
do
if (round(x_coordinate) < 211 and round(y_coordinate) < 66 and round(z_coordinate) < 93)
do
World strike lightning at coordinates
```

Now, whenever the player moves, the mod will check their location. If the player's **x**, **y** and **z** coordinates are above a threshold value, the mod will check the **x**, **y** and **z** coordinates to make sure they are also below another threshold value. If they are, then the player must be within our desired location, and the mod will create a lightning strike.

Be sure to test your mod by clicking the “MOD” button at the top of the screen and running your mod in Minecraft.

## Common Questions and Errors

There is plenty of room for mistakes in this mod. If a student is receiving an error when they attempt to run their mod, check to make sure their spellings are consistent throughout the mod, otherwise they may be telling the computer to take data from variables that don't exist!

If their mod is not producing errors, but is not causing a lightning strike, double check their **>** and **<** signs, to make sure they are oriented correctly. Students will often get them backwards and create mods that will only activate when the player's **x** coordinate is less than '210', while also being greater than '211'.

We hope you have found this sample curriculum helpful, easy to follow, and complete. If you have comments or questions or find any errors, please send an email to: [mltm@maine.edu](mailto:mltm@maine.edu). Thank you.

Syntax Conventions used in document. New terms of any type are in *Italics*, for the first time; definitions are also in *italics*. References to use any code blocks (LTM, etc.) are **Bold**. User typed code is in *Courier New* font. All references to LTM use the case and naming conventions that are used in the actual Blockly or code examples, for clarity and consistency.