

# Chapter 1

## Basics on Plugin Creation

**By Goinza**

This guide will help you with the basics of how to create a plugin for SRPG Studio.

### Requirements

To use this guide, you will need these two things:

- Basic knowledge about programming: this software uses JavaScript, so ideally you need to know how to use that language. But if you have knowledge about other languages, you can adapt to it, specially if you already used other object-oriented languages.
- A text editor: you can use a simple editor like Notepad++, or you can use a more advanced one like Visual Studio Code. The selection is up to personal preference. But avoid using the regular Notepad, as that editor lacks some features that are important to write code.

### Creating the Plugin

Here we will explain the basics of how to create a plugin, explaining the different methods available and which ones you should use and which ones you should not use.

#### Method #1 - Editing the original scripts

This method is NOT recommended, but it is the easier to use and to explain. You can use this to make quick changes if you don't want to make a new file. Keep in mind that any change to the code can be easily restored to default using an option in the editor. You need to go to Help→Restore Data→Script Folder and hit Ok.

Every project has a folder called Script, where all the JavaScript files reside. If you modify one of those files, you can change how the game works. But this is not the right way to create a plugin: those files will reset to default each time the software updates, and even if that wasn't the case, it would be difficult to keep track of each change.

For this reason, every change should have its own separate file. This is why the Plugin folder exists. That is the folder where you can put every custom plugin.

#### Method #2 – Creating new files

This is the recommended method to make new plugins.

The first step to make a plugin is to create a new file in the Plugin folder of your project. Every custom plugin must be inside that folder, although you can put the files in sub-folders too to organize it better.

It is important that each file must have the .js type. If you want to leave files of other formats in there, like for example a readme.txt file to explain how the plugin works, you need to make sure that the first character of the file's name is '\$'. For example, use \$readme.txt instead of readme.txt.

Also, if you want any plugin "disabled" while being in the Plugin folder, you can do the same as with the readme file, because the engine will ignore every file of the Plugin folder that starts with the '\$' character.

Note that, inside this method, there are two ways to change the functions of the engine. You can overwrite a function or use the call function to make it use both the default and modified versions of the function. Keep in mind that both methods are valid, and can be used for different functions inside the same file.

## Method #2.A – Overwriting functions

To use this method, we have to overwrite a function that already exists in the original files, making a new version of said function.

Let's use a small function as an example on how to do this. The next function shown is the function called getHit, and is part of the object in the variable AbilityCalculator. It is located in line 19 of singleton-calculator.js

```
var AbilityCalculator = {  
  
  // ...  
  
  getHit: function(unit, weapon) {  
  
    // Hit rate formula. Weapon hit rate + (Ski * 3)  
  
    return weapon.getHit() + (RealBonus.getSki(unit) * 3);  
  
  },  
  
  // ...  
  
};
```

Before we can modify this function, we need to change how the function is declared. While you can copy and paste the entire object and do modifications like that, it is not recommended. You only need to grab the function that you are going to modify. Here is how it would look if we write the getHit function without any changes to a new file.

```
AbilityCalculator.getHit = function(unit, weapon) {

    return weapon.getHit() + (RealBonus.getSki(unit) * 3);

}
```

There are a couple of changes to how the function is written in order to work:

- First, the function need to be called using the name of the object it belongs. In this case, the object is called AbilityCalculator, so instead of calling it getHit, we will call it AbilityCalculator.getHit.
- The other change, while it may seem very small, is important to make it work. You have to change the ':' between the name of the function and the word function and replace it for a '='.

## Method #2.B – Using the call() function

Some times, instead of completely overwriting the function, you would like to add some additional code without modifying the original script. For example, let's grab the same getHit function, but instead of modifying it, we will use the same formula and add the Luck stat of the unit to the result. If we used the method 2.A, the code would look like this.

```
AbilityCalculator.getHit = function(unit, weapon) {

    return weapon.getHit() + (RealBonus.getSki(unit) * 3) +
        RealBonus.getLuk(unit);

}
```

While this method works, it is not ideally. If you know that you are not going to modify the original code, just add new code to it, then you need to use the call() function. What this does is call the original function and returns the result. After this is done, you save the result in a local variable and do the modifications you need before returning the final result. For example, using the same code as before, using the call function would look like this.

```
var oldHitFunction = AbilityCalculator.getHit;
AbilityCalculator.getHit = function(unit, weapon) {
    var hit = oldHitFunction.call(this, unit, weapon);

    return hit + RealBonus.getLuk(unit);
}
```

As you can see, the original function was stored in a variable that can be called whenever you want in the new version of the function. In this example, it was called at the first line of code, but it can also be called later, depending on what you want to do.

## Advantage of method 2.B over 2.A

When reading the script files, the software will give priority to the files inside the Plugin folder over those in the Script folder, so all custom plugins will overwrite the default files. But if there are two files in the Plugin folder that have the same function, only one of them will work.

A way to partially avoid this clash of plugins is to use the `call()` function, as explained in the method 2.B. For example, we have two plugins, where one overwrites the `getHit` function to add the Luck stat and the other does the same but add Speed instead. In this case there is a clash, so let's say that the software ends up grabbing the function that add the Luck stat. What happened is that the Speed version overwrote the default function, and then the Luck version overwrote the Speed version. That is a problem, so the solution in this case is to make both functions use the call function. This way, the Luck version, when using the call function, will call the Speed version and then the Speed version will call the default one. Note that if one of them doesn't have the call function, this chain of calls will break.

## Warning about the use of method 2.B

If one of your files contains at least one function that uses the `call()` function, you face a potential problem. Each function that will be called needs to be stored into a variable first. So, if one of the variables of those functions has the same name as another variable storing another function, then you risk the problem of calling the wrong function. To avoid this, we wrap all those variables inside a new function, so their reach doesn't go beyond the file and can't be accessed by other plugins. For example, this is how you should use the `getHit` function that was explained before.

```
(function() {  
    var alias1 = AbilityCalculator.getHit;  
    AbilityCalculator.getHit = function(unit, weapon) {  
  
        var hit = alias1.call(this, unit, weapon);  
  
        return hit + RealBonus.getLuk(unit);  
  
    }  
})();
```

Now that this is done, you don't have to worry about the name of the variable, so you can use generic names like the one above: `alias1`, `alias2`, etc.

## Conclusion

If possible, always use the call function to improve the compatibility of your plugin with others made by you or other people.