destron media
Game Maker Education

The Complete Beginners Guide To GML Programming

Written by flexaplex

PDF by Destron

Ver 1.2

CONTENTS

Table of Contents

FORWARD

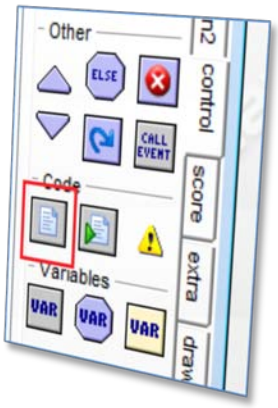
If you are new to programming the prospect of using GML may seem daunting however reading this guide is likely to be a great help to you. How easily you can learn to use GML will depend on how much experience you have already gained using drag & drop and your ability to think logically. However with some practice and learning most people will be able to grasp the general concepts behind coding.

The guide will be best suited for people who have been messing around with DnD (Drag & Drop) for a little while and are now looking to move onto using GML, however it can be useful for all beginning levels but it is suggested that you know how to use Game Maker and it's interface before proceeding with this guide. I am of course always interested in suggestions people have on changing / adding content to the guide and any feedback in general.

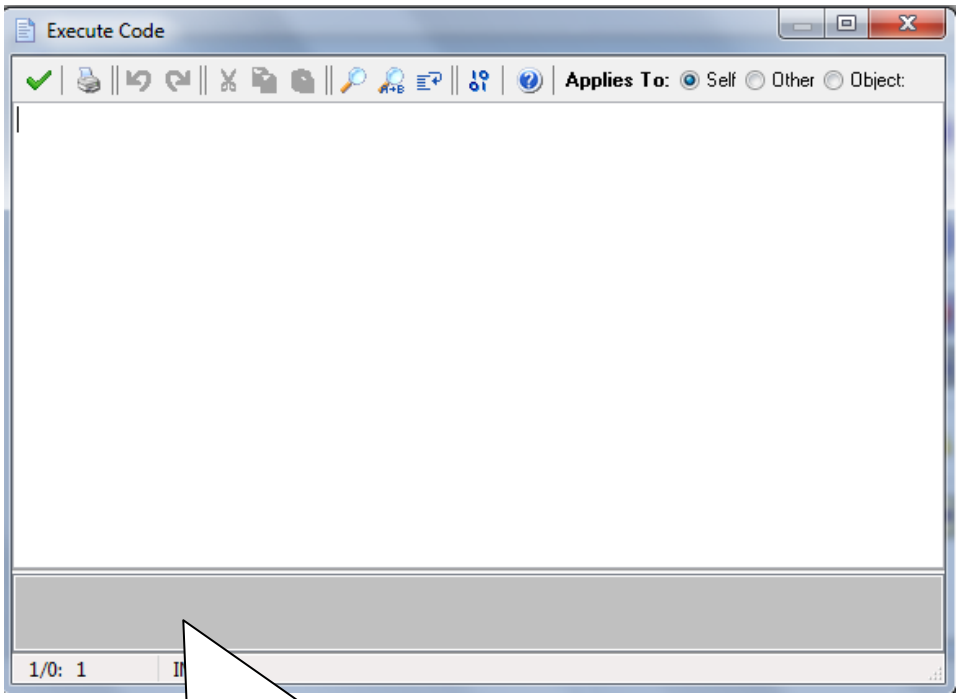
This guide was originally written by flexaplex from the GMC, and is being reproduced in PDF format by Destron, with permission from the original author under the GNU GPL license. If you received this guide without a copy of the GNU GPL license, please visit http://gm.destronmedia.com for a complete archive.

One thing you will notice while programming in GML is that while you are typing, some words will change colors, and some will be bold. This is known as syntax highlighting and it is designed to assist you while you are writing GML in the code editor. All of those colors are not there just to look pretty, they actually mean something. They can help you be sure you are calling resources by their right name, but they can also serve as a warning when you accidently use something you shouldn't. For more information on syntax highlighting see Appendix A. Also, while looking through the code in this guide you will see text *that looks like this;* This text is code comments, a place for the programmer to put in comments so that they will know what they did next time they look at it, or someone else will know what that they did if the release the code or someone else works on it. This text is completely ignored by GM, and can contain anything you like. There are two ways to use comments … You can put // before it , but that only works on the current line. If you need more than one line you can enclose the entire comment with /* and */. You will see examples of both while reading this guide.

Getting Started



The first thing you need to do when wanting to code is to drag an 'execute a piece of code' action into the event where you wish to use code. The action can be found under the 'control' tab of the DnD actions, once you have dragged it into the action area the script editor will open, here is where you type any GML you want to use.



This part of the code editor window can be your best friend at times! As you type code it sill suggest functions for you based on what you have typed, as well as show you the arguments that go along with them. Some of them can use in insane amount of arguments, so this is a handy addition to the code editor!

Functions

A function is separate piece of code which has already been written and has a specific purpose. In Game Maker there are many different in-built functions which you can use to do things, for example:

```
move_towards_point(x,y,sp);    //this moves you towards a point
room_goto_next();              //this makes you go to the next room
draw_text(x,y,string);         //this draws text on the screen
instance_create(x,y,obj);      //this creates an instance of an object
```

You may not be aware of it but you have more than likely used functions yourself already, this is because a lot of the DnD actions are actually just functions. If you look you will see that there is actually a DnD action to **move towards a point**, **goto the next room**, **draw text** and **create an object**. When you call a function in code it is exactly the same as calling the action from DnD except you are just doing it in a different way. There are however a lot more functions you can use in GML which are not available to use in DnD, all of them are listed in the manual with an explanation of what they do.

To use a function correctly in GML you simply have to put the function's name followed by parenthesis (). You can see in the above code that you can also use values inside the parenthesis, these are referred to as arguments or parameters. To add these you must place values between the parenthesis with a comma separating them. Arguments work exactly the same as the values you use in DnD actions, they are simply values which you are passing to the function so it can use them in its code. Note though unlike in DnD. in GML you could now type in the wrong number of arguments; if you do this GM will give you an error as you must use the same number of arguments that is stated in the manual.

When you use functions in GML there is actually an extra element which you don't get with DnD, this is return values. Instead of values you give a script, a return values is a value the script gives to you after it has finished executing. The returned value will be something calculated in the function when it runs and will likely to be of use to you in some way. In the manual it will always state what a function returns, however note that not all functions actually return a value.

So how do you use these return values? Well most often you will want to assign them to a variable which we have not covered yet, so I have explained this later in the 'some uses of variables' section.

Variables

Variables are an essential part of any programming language, GML is no exception. A variable is like a reference for storing a value; this reference has to be given a name. The name can be anything you like as long as it does not clash with any names that already exist. An example:

```
apples = 4;
```

In this example the variable is given the name apples, this variable apples is then assigned (given) a value of 4. In GML the equals sign is always used to assign a value to a variable.

Once you have assigned a variable a value from now on when you call upon this variable at any point it will be the equivalent to calling that value which you assigned to it. For example if you were to now use:

```
fruit = apples;
```

It will set the variable fruit to 4 as the variable apples has the value 4. Or if you were to have another variable called pears with a value of 2 you could use:

```
fruit = apples+pears;
```

fruit would now be set to 6 (as it is 4+2). You could then for example go on further to use this fruit variable:

```
nutrition = fruit*2;
```

This would set the variable nutrition to 12 (the * is the symbol for multiply in GML, so it the previously calculated 6 multiplied by 2 which equals 12).

Strings

As well as 'number' values (the proper terminology being **real** values) variables can also be set as 'text' values (the proper terminology being **string** values). To do this you need to put text inside 2 quotes either " " or ' '. For example you can set a string to a variable like so:

```
name = ‘John’;
```

or

```
name = "John";
```

Using either the variable name will then have a value equal to the string "John". You can now do whatever you wish with this variable just like you could with the apple variable, most commonly you would want to draw the variable, which you can do using:

```
draw_text(x,y,name);
```

This will then draw the text "John" at the instance's x and y position.
Just like real values you can also combine strings together using the + sign. For example if have the variables:

```
first_name = "John";
last_name = "Smith";
```

You can draw the variables joined together; I will show some different formats you might wish to use to do this in:

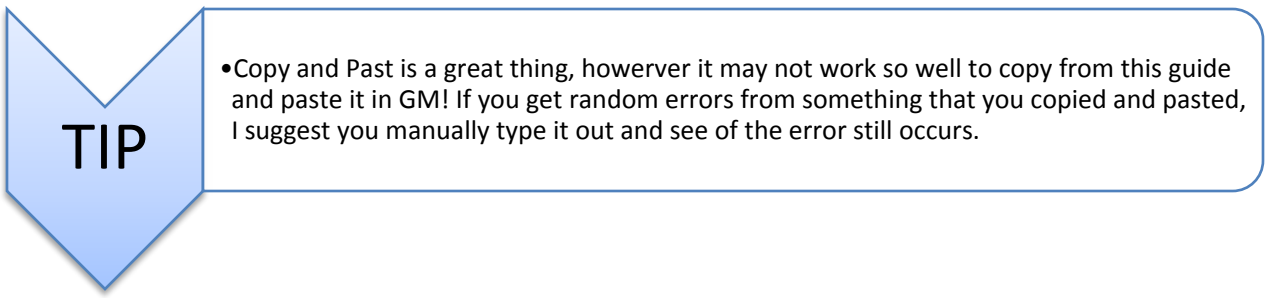
```
draw_text(x,y,first_name+last_name); //this would draw  JohnSmith
draw_text(x,y,first_name+" "+last_name); //this would draw  John Smith
draw_text(x,y,"NAME: "+first_name+" "+last_name);//this would draw
NAME: John Smith
draw_text(x,y,first_name+"#"+last_name);
/*
this one is slightly different as it uses the special character #, when
used in GML it is treated as a return carraige (a new line) so this
would draw

John
Smith */
```

As well joining strings together you may also wish to join a string to a real value. To do this you need to first convert the real value into a string, this can be done via the **string()** function. For example if you use this:

```
num = 1;
item = potion;
draw_text(x,y,item+string(num));
```

It will draw the string "potion1". If you try to combine a string to a real value without using string() first you will get a 'cannot compare arguments' error.



Global/local variables

So far I have been explaining how to use variables but not exactly where they are set to.

- local variables

When you set a variable normally like I have been showing then this variable is set **locally** to the object you are putting the code in (well this is technically not correct it is in fact set locally to every instance of the object but I would not worry about that for now). When a variable is set locally to an object it means that only that object will be able to call upon it. For example if you put this in the creation event of an **obj_player**:

```
life = 5;
```

Then only **obj_player** is going to have the variable power set to it. If you want to call the variable from inside another object like an enemy for example to get how much damage the player can do you need to first put the object name with a dot before it like so:

```
damage = obj_player.life;
```

TIP: The purple text indicates a resource, so if the name stays black, you probably made a mistake!

Doing this calls the object's variable. If you wish to set a variable in another object you should do the same also... i.e. if you use :

```
obj_player.life = 6;
```

It will set the player's life variable to 6. A more common example of when you need to call upon a variable in another object is when calling upon another object's position. For example if you use this in an object's end step event:

```
x = obj_player.x+40;
y = obj_player.y;
```

This will set your position to the right of the player every step. You should note that **x** is a variable just like any other we have been talking about, however it is a special in-built variable... I will go into more detail about these later.

- global variables

As well as setting variables locally to an object you can also set them globally. To do this you just need to put the word **global** with a dot before it. For example:

```
global.name = "Mr Cool";
```

This will now set the **global variable name** to **Mr. Cool**. When a variable is set globally all the objects will have access to the variable, you can also set the variable from any object you wish. To call upon the variable you just need to put the same **global.** in front of it. For example if you want to draw the title you can use:

```
draw_text(x,y,"Name: "+global.name);
```

Most commonly you will want to use global variables when you are changing rooms but want to keep a variable value known. You cannot use local variables to do this as when you leave a room the objects will be destroyed and their variables will be cleared along with them, an example of this would be a game score variable. Something you should note that with such a variable is that you often need to carefully consider where you wish to initially set the variable value. For example if you want to initialize a global game score variable like so:

```
global.game_score = 0;
```

Where would you put this? A common mistake is to put it in the creation event or the room start event of an object in your first level. The problem with this is that if you die and restart the room then the creation/room start event is going to be executed again and the variable is then going to be set back to 0. There are a few ways to deal with this problem; one is to use the game start event instead (you need to make sure the room where you set this is the first room in your game though!) or a more preferable way is to have a menu room at the very start of the game and set the variable in an object there. When you exit to a menu you will probably want to wipe the score anyway... you cannot do this using the game start event unless you completely restart the game when you go want to go back to the menu. You will likely find a menu room is useful for setting a lot of variables in your game if you have one. Generally when setting/changing global variables you would want to do so in a controller object.

- var

There is also another way you can use variables and this is by declaring them with **var**. When you declare a variable with **var** it makes the variable local to just the script you are using it in (think of it like a 'temporary' variable which belongs to the script). This means that no object will be able to call upon it after the script has finished executing and you will not even be able to call it from any other scripts that are executed during the event. So make sure you know that you are not going to use the variable elsewhere otherwise you will get an unknown variable errors. How you declare a variable with **var** is like so:

```
var nearest_enemy;
```

Doing this the variable **nearest_enemy** will now be localized to the script you call it in. All you basically have to do is put the word **var** before the variable (you should see the word go bold), you can do this anywhere in the script (as long as it is before you use the variable). There are some thing to note with the syntax when using **var:**

1) You HAVE to use a semi-colon at the end of the line, in this particular case GM will throw an error if you don't.
2) You cannot assign a value to a variable while declaring it with var, ie you CAN'T do this:

```
var nearest_enemy = enemy1;
```

3) You still need to initialize the variable though (GM doesn't just know what it is), i.e.:

```
var nearest_enemy;
nearest_enemy = enemy1;
```

4) You can declare several variables with var at the same time using a comma, for example:

```
var nearest_enemy, max_health, dist;
```

This will declare **nearest_enemy, max_health** and **dist** all local to the script you use it in.

You might be wondering why you would want to declare a variable local to a script. Well the main reason is memory... a variable declared with **var** is cleared from memory as soon as the script has finished executing. The other reason is to stop variable names clashing. It is considered good practice to declare variables with **var** for scripts when they are only going to be used in the script itself... especially if you release the script for other people to use as you do not want your variable names clashing with variables already in their game.

As well as **var** there is finally another similar way to assign a variable... this is with **globalvar**. This follow all the same syntax rules as declaring with var (need to use a semi-colon, can't assign values, need to initialize still and can declare multiple variables using commas). An example would be:

```
globalvar game_score;
```

When you declare a variable with **globalvar** all scripts will now recognize the variable... it is basically exactly the same as using **global.** before a variable. The advantage of using **globalvar** is you only have to use it once and then the variable is globalised for the rest of the game... now you can use the variable as a global variable without having to put **global.** before it all the time. If you use it though make sure you keep track of which variables you have declared global otherwise you may start making errors.

-In-built variables

Up to now I have only been referring to variables that you create yourself, however there are also some variables that are in-built as well. These variables are set to objects by Game Maker automatically and are used to do specific things within the program. You will probably recognize a lot of these variables but may of not realized that they were variables at the time, for example; **direction, gravity, speed, score, lives and even x, y** are all just **in-built variables**. Just like normal variables there are both local and global variables for the objects, you can see a full list of the **in-built variables** by clicking the scripts tab then **'Show Built-in Variables'**, or see Appendix B, they are also all explained briefly in the manual if you look for them. Whenever you type an **in-built variable** in GML it will turn the color blue, you should take note of this so you do not try and use the name of an in-built variable for one you are trying to create yourself as doing this can cause unwanted results.

Note that you should use in-built variables just like normal variables. You can call upon their values and you can also set your own values to most of them, however some of them you cannot. These are marked as 'read-only' and are depicted with a * character afterwards in the manual, these variables you cannot change the value of as they are only set by Game Maker alone in a specific way.

-Uses of variables

I am not going to show any uses of variables yet as you often want to be using them alongside if statements which I am about to cover, you will see I have put a section on some uses of variables straight after the next if statement section.

If / else statements

If statements are used when wanting things to happen only if a certain condition is met. They take the form:

```
if (a condition is met)
{
perform actions
}
```

When the condition is met (i.e. it is true) then all the actions you put inside the braces {} will be performed. If the condition is not met then none of the actions will be performed. Note that you need a closing brace } for every opening brace that you use { if you do not have an equal number then you will get an error. It is important to use braces so then GM knows which actions it is only supposed to perform when the condition is true. If you don't use braces then only the first action you put will be conditional on the if statement and the others will be executed all the time regardless.

-Checking Boolean Values

A Boolean value is one that can only be true or false. Here is a basic example of testing a Boolean in an if statement:

```
if (place_meeting(x,y,obj_wall))
{
speed = 0;
}
```

This code would check whether you are meeting a wall or not and if so it would set the speed to 0. This is an example of a function returning a value that can only be true or false. You can also check variables value, for example if you have a Boolean variable called **message_allowed**, then you could use this code:

```
if (message_allowed)
{
show_message("Hello");
}
```

This would only show "Hello" when **message_allowed** is set to true and would not do anything if it's set to false. You could also check for a Boolean value being false, this can be done using the **!** operator (which means **not** in GML). i.e.:

```
if !(place_meeting(x,y,obj_wall))
```

Will check if you are **not** meeting a wall, and

```
if !(message_allowed)
```

Will check if **message_allowed** is not true (ie it is false).

-Checking Variable Comparisons

As well as just checking Boolean values you can also check variable comparisons. A basic example would be:

```
if (health == 100)
{
lives = 3;
}
```

This code would only set lives to 3 when health is **equal** to 100. Notice the difference that a single = is used when *assigning* a value to something, but a double == is used when *comparing* values. As well as comparing values to be equal to each other you can also compare:

```
if (health < 100)
```

This will check is if health is **less than** 100.

```
if (health > 100)
```

This will check is if health is **greater than** 100.

```
if (health <= 100)
```

This will check is if health is **less than or equal** to 100.

```
if (health >= 100)
```

This will check is if health is **greater than or equal** to 100.

```
if (health != 100)
```

This will check is if health is **not equal** to 100. NOTE: Make sure you do not try to use if (!health == 100) when trying to check if a variable is not equal to a value, this is a common mistake. The problem is GM will actually read the code as if (!health) == 100, this is something completely different to what you want. If you do want to use this make sure you use brackets in the correct location, i.e. if !(health == 100) this is now the same as using if (health != 100), this error occurs most often when brackets are not used with if statements.

That is all the comparisons you can use in GML.

-Combining conditions

There is also the possibility to combine conditions, for example:

```
if (x > 10 && x < 100)
{
zone = 1;
}
```

This checks if x is greater than 10 **and** x is less than 100 and sets the variable zone to 1 if it is (the && sign means **and** in GML). For this condition to be met **x** *must* be both greater than 10 and less than 100. Another combination you can use is:

```
if (sprite_index == spr_walking || in_air)
{
image_speed = 3;
}
```

This checks if the **sprite_index** is walking **or** the variable **in_air** is true (the || sign mean **or** in GML). For this condition to be met the **sprite index** can be either walking **or** the variable **in_air** can be true or both condition can be true, basically the only time when it will not be met is when the **sprite_index** is not walking and **in_air** is false.

The last way which you can check combinations is using ^^ (the ^^ sign means **xor** in GML). This is exactly the same as || except now the condition is not met when *both* are true, only when one or the other is true. You do not really need to concern yourself with this operator though as it is infrequently used and you are unlikely to need it.

-Else Statements

Else statements are used when wanting something to happen upon the if statement condition not being met. It takes the form:

```
if (condition is met)
{
perform first action
}
else
{
perform other action
}
```

Basically what happens is if the if condition is met it performs the first action as normal otherwise (if it is not met) it performs the other action. For example:

```
if (message_allowed)
{
show_message("Hello");
}
else
{
show_message("Disabled");
}
```

Like normal this will show "Hello" when the variable **message_allowed** is true but now when **message_allowed** is not true it will show "Disabled". Another example:

```
if (x > 10 && x < 100)
{
zone = 1;
}
else
{
zone = 2;
}
```

This will set **zone** to 1 when **z** is greater than 10 **and** less than 100 as shown previously otherwise (if **x** <= 10 or **x** >= 100) it will set **zone** to 2. You can also combine else statements with an if statement and then even multiple else statements if you want. For example:

```
if (value == 3)
{
show_message("1st statement");
}
else if (value == 4)
{
 show_message("2nd statement");
}
else if (value == 5)
{
  show_message("3rd statement");
}
else
{
  show_message("other statement");
}
```

What this does is it will show "1st statement" when value is 3 otherwise it will show "2nd statement" when value is 4 otherwise it will show "3rd statement" if value is 5 otherwise (if value is not 3,4 or 5) it will show "other statement".

-Switch Statements

Switch statements can be used when wanting to perform different actions depending on the value of a single variable. In fact it pretty much is used in replacement to that long if, else code I just wrote. Taking that as an example it could be written instead using a switch statement like so:

```
switch (value)
{
case 3: {show_message("1st statement"); break;}
case 4: {show_message("2nd statement"); break;}
case 5: {show_message("3rd statement"); break;}
default: {show_message("other statement");}
}
```

The advantage of doing so is it is then easier to read and recognize. How a switch statement works is it checks the value given after it in the brackets (often this is just a variable used here) then it checks for any cases you have put which match this value. Upon finding the case it then executes **ALL** the rest of the code after it including the code in other cases. This is why you need to put that **break** there otherwise all the rest of the code will be executed included the cases where the value is not matched, it can be easy to forget to use break so try to always remember. The default case at the end is used when none of the given cases match the value, this is the equivalent of using that final else statement at the end.

It is important to note that all this is doing is checking for a matching **value** it is not actually checking for a variable matching a value. You cannot use things like switch (**x, y**) to switch 2 variables and you **cannot** use comparisons like:

```
switch (x)
{
case < 40: {}
case > 40: {}
}
```

As things like this do not work you will need to use a normal if statement instead.

There is another thing to note when using switch statements, which is when you want multiple cases to perform the same action. It can now be done easily like so:

```
switch (value)
{
case 4: case 5: case 9: {show_message("1st action"); break;}
case 2: {show_message("2nd action");}
}
```

This will show "1st action" when value is equal to 4,5 or 9 and show "2nd action" when value is equal to 2. It is the equivalent of using the following if statement set:

```
if (value == 4 || value == 5 || value == 9)
{
show_message("1st action");
}
else if (value == 2)
{
show_message("2nd action");
}
```

-Expressions

This part is more advanced and you are unlikely to need to know this at a beginners level so you can skip straight to the 'Uses Of Variables' section if you cannot follow it.

In the manual you might have seen that if statements are actually explained to have the form:

```
if (expression is true)
{
perform statement
}
```

Well what we have been dealing with are referred to as expressions, an expression is a combination of values, variables, operators, or functions that are calculated to a value. For example when you use:

```
x < 100
```

What GM actually does is it reads that expression and returns a value of 1 (if the expression is true) or 0 (if the expression is false), effectively that expression is just equivalent to the value of 1 or 0 depending on whenever it is true or not. You can even assign an expression to variable, like so:

```
less_than_hundred = (x < 100);
```

The variable **less_than_hundred** will then equal 1 or 0 depending on whether the expression is true or not because like I said the expression x < 100 when ran by GM is effectively just the evaluated value (in fact **all** assignment values are referred to as expressions as we will come onto in a bit). Now how if statements work is they evaluate the *entire* expression within the brackets after it, and then check whether the expression is true or false. If it is true then it executes the code inside the first braces, if it is false it executes the code inside the else braces. General code, like assignments is referred to as a statement in case you are wondering what it means in the manual by this.

So what GM is actually looking at is this:

```
if (entire expression) is true
{
perform statements
}
```

That is basically it.

But just to add to the confusion, there is another thing that GM has to deal with... Expressions do not always have to return a Boolean true or false value. For example (x == 0)+7 is an expression, it will equal 7 or 8 depending on whether x == 0 or not. Even 4+2 is an expression it will equal 6, even a value just of its own is an expression. As I mentioned earlier all assignment are just calculated expression, now when assigning these expression to a variable it make sense for them to be able to be non Boolean values but when using it as a condition in an if statement they often don't really make any sense. For example:

```
if ((x == 0)+7)
```

or

```
if (4+2)
```

Neither of these you would have any practical use, however GM still needs to be able to handle them if they are used. It does actually says in the manual how GM handles things:

> If the (rounded) value is <=0 (false) the statement after else is executed, otherwise (true) the other statement is executed

This means (note the *rounded* bit in there) that if an expression in an if statement is evaluated as < 0.5 it is considered false and if it is evaluated as >= 0.5 it is considered true. You may actually occasionally see this functionality exploited by people to shorten code, for example in this code:

```
col_inst = collision_line(x1,y1,x2,y2,obj,prec,notme);
if (col_inst)
{
move_towards_point(col_inst.x, col_inst.y, 4);
}
```

If you check the manual you will see that the **collision_line** function does not actually return a Boolean true or false, it in fact returns the instance id of an instance if one is found, if one is not found it returns a negative number, therefore the **col_inst** variable is actually being set to a value like 100102 or -4. The reason this code still works is because when it finds a an instance it will return an instance id which is a large positive number (therefore **col_inst** will be evaluated as true since it will be >= 0.5) and when it does not find an instance it will return a negative value (therefore **col_inst** will be evaluated as false since it will be < 0.5). The non shortened code should actually look like this:

```
col_inst = collision_line(x1,y1,x2,y2,obj,prec,notme);
if (col_inst > 0)
{
move_towards_point(col_inst.x, col_inst.y, 4);
}
```

It is technically bad notation to drop the > 0 comparison and exploit how GM handles non Boolean expressions to make the code shorter, however it is often done by people when using functions which return an instance id and in some other scenarios also, so try to watch out for people doing it in case it causes you any confusion.

Some uses of variables

Variables are going to be useful in most things you want to do however I will show some common usages.

-Simple values

Say you want to keep track of how much ammo in a gun you have. The way to do this is by creating an ammo variable yourself. So first you want to initialize the variable, this will probably be done in the creation event like so:

```
ammo = 100;
```

This will set the variable **ammo** equal to 100 when the instance is created. From here you can do whatever you wish with the variable, for example you could draw the value of the **ammo** using:

```
draw_text(x,y,string(ammo));
```

NOTE: here I have used the string() function, this function turns a real number into a string value. This is done because game maker draws values from strings not real.

You can also decrease the ammo in a firing event, for example:

```
ammo -= 1;
```

TIP

*ammo -= 1; is a little coding shorthand which means the same as ammo = ammo-1; it is the same as ticking the relative box in DnD. The same can also be done with ammo += 5; to add 5 to the ammo. In the same way you can also use /= to divide a variable by a value and *= to multiply a variable by a value.*

While I am on this subject I would also like to mention another trick of multiplying string values, doing these results in a string repeating the given number of times. For example:

```
draw_text(x,y,3 * 'car');
```

Note: the number value must be used before the string value.

This will actually draw the text "carcarcar" (as it results in car written 3 times). Doing this for example can be useful when wanting to blank out a password:

```
draw_text(x,y,password_length * "*");
```

-Storing the value of functions

A lot of functions return values; you will often want these values to be stored in variables. For example with the **instance_create** function it returns the id of the instance it has just created, you could then assign this id to a variable like so:

```
obj = instance_create(x,y,obj_enemy);
```

The variable **obj** will now be equal to the id of the instance created and you can now use this to do anything you wish with. You could for example use this to move or destroy the particular instance for example.

-On/off switches

On/off switches are often needed for many things. For on/off switches you only want to be giving a variable two values either 0 for off or 1 for on. In game maker there are also built-in constants for this as well which are **false** (the same as 0) and **true** (the same as 1), a variable like this that can only be true/false is called a Boolean variable. I will give some examples where on/off switches can be useful. Say you want to show some text only for a limited number of time you could make a variable **show_text** like:

creation event

```
show_text = true;
```

Then in the draw event if you use:

```
if (show_text)
{
draw_text(x,y,"text");
}
```

This code will only draw the text if the variable **show_text** is true (a value of 1). So now if you set the variable **show_text** = false at any point the text will no longer be shown. For this example you could use an alarm. So if you add in the creation event:

```
show_text = true;
alarm[0] = 120;
```

then in an alarm 0 event:

```
show_text = false;
```

This will set an alarm to go off after 120 steps, after this happens the **show_text** variable will be set to false and the text will no longer be drawn.

A similar technique can be used when wishing for a collision event to only occur once, for example:

collision event:

```
if !(hit)
{
//do collision
hit = true;
}
```

This code only does the collision if the variable hit is false... then after it does the collision it sets the variable **hit** to true so it doesn't happen again. The symbol **!** in GML mean **not**, so that if statement is checking whether the instance is **not** hit and only doing the collision code if so. A small problem arises in this example that once it collides and you move away from the collision you can no longer collide with the object again as the hit variable is still set to true. So to get around this you can use this code in the step event:

```
if !(place_meeting(x,y,hitobject))
{
hit = false;
}
```

where you replace **hitobject** with the name of the object which the above code is hitting. What this does is it sets the variable hit back equal to false when the object is no longer in collision with the object.

There are many other uses for on/off switches, dragging and dropping an object is another common example.

-Toggling a variable

This is very commonly needed when dealing with Boolean variables; you often want to switch things on and off like showing text for example. A common **incorrect** way of doing this is by doing the following:

```
if (show_text = true)
{
show_text = false;
}
if (show_text = false)
{
show_text = true;
}
```

You can see why this doesn't work if you follow through the code:

- If **show_text** starts of false: after the first if statement the variable isn't changed so is still false (as the if statement isn't met) then after the second if statement the variable is changed to true (as the if statement is met)
- If **show_text** starts of true: after the first if statement the variable is changed so is still false (as the if statement is met) then after the second if statement the variable is changed again back to true (as the if statement is now met due to the variable being changed before)

  So as you can see the **show_tex**t variable always winds up true.

Now a way to overcome this is to use the following:

```
if (show_text)
{
show_text = false;
}
else
{
show_text = true;
}
```

This works because the second if statement is now not executed when **show_text** was originally true because the else statement will not be met when this is the case.

Although this code works perfectly fine there is actually a quicker, more efficient way of doing this by using the following:

```
show_text = !show_text;
```

You can use this simple piece of code to toggle any Boolean variable, the **!** sign in GML is the **not** sign, when applies to a Boolean variable it makes it the 'opposite' to what it is which is the exactly what we wanted.

With statements

With statements can be used when wanting to call upon another object, you do this by putting an **object index** or **instance id** after the **with** statement. For example:

```
with (obj_player)
{
x = 50;
}
```

If you use this it will set **obj_player's** x value to 50. Whenever you use a **with** statement you are effectively executing code inside the object, this means that if you use a variable you will now be accessing them local to **obj_player** not the object you are using the code in, this is why you are now able to change **obj_player's** x variable. This also means that when you call functions they will be executed locally to the object also, so say you want to destroy the object you would use:

```
with (obj_player)
{
instance_destroy();
}
```

In fact this is the only way of destroying another object as the **instance_destroy** functions doesn't take any arguments and only destroys the instance it is executed under.

There is also another important aspect to the **with** statement. So far I have just shown it used on **obj_player**, presuming there would only be 1 player object in the room, however what happens if you use it on an object which has many instances in the room? Say for example you have several **obj_enemy** instances in the room and you use this code:

```
with (obj_enemy)
{
hspeed = 5;
}
```

What this will do is it will actually set the **hspeed** equal to 5 for all the instances of **obj_enemy** in the room. It does this by cycling through all the instances one at a time, and executing the code you put under every single one of them. This can make the **with** statement very useful but also very slow sometimes if there are lots of instances of the object in the room. You may have tried to do this before using a different code:

```
obj_enemy.hspeed = 5;
```

However this code is completely different. When you call an object index like that it will only set the **hspeed** for the 1st instance of **obj_enemy** in the room... the rest will not be called upon at all.

There are many different scenarios where you may want to use a **with** statement to loop through every instance of an object, you should always keep in your mind the possibility of using one when dealing with instances. For example you can use code which only does things to specific instances, say you want to set the **hspeed** for only red enemies, you could use this:

```
with (obj_enemy)
{
      if (color == "red")
      {
            hspeed = 5;
      }
}
```

So, if using an object index in a **with** statement loops through every instance of the object in the room, how do you call upon a specific instance of an object? Well to do this you just use the instance id. For example if you want to destroy the instance of an enemy you just collided with (remember you use **other** to get the id of the other instance in collision), you could use this in the collision event with **obj_enemy**:

```
with (other)  //other in the collision event is equal to the id of the
instance you have collided with
{
instance_destroy();
}
```

Comments proceeded with a // should always be on a single line, the formatting of this guide makes this one, and many more in the guide on two lines. Be mindful of this if you copy this code!

-Using other and var within a with statement

When you use **other** actually within a **with** statement (note this is completely different from calling the other instance for the with statement which I have just shown) it now applies to the instance the with statement was executed under. You can then use this for example to set a local variable in the original calling instance. For example if you wish to destroy the instance of an enemy furthest to the right you can use this code:

```
greatest_x = -1;
inst_greatest_x = noone;
with (argument0)
{
      if (x > other.greatest_x)
      {
            other.greatest_x = x;
            inst_greatest_x = id;
      }
}
with (inst_greatest_x) {instance_destroy();
}
```

See how you are now checking and setting the **greatest_x** variable from the original instance.

Often with scripts like these however the variables being used are actually only needed locally in the script, in this case use can use the more preferable solution to declaring the variable with var:

```
var greatest_x,inst_greatest_x;
greatest_x = -1;
inst_greatest_x = noone;
with (argument0)
{
      if (x > greatest_x)
      {
            greatest_x = x;
            inst_greatest_x = id;
      }
}
with (inst_greatest_x)  {instance_destroy();
}
```

This works because once you var a variable it is localised to the script not the object, so it doesn't matter what object you call the variable from it will still just be seen as belonging to the script.

For loops

Most of the time you see a for loop in the form:

```
for (i = 0; i < 5; i += 1)
{
      draw_text(10,10+i*8,"hello");
}
```

*-You may wonder what this **i** is?*

Well it is just a variable, like any other variable. i is often used as a variable name because it is short (and was given the letter i from the word iterator), since the variable is rarely needed outside the for loop there is no need to give it a meaningful name, you can though use any variable name you wish.

-How does the for loop work?

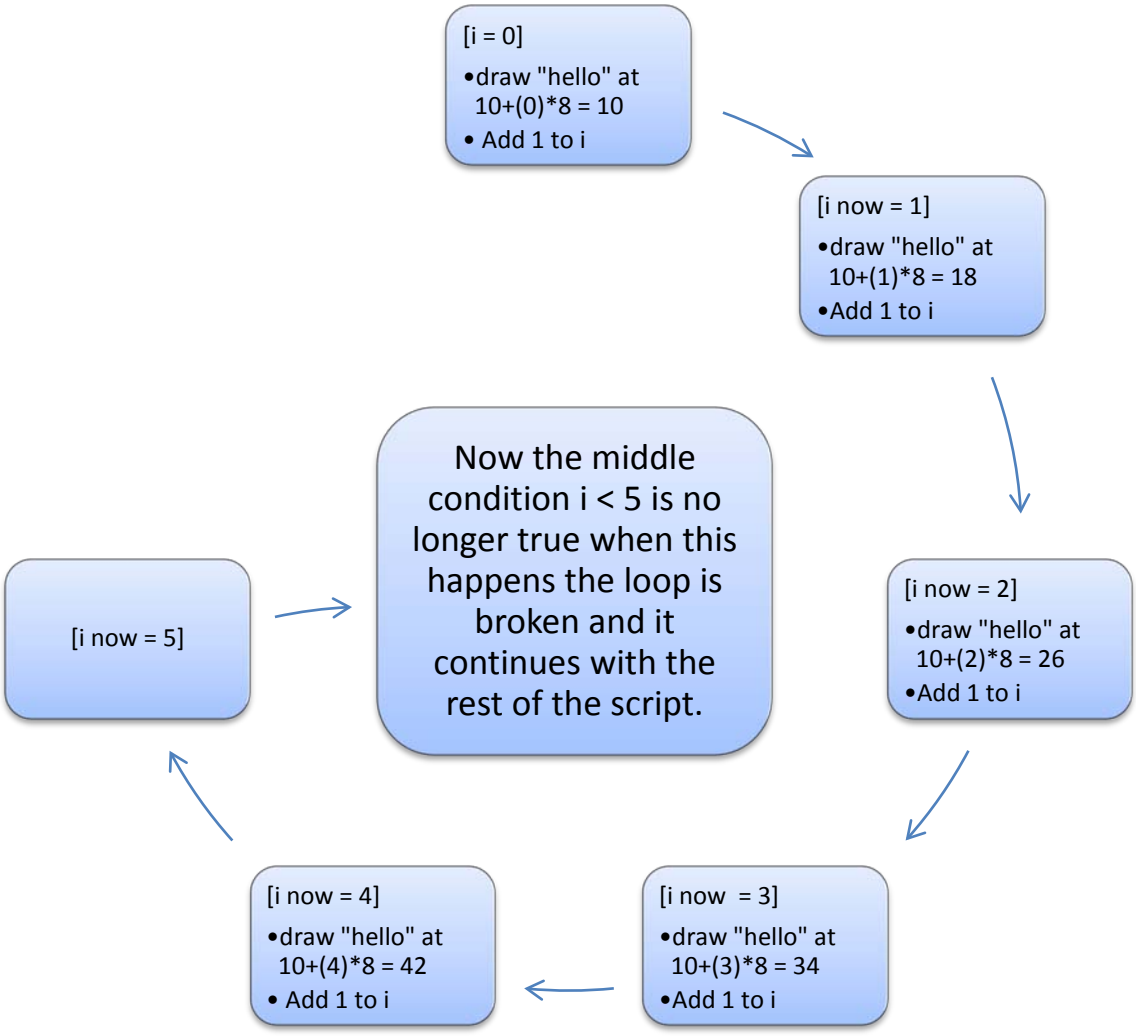
Well at the start it executes the first statement, in this example **i = 0**, so all that will be happening here is the variable i will be set to 0. Next the middle expression is checked. If the middle statement is not correct (false) then; the for loop will be broken completely, the code in the braces will be skipped over and the rest of the code in the event will then be executed. If the middle statement is correct (true) then it executes the code between the braces, in this case it draws text, you can put any code you wish here however. After it has executed the code in the braces it then executes the last statement in the for loop, in this example i += 1 which is just adding 1 to the variable i. Then it just cycles through the whole thing again, missing out the initial i = 0 statement though and starting from the check of if i < 5, eventually this check will always become false, in this case because you are adding 1 to i each time, it will be false after 5 iterations (loops) when i will have added up to 5 (which isn't less than 5).

If you do however put a middle expression in that never becomes false then this will cause the game to crash. This is an infinite loop.

So what will the result of this for loop look like?
Well it will draw the following text:

```
hello
hello
hello
hello
hello
```

Each hello will be draw at x position 10, the top hello text will be draw at y position 10 then all the others will be drawn 8 pixels below each other. To explain it better, I will show exactly what game maker is doing when it reads that code, (I'm only showing the y value it is drawn at, as the x value is obvious):



-Why use a for loop instead of a while loop?

Well it's just preference of notation really, everything you can with a for statement you can also do with a while or a do statement, for loops are generally used when dealing with the need for a changing variable. The previous code though could be written:

```
i = 0;
while (i < 5)
{
      draw_text(10,10+i*8,"hello");
      i += 1;
}
```

Or

```
i = 0;
do
{
      draw_text(10,10+i*8,"hello");
      i += 1;
}
until (i >= 5)
```

Perhaps thinking of it like this can help you understand what exactly is happening in a for statement as well.

Well lots of things, they are very useful. You generally want to be looking to use them when you have to do something repetitive which is in a pattern (imagine drawing 100 dots in a line without using a loop of some kind). They save time coding but also have the major advantage of leaving your code customizable... say in your game you want to draw dots in a line but you want to ask the user how many dots they want to draw. What you can use is:

```
var dots,i;
dots = get_integer("How many dots",3);
for (i = 0; i < dots; i += 1)
{
      draw_sprite(spr_dot,-1,10+i*5,10);
}
```

This will then loop the amount of times equal to the dots variable given by the user. You're going to have great difficulty trying to do this without the use of a loop.

Another common time when for loops are used is when you are dealing with arrays, which we are about to come on to.

Arrays

To explain what arrays are I am going to jump straight into an example. Say you want to draw a message in your game, but you wish for this message to be a randomly chosen from several messages. There are a few ways of doing this, I am going to show you a technique by setting up an array in the creation event like so:

```
message[0] = "that's not funny";
message[1] = "don't look down!";
message[2] = "stop sleeping on the job";
message[3] = "play nice";
message[4] = "how can you see with those glasses on?";
message[5] = "you should probably turn around";
```

You should hopefully be able to see from this that an array is like a 'list' which is a good way to think of them. Say now if you use this in the draw event:

```
draw_text(x,y,message[3]);
```

This is now going to draw the text "play nice" as that is what **message[3]** was set to. Another way to do this is by using a variable like so:

```
var message_choice;
message_choice = 3;
draw_text(x,y,message[message_choice]);
```

Here is where the usefulness of arrays starts to come in. You should be able to see now that you change the value of the variable **message_choice** from 0 to 5 and it will change the corresponding message which is shown. Using this you can set up the variable **message_choice** to be given a random number from 0-5 like so:

```
message_choice = floor(random(6));
```

note **random(6)** will return a random number from 0-6 excluding the number 6 itself. **floor()** will then round this number down to the nearest integer (another word for whole number) this is used because the **random()** function gives numbers with decimals in such as 4.1938 this is obviously not wanted. The result will be a random integer value either 0,1,2,3,4,5

Applying all the code together now you get:

```
var message_choice;
message_choice = floor(random(6));
draw_text(x,y,message[message_choice]);
```

This will now draw a randomly chosen message of the 6 messages you wrote in the creation event.

-Using for loops with arrays

Using for loops is a common way of getting the advantages of an array system. When you set things up using arrays instead of variables you can loop through a value to call upon every 'item' in a list. For example with the previous example, say you now wish to draw all the messages at once, this can be done very easily using a for loop like so:

```
var i;
for (i = 0; i < 6; i += 1)
{
      draw_text(x,y+i*15,message[i]);
}
```

What this is doing is it is looping through every value of i and thus looping through every message to draw. You should hopefully be able to see how much easier this is to now to do than if you had used variables.

-2D Arrays

I have so far only been showing you examples of 1D arrays, you can however also use 2D arrays. These are a bit more difficult to understand but I will again jump straight into an example to show what these are. Say you want to draw a made tic-tac-toe game. You could set up a 2D array to do this like so:

```
square[0,0] = "X";
square[0,1] = "0";
square[0,2] = "X";
square[1,0] = "X";
square[1,1] = "X";
square[1,2] = "0";
square[2,0] = "0";
square[2,1] = "X";
square[2,2] = "0";
```

What this is doing is it is creating like a board or a grid. Where the array gives reference to the marker that is in each square. It may be easier to see what's happening if you reorganize the code like this:

```
square[0,0] = "X"; square[0,1] = "0"; square[0,2] = "X";
square[1,0] = "X"; square[1,1] = "X"; square[1,2] = "0";
square[2,0] = "0"; square[2,1] = "X"; square[2,2] = "0";
```

You can see now hopefully see the created grid easier. From this grid you can now change the value of a square easily, for example if I was to put:

```
square[1,1] = "0";
```

It would change the middle square from a "X" to a "0".

You can now also draw the board using a double for loop like so:

```
var i,j;
for (i = 0; i < 3; i += 1)
{
      for (j = 0; j < 3; j += 1)
      {
            draw_text(10+j*10,10+i*10,square[i,j]);
      }
}
```

This may prove difficult for you to understand, if it does I advise messing around with the code a bit and seeing what happens hopefully by doing this you should what the code is doing.

The most important thing though in setting the board up in an array is now it makes searching for whether there is a winner or not much easier. The reason is you can use for loops to loop through all the rows and columns from the array indexes. I will show you some an example of some code you could use to check for a winner:

```
var i,j,count,val;
winner = "noone";
for (i = 0; i < 3; i += 1)
{
      count = 0;
      for (j = 0; j < 3; j += 1)
      {
            if (square[i,j] == " ") {val = -4;}
            if (square[i,j] == "X") {val = 1;}
            if (square[i,j] == "0") {val = 0;}
            count += val;
      }
      if (count == 3) {winner = "X's";}
      if (count == 0) {winner = "0's";}
}
```

This code is just checking the horizontal rows. It does it by converting the text into numbers -4 for a blank square, 1 for a "X" square and 0 for a "0" square. Then adding up all the values in the row; if a row adds up to 3 then the X's must of won, if a row adds up to 0 then the 0's must of won. You could do similar checks for the vertical columns and a slightly different check for the diagonals.

There a several uses for 2D arrays. A common use is when wanting to something like an inventory/weapon system. An example of an inventory set-up could be (this would be put in the creation event on an object, often a controller object):

```
spell[0,0] = "Dragon's Eye";  //spell name
spell[0,1] = 5;               //spell power
spell[0,2] = 430;             //spell cost
spell[0,3] = 30;              //spell time

spell[1,0] = "Moonstone";
spell[1,1] = 20;
spell[1,2] = 600;
spell[1,3] = 25;

spell[2,0] = "Fae Dust";
spell[2,1] = 3;
spell[2,2] = 210;
spell[2,3] = 5;
```

The way this is set-up is so the 1st index corresponds to the spell type and the 2nd index corresponds to a specific characteristic value. Setting up a system like this has massive advantages in terms of flexibility, it's a lot neater/easier to set-up and it is also easier to read. An example of code you can now use to cast a spell would be:

```
other.health -= spell[current_spell,1];  //setting health off other
instance depending on the spell power
money -= spell[current_spell,2];  //subtracting money depending on
spell cost
alarm[0] = spell[current_spell,3];  //setting alarm to end spell
depending on spell time
```

What this does is it now calls upon the different spell characteristic values set in the 2D array depending on what a **current_spell** variable is set to. By using this you can now easily change between spells just by changing the **current_spell** variable. For example if **current_spell** is set to 1 it will now be executing this code:

```
other.health  -=  20;
money         -=  600;
alarm[0]       =  25;
```

As these are the characteristic values you have set in the 2D array for the Moonstone spell. The spell name isn't being used here it would often be used separately in the draw event by using something like:

```
draw_text(x,y,"Spell Name: "+spell[current_spell,0]);
```

-Easy array initialization setup

Setting up an array can be a messy and monotonous task if done incorrectly. I will show you good technique of setting up an array:

```
inv_num = 0;
inv_item[inv_num] = "item 1"; inv_num += 1;
inv_item[inv_num] = "item 2"; inv_num += 1;
inv_item[inv_num] = "item 3"; inv_num += 1;
inv_item[inv_num] = "item 4"; inv_num += 1;
inv_item[inv_num] = "item 5"; inv_num += 1;
```

Using this will result the same as the following array assignment:

```
inv_item[0] = "item 1";
inv_item[1] = "item 2";
inv_item[2] = "item 3";
inv_item[3] = "item 4";
inv_item[4] = "item 5";
```

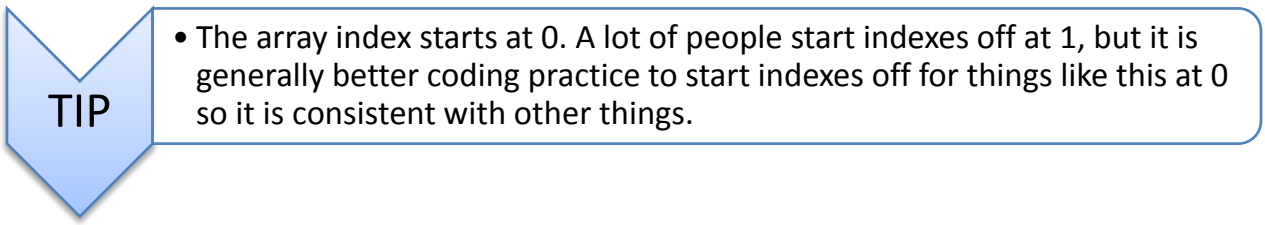
Using the 1st method over the 2nd has several advantages. Firstly the **inv_num** variable is calculated automatically, saving you counting and putting it in manually. Secondly it's more customizable, if you wish to insert an inventory line using the 2nd method you will need to edit all the array indexes, using the 1st method it can just be inserted straight off. Thirdly it takes less time for you to write, once you have set up the line:

```
inv_item[inv_num] = "item 2"; inv_num += 1;
```

You can then just copy/paste the line down, leaving:

```
inv_item[inv_num] = "item 2"; inv_num += 1;
inv_item[inv_num] = "item 2"; inv_num += 1;
inv_item[inv_num] = "item 2"; inv_num += 1;
inv_item[inv_num] = "item 2"; inv_num += 1;
```

Then all you have to do is edit the assignment value for each array, whereas with the 2nd method you need to change the index for each array as well.



If you want to use this set-up with a 2D array, you could do so like this:

```
x_ind = 0; y_ind = 0;

inv_item[x_ind, y_ind] = "item 1"; y_ind += 1;
inv_item[x_ind, y_ind] = "item 2"; y_ind += 1;
inv_item[x_ind, y_ind] = "item 3"; y_ind += 1;
x_ind += 1; y_ind = 0;

inv_item[x_ind, y_ind] = "item 4"; y_ind += 1;
inv_item[x_ind, y_ind] = "item 5"; y_ind += 1;
inv_item[x_ind, y_ind] = "item 6"; y_ind += 1;
x_ind += 1; y_ind = 0;

inv_item[x_ind, y_ind] = "item 7"; y_ind += 1;
inv_item[x_ind, y_ind] = "item 8"; y_ind += 1;
inv_item[x_ind, y_ind] = "item 9"; y_ind += 1;
x_ind += 1;
```

Scripts

A way to think of a script is just like the DnD 'execute a piece of code' action except with a few more possibilities, another way to think of them is just like a function which you are writing yourself. To create a new script just add a script and then give it a name. To use it you can then either use the 'Execute Script' DnD action or you can call it by its name in another script (or the execute a piece of code' action) followed by (). For example if you make a script and call it **set_gravity**, then to call this script you can just use the following code:

```
set_gravity();
```

- So what are the advantages of scripts?

Well for starters doing things in scripts can be easier because it saves you trailing through your objects all the time, it also helps that you can minimize a script window whereas game maker doesn't allow you to minimize the normal execute code action.

Another advantage is the ability to call it over and over again instead of just once like the execute code action, this saves you repeatedly writing out the same piece of code. When you call a script the code is executed inside the instance that calls it, therefore like in the example I'm about to show you, you can call upon and manipulate local variables to that object and all variables assignments will also be made to the calling object.

But the main advantage of using scripts is the ability to use arguments and to get a return value from them.

-Arguments

These are like values that you are giving to the script, to use them you have to put a value in-between the brackets with commas between them, for example:

```
set_gravity(0.4, 270);
```

This will then pass the values 0.4 and 270 to the script **set_gravity**. These values are stored in what are referred to as **arguments**, in this case **argument0** (arguments start from index 0) is equal to 0.4 and **argument1** is equal to 270. So for example to call these values in the script you can use:

```
//set_gravity(gravity amount, gravity direction) sets the gravity
gravity = argument0;
gravity_direction = argument1;
```

This is then setting the gravity to that of argument0 (which we have put as 0.4) and the **gravity_direction** to that of **argument1** (which we have set to 270). The first line of that is just a comment, it is often a good idea to comment any scripts you use saying when each argument should be and what the script does.

Now you have set this script up you can call it with any arguments you want, for example:

```
set_gravity(3,90);
```

This will now set the gravity to 3 upwards.

There is another thing to note and that is that arguments can also be used as an array. For example the previous script could actually have been written like:

```
//set_gravity(gravity amount, gravity direction) sets the gravity
gravity = argument[0];
gravity_direction = argument[1];
```

Which would do exactly same thing, it is sometimes useful to be able to call the arguments as an array, for example when you wish to loop through them.

Something else to be aware of is that you can only give a script a **maximum of 16 arguments**. Also Game Maker will automatically assign all 16 arguments a value even if you don't put one in yourself, a value of 0 is given to all arguments that you don't assign at the end of calling a script. This is why (and you should take note of this) that calling a script is actually quite slow in comparison to executing code inside an object normally, it is therefore not advised to use scripts for very simple things.

-Return

Instead of values you put into a script this is a actually a value that the script gives out to the instance calling it. For example if you set up a script called multiply:

```
//multiply() multiplies numbers
var sum;
sum = 4*5;
return sum;
```

NOTE: I have declared the sum variable using **var**, this practice should always be made with variables used only in the script.

This will calculate the sum as 20 and then it will return the value, so basically this script is going to return 20. Now if you call this from an object using a variable assignment like so:

```
value = multiply();
```

It is going to first execute the script which is going to return 20, this return value will then be assigned to the variable value. So now the variable value will be equal to 20.

You may of seen something similar to this happening with **instance_create()**… for example you can use:

```
obj = instance_create(x,y,obj_enemy);
```

instance_create() is just a built-in function, which acts exactly like a script. You give it arguments, in this case argument0 is x, arugment1 is y, argument2 is the object. Then after game maker has executed it, it gives you a return value, which is the id of the object it has created. This return value is now stored in the variable **obj** for you to use. Note though that not all scripts need to return a value, it is your choice, and also it is only possible to return 1 value for a script. For example if you want to calculate the x and y differences of 2 objects and return the values from a script you might try using:

```
//get_coord_differences(obj1,obj2) calculates x and y differences of
the 2 objects given

return argument0.x-argument1.x;
return argument0.y-argument1.y;
```

This however will not work. The reason being as soon as a script returns a value the script is exited and all the code after it is not run, therefore it will only return the differences in the x values, anyway even if it returned both values how would you be able to use both values at the same time? There are many ways you get around this problem I will show you 2. The first way is probably the 'purist' way to solve this problem which is to put the values into a **ds_list** then return the list id. For example:

```
//get_coord_differences(obj1,obj2,ds_list index) calculates x and y
differences of the 2 objects and adds them to the given ds_list

ds_list_add(argument2,argument0.x-argument1.x);
ds_list_add(argument2,argument0.y-argument1.y);
```

This now puts the values into a given **ds_list** which should be created before executing the script, you can then call the values from the **ds_list**, an example of doing so:

```
var return_list
return_list = ds_list_create();
get_coord_differences(id,other.id,return_list);
x_dif = ds_list_find_value(return_list,0);
y_dif = ds_list_find_value(return_list,1);
ds_list_destroy(return_list);
```

The second way is to create variables in the script and then call from these variables, for example:

```
//get_coord_differences(obj1,obj2) calculates x and y differences of
the 2 objects and assigns them to variables to use

return0 = argument0.x-argument1.x;
return1 = argument0.y-argument1.y;
```

Then an example of calling the script:

```
get_coord_differences(id,other.id);
x_dif = return0;
y_dif = return1;
```

This method is faster than using a **ds_list** however it involves making new variables so is slightly worse for memory; you can decide yourself which method you prefer.

Final notes

If you have managed to make it this far then well done, it is a very long guide so you may have to read back over some of it to recap. What you can learn by just reading is often limited I advise that you try out and mess with some of the concepts I have just been teaching you, experimentation in programming is a key part of understanding.

If you have any questions at this point then I advise making a new topic in the Novice & Intermediate Q&A forum as it will be a lot easier for people to help you there.

Good luck.

Appendix A – Syntax Highlighting

When using the Game Maker code editor, you will observe that as you type, the text will change colors, and some will become bold. This is known as syntax highlighting and it can help you greatly while you are programming. If your intention is to call a resource such as a sprite, object, or script then you can catch a mistake should you happen to call it the wrong name in your code because the color will not change. On the other hand, if you accidently try to create a variable that may already be reserved then you can also catch that mistake because it **will** change color. For example, let's say you are creating an enemy object, maybe a boss, and it needs a life meter. This will of course, require a variable to hold the amount of life you enemy holds, so you throw in the code editor:

```
health = 100;
```

Seems like a good name for a variable that holds the amount of health something has, but wait! It turned blue… That is because **health** is a built in variable that ties to the DnD health bar action. Here is a list of the syntax highlighting you will see while using Game Maker.

Normal Text
Keywords
Comment
Object Name
Room Name
Sprite Name
Sound Name
Background Name
Script Name
Local Variable
Global Variable
Functions
Constants
Paths
Fonts
Timelines

*Note that these are the default colors, these colors can be changed by going to File>Preferences> and clicking the colors tab.

Appendix B – Built In Variables

This is a list of all the built in variables in Game Maker

Global	Global (Cont)	Local
global: argument	global: event_type	local: alarm
global: argument0	global: fps	local: bbox_bottom
global: argument1	global: game_id	local: bbox_left
global: argument10	global: health	local: bbox_right
global: argument11	global: instance_count	local: bbox_top
global: argument12	global: instance_id	local: depth
global: argument13	global: keyboard_key	local: direction
global: argument14	global: keyboard_lastchar	local: friction
global: argument15	global: keyboard_lastkey	local: gravity
global: argument2	global: keyboard_string	local: gravity_direction
global: argument3	global: lives	local: hspeed
global: argument4	global: mouse_button	local: id
global: argument5	global: mouse_lastbutton	local: image_alpha
global: argument6	global: mouse_x	local: image_angle
global: argument7	global: mouse_y	local: image_blend
global: argument8	global: program_directory	local: image_index
global: argument9	global: room	local: image_number
global: argument_relative	global: room_caption	local: image_single
global: background_alpha	global: room_first	local: image_speed
global: background_blend	global: room_height	local: image_xscale
global: background_color	global: room_last	local: image_yscale
global: background_foreground	global: room_persistent	local: mask_index
global: background_height	global: room_speed	local: object_index
global: background_hspeed	global: room_width	local: path_endaction
global: background_htiled	global: score	local: path_index
global: background_index	global: secure_mode	local: path_orientation
global: background_showcolor	global: show_health	local: path_position
global: background_visible	global: show_lives	local: path_positionprevious
global: background_vspeed	global: show_score	local: path_scale
global: background_vtiled	global: temp_directory	local: path_speed
global: background_width	global: transition_kind	local: persistent
global: background_x	global: transition_steps	local: solid
global: background_xscale	global: view_angle	local: speed
global: background_y	global: view_current	local: sprite_height
global: background_yscale	global: view_enabled	local: sprite_index
global: caption_health	global: view_hborder	local: sprite_width
global: caption_lives	global: view_hport	local: sprite_xoffset
global: caption_score	global: view_hspeed	local: sprite_yoffset
global: current_day	global: view_hview	local: timeline_index
global: current_hour	global: view_object	local: timeline_position
global: current_minute	global: view_vborder	local: timeline_speed
global: current_month	global: view_visible	local: visible
global: current_second	global: view_vspeed	local: vspeed
global: current_time	global: view_wport	local: x
global: current_weekday	global: view_wview	local: xprevious
global: current_year	global: view_xport	local: xstart
global: cursor_sprite	global: view_xview	local: y
global: error_last	global: view_yport	local: yprevious
global: error_occurred	global: view_yview	local: ystart
global: event_action	global: working_directory	
global: event_number		
global: event_object		

Credits

flexaplex - Writhing the original version of this guide and posting it on the GMC. The original topic can be found HERE.

Destron – The PDF version of this guide (what your reading now), minor editing and spelling correction, added a couple small bits here and there.

The GMC for comments made on the guide

Mark Overmars for creating Game Maker

YoYo Games for continuing Game Maker development.

Where to get help

Stumped? Don't quite understand something? With Game Makers massive user base there is always help available!

The original GMC thread can be found HERE
We are always willing to help readers at GM@DM, you can contact us HERE
Fellow users of the GMC always love to lend a hand, you can also try posting your problem HERE**

*Please do not assume that the original author will still support this guide, or have the time to offer help. Ask if they have time first, if not, seek another method. It's quite possible, that you could be reading this long after it was published and the author has moved on.

** When posting on the GMC forums, the best way to get a quick and helpful response is to give your post a meaningful subject, post as much detail as possible including your code, and if possible post what you have tried to fix the problem yourself. The GMC loves to help people who are stuck, but are always more responsive to those who try to help themselves first!