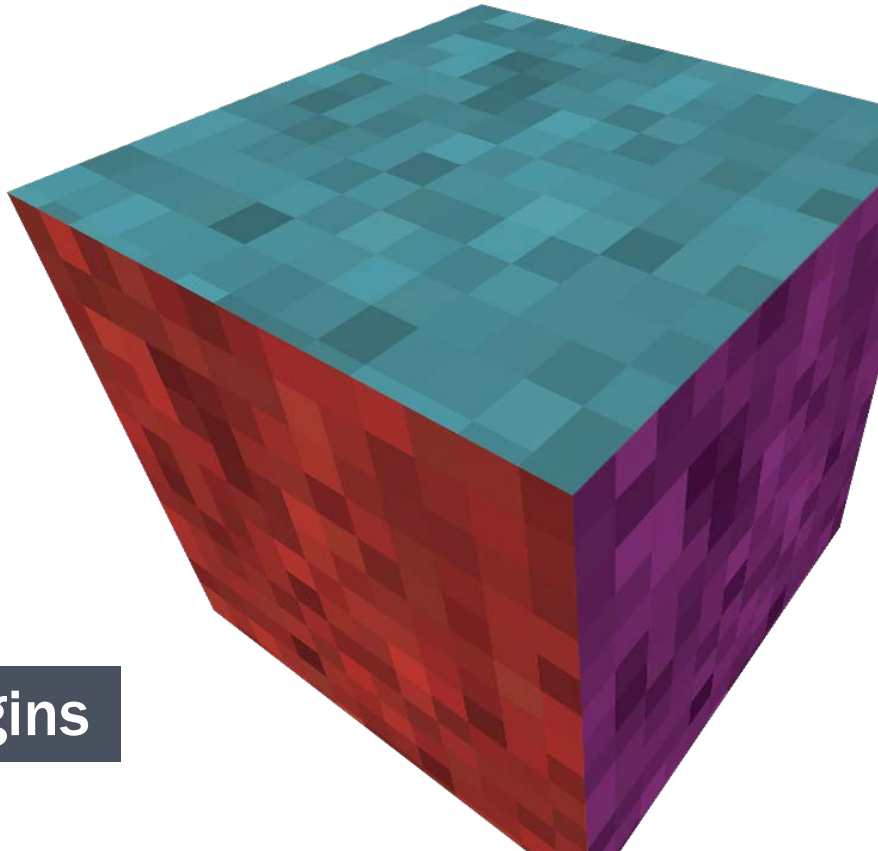
A Beginner's Guide to Writing Minecraft® Plugins in JavaScript

Walter Higgins

A Beginner's Guide to Writing Minecraft® Plugins in JavaScript



Peachpit Press

Walter Higgins

Having explained the basics of how arrays work, Listing 8.1 should be a little easier to understand now. The **`random()`** function rolls a die (whose sides are the length of the array: 4) and then assigns the random number returned to a variable called **`index`**. You then use that index to get an item from the **`greetings`** array. You are effectively plucking a random item from the list of greetings. Try it for yourself by issuing the following commands at the in-game prompt:

```
/js refresh(); // reloads plugins
/js var greetings = require('greetings');
/js greetings.random();
/js greetings.random();
/js greetings.random();
```

You should definitely call **`greetings.random()`** a couple of times to verify it returns a random greeting each time. Remember, you can call up the previous command at the in-game prompt by pressing **`/`** and then pressing the Up arrow key.

So far you've looked only at constructing arrays and getting individual items from arrays. Arrays are powerful, and there are many things you can do with them.

Once you've constructed an array, you can add new items to the end of the array using a function called **`push`**. The **`push()`** function is used like this to add a new item:

```
/js farmAnimals.push('Horse');
```

`'Horse'` is added to the end of the array. After the previous command is executed, the **`farmAnimals`** array would look like this:

```
[0] 'Sheep'
[1] 'Cow'
[2] 'Pig'
[3] 'Chicken'
[4] 'Horse'   <-- New item appended
```

The length of the array would change from 4 to 5. You can check this by issuing the command **`/js farmAnimals.length`**. The **`push()`** function cannot be called on its own. It's a special type of function called a *method*, which means it's a function that belongs to a particular object, so it can be called only using the form **`object.method()`**, with **`object`** in this case being **`farmAnimals`** and **`method()`** being **`push()`**. We'll explore objects more in later chapters.

The **push()** method always *appends* items to the end of the array. If you want to insert an item into the array at a position other than the end, you'll need to use the **splice()** method instead. Here's how you insert a new animal into the **farmAnimals** array at position 2:

```
/js farmAnimals.splice( 2, 0, "Cat" );
```

This is what the array will look like after you run the previous command:

```
[0] 'Sheep'
[1] 'Cow'
[2] 'Cat'     <-- New item inserted
[3] 'Pig'
[4] 'Chicken'
[5] 'Horse'
```

You can see that the new item is inserted at position 2 and that the indexes for all the items after position 2 have changed. **'Pig'** is bumped from index 2 to index 3, **'Chicken'** from index 3 to index 4, and so on. The **splice()** method lets you insert items anywhere in an array. The first parameter is the position you want to insert the items, the second parameter is how many items you want to remove (if you're only inserting items, then leave this as 0), and the third and subsequent parameters are the items you want to insert. You can insert one or more items at a time.

```
/js farmAnimals.splice( 1, 0, "Ocelot", "Wolf" );
```

This is what the array would look like after running the previous command:

```
[0] 'Sheep'
[1] 'Ocelot'  <-- New items inserted
[2] 'Wolf'    <-- New items inserted
[3] 'Cow'
[4] 'Cat'
[5] 'Pig'
[6] 'Chicken'
[7] 'Horse'
```

Now let's say you want to remove some items from an array. The list of farm animals you've constructed so far is starting to look crowded, and there are definitely some animals in that list that shouldn't be there (wolves and farm animals

don't mix). As hinted at previously, the **splice()** method can also be used to *remove* items from the array. Let's start by removing the **Cat** item from the array.

```
/js farmAnimals.splice( 4, 1 );
```

The output from the previous command will be an array of items removed, so in your display, you'll see something like this:

```
[ "Cat" ]
```

That's because the **splice()** method does not return the array it spliced; instead, it returns the items it removed from the array. Remember, the first parameter you pass to **splice()** is the index of the item, and the second parameter is always the number of items you want to remove. If no additional parameters are provided, then **splice()** will only remove items and not insert new items. To see what your **farmAnimals** array looks like now, run the **/js farmAnimals** statement. Your array will look something like this in memory:

```
[0] 'Sheep'
[1] 'Ocelot'
[2] 'Wolf'
[3] 'Cow'
[4] 'Pig'
[5] 'Chicken'
[6] 'Horse'
```

Now let's tidy up the array some more by removing the pesky ocelot and wolf.

```
/js farmAnimals.splice( 1, 2 );
```

The previous statement says "Starting at index 1, remove 2 items." The array will now look like this:

```
[0] 'Sheep'
[3] 'Cow'
[4] 'Pig'
[5] 'Chicken'
[6] 'Horse'
```

TABLE 8.1 lists a couple of other useful array methods.

You can learn more about the **Array** object and its methods and properties at **https://developer.mozilla.org/en-US/docs/Web/Javascript/Reference/Global_Objects/Array**. In later chapters you'll learn how to process all of the items in an array using JavaScript's looping statements.

TABLE 8.1 Array Methods

METHOD NAME	DESCRIPTION
`.push()`	Adds one or more elements to the end of an array and returns the new length of the array.
`.pop()`	Removes and returns the first item in the array.
`.unshift()`	Adds one or more elements to the beginning of an array and returns the new length of the array.
`.shift()`	Removes the first element from an array and returns that element. This method changes the length of the array.
`.reverse()`	Reverses an array in place. The first array element becomes the last, and the last becomes the first.
`.splice()`	Changes the content of an array, adding new elements while removing old elements.
`.slice()`	Returns a shallow copy of a portion of an array into a new array object.
`.join()`	Joins all elements of an array into a string.
`.indexOf()`	Returns the first index at which a given element can be found in the array, or returns -1 if it is not present.
`.sort()`	Sorts the elements of an array in place and returns the array. The default sort order is `Alphabetic`.

First Steps with Events

IMPORTANT

This new **`greetPlayers.js`** module should be saved in the **`scriptcraft/plugins`** folder, not the **`scriptcraft/modules`** folder, because you'll want this module to load automatically when the server starts up.

So, you have a new module **`greetings.js`** with a single function called **`random()`** that returns a random greeting. What you want is for every player who connects to be greeted with a random greeting. Let's dive right in and create a new module called **`greetPlayers.js`**.

Type **LISTING 8.2** into your new **`greetPlayers.js`** file.

Make sure to save your file and then run the JavaScript **`refresh()`** function to reload ScriptCraft using **`/js refresh()`** from the in-game prompt or **`js refresh()`** from the server prompt.

LISTING 8.2 Greeting Players as They Join the Server

```
var greeting = require('greetings');
function greetPlayer( event ) {
  var player = event.player;
  var message = greeting.random() + player.name;
  echo( player, message );
};
events.connect( greetPlayer );
```

Now disconnect from your server and rejoin the server. You should see something like this in your screen when you join the server:

```
Konnichiwa walterh
```

The message will of course be different for you. **FIGURE 8.1** shows where you should expect the greeting to appear when you join the server. The `echo()` function is provided by ScriptCraft as a way to send messages to players. It takes two parameters, the first being the player and the second being the message you want to send to that player.

FIGURE 8.1 Greeting players

ScriptCraft comes bundled with a built-in variable called **events**. The **events** variable is used to listen and react to events in the game. There are approximately 200 different types of events you can respond to in Minecraft. When you *register* for an *event* in your code, you are telling the server that you want to be notified when a particular type of activity occurs in the game. You register by giving the server a function that won't be called immediately but will be called only when the activity occurs. In the Listing 8.2 code, you are basically saying to Minecraft "Hey, whenever someone connects to the game, I want you (the server) to call this **greetPlayer** function."

This is the first time you've seen functions used as parameters to another function call. You call the **events.connect()** function by passing it another function as a parameter. This style of coding—passing functions as parameters to other functions—is called *functional programming*. The important thing to note here is that at no point in this module is the **greetPlayer()** function actually called. All you do is register it using the **events.connect()** function so that it will be called later each time a player connects. The **greetPlayer()** function is called an *event-handling* function because its purpose is to handle events, specifically the event that is fired by the server whenever a player connects to the game.

We'll look more closely at events and event-handling functions in a later chapter.

More on Modules

From looking at the **greetPlayers** module, you can see the first thing it does is load another module, **greetings**, which you created earlier in this chapter. If you remember, the **greetings** module in turn loads yet another module—the **dice** module you worked on in the previous chapters. This is a classic example of how programming is typically done. You start with small dedicated modules and then work on bigger and bigger modules piecing them together to solve a problem. In programming, a big part of problem solving is breaking problems down into smaller and smaller problems, solving each of these smaller problems, and then piecing together the solutions. When a program module relies on another module, we say it *depends* on the module. A module that loads another module *depends* on that module, and that module in turn will *depend* on other modules. All of the modules that are required—either directly or indirectly—by a program are called *dependencies*. You can see in **FIGURE 8.2** the relationship between the **greetPlayers**, **greetings**, and **dice** modules.