Because in JavaScript, it doesn't matter whether it's passed by value or by reference or whatever. What matters is mutation vs assignment of the parameters passed into a function.
OK, let me do my best to explain what I mean. Let's say you have a few objects.

```
var object1 = {};
var object2 = {};
```

What we have done is "assignment"... We've assigned 2 separate empty objects to the variables "object1" and "object2".

Now, let's say that we like object1 better... So, we "assign" a new variable.

```
var favoriteObject = object1;
```

Next, for whatever reason, we decide that we like object 2 better. So, we do a little re-assignment.

```
favoriteObject = object2;
```

Nothing happened to object1 or to object2. We haven't changed any data at all. All we did was re-assign what our favorite object is. It is important to know that object2 and favoriteObject are both assigned to the same object. We can change that object via either of those variables.

```
object2.name = 'Fred';
console.log(favoriteObject.name) // Logs Fred
favoriteObject.name = 'Joe';
console.log(object2.name); // Logs Joe
```

OK, now let's look at primitives like strings for example

```
var string1 = 'Hello world';
var string2 = 'Goodbye world';
```

Again, we pick a favorite.

```
var favoriteString = string1;
```

Both our favoriteString and string1 variables are assigned to 'Hello world'. Now, what if we want to change our favoriteString??? What will happen???

```
favoriteString = 'Hello everyone';
console.log(favoriteString); // Logs 'Hello everyone'
console.log(string1); // Logs 'Hello world'
```

Uh oh.... What has happened. We couldn't change string1 by changing favoriteString... Why?? Because we didn't *change* our string *object*. All we did was "RE ASSIGN" the favoriteString *variable* to a new string. This essentially disconnected it from string1. In the

previous example, when we renamed our object, we didn't assign anything. (Well, not to the *variable itself*, ... we did, however, assign the name property to a new string.) Instead, we mutated the object which keeps the connections between the 2 variables and the underlying objects. (Even if we had wanted to modify or *mutate* the string object *itself*, we couldn't have, because strings are actually immutable in JavaScript.)
Now, on to functions and passing parameters.... When you call a function, and pass a parameter, what you are essentially doing is an "assignment" to a new variable, and it works exactly the same as if you assigned using the equal (=) sign.

Take these examples.

```
var myString = 'hello';

// Assign to a new variable (just like when you pass to a function)
var param1 = myString;
param1 = 'world'; // Re assignment

console.log(myString); // Logs 'hello'
console.log(param1); // Logs 'world'
```
Now, the same thing, but with a function

Function fname()

```
{

}
```

```
function myFunc(param1) {
   param1 = 'world';

   console.log(param1); // Logs 'world'
}

var myString = 'hello';
// Calls myFunc and assigns param1 to myString just like param1 = myString
myFunc(myString);

console.log(myString); // logs 'hello'
```
OK, now let’s give a few examples using objects instead... first, without the function.

```
var myObject = {
   firstName: 'Joe',
  lastName: 'Smith'
};

// Assign to a new variable (just like when you pass to a function)
var otherObj = myObject;

// Let's mutate our object
otherObj.firstName = 'Sue'; // I guess Joe decided to be a girl

console.log(myObject.firstName); // Logs 'Sue'
console.log(otherObj.firstName); // Logs 'Sue'

// Now, let's reassign the variable
otherObj = {
  firstName: 'Jack',
  lastName: 'Frost'
};

// Now, otherObj and myObject are assigned to 2 very different objects
// And mutating one object has no influence on the other
console.log(myObject.firstName); // Logs 'Sue'
console.log(otherObj.firstName); // Logs 'Jack';
```

Now, the same thing, but with a function call

```
function myFunc(otherObj) {

  // Let's mutate our object
  otherObj.firstName = 'Sue';
  console.log(otherObj.firstName); // Logs 'Sue'

  // Now let's re-assign
  otherObj = {
    firstName: 'Jack',
    lastName: 'Frost'
  };
  console.log(otherObj.firstName); // Logs 'Jack'

  // Again, otherObj and myObject are assigned to 2 very different objects
  // And mutating one object doesn't magically mutate the other
}

var myObject = {
  firstName: 'Joe',
  lastName: 'Smith'
};

// Calls myFunc and assigns otherObj to myObject just like otherObj = myObject
myFunc(myObject);

console.log(myObject.firstName); // Logs 'Sue', just like before
```

OK, if you read through this entire post, perhaps you now have a better understanding of how function calls work in JavaScript. It doesn't matter whether something is passed by reference or by value... What matters is assignment vs mutation.

Every time you pass a variable to a function, you are "Assigning" to whatever the name of the parameter variable is, just like if you used the equal (=) sign.

Always remember that the equals sign (=) means assignment. Always remember that passing a parameter to a function *in JavaScript* also means assignment. They are the same and the 2 variables are connected in exactly the same way (which is to say they aren't, unless you count that they are assigned to the same object).
The only time that "modifying a variable" affects a different variable is when the underlying object is mutated (in which case you haven't modified the variable, but the object itself.

There is no point in making a distinction between objects and primitives, because it works the same exact way as if you didn't have a function and just used the equal sign to assign to a new variable.

The only gotcha is when the name of the variable you pass into the function is the same as the name of the function parameter. When this happens, you have to treat the parameter inside the function as if it was a whole new variable private to the function (because it is)

```
function myFunc(myString) {
   // myString is private and does not affect the outer variable
   myString = 'hello';
}

var myString = 'test';
myString = myString; // Does nothing, myString is still 'test';

myFunc(myString);
console.log(myString); // Logs 'test'
```