Title: Understanding pass by value vs pass by reference

Answer:

Numbers, Strings and Booleans

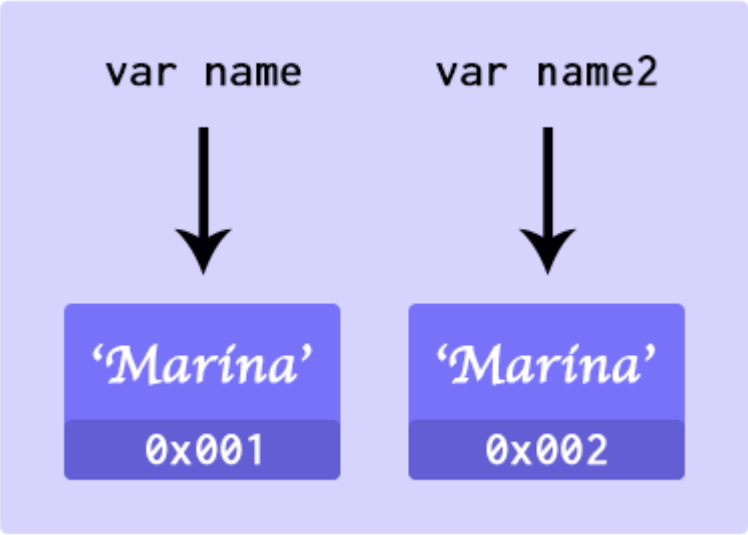
In JavaScript, Primitive Types such as `undefined`, `null`, `string`, `number`, `boolean` and `symbol` are passed by value.

```
let name = 'Marina';
let name2 = name;
```

```
console.log({name, name2});
 >> { name: 'Marina', name2: 'Marina' }
```

```
name = 'Vinicius';
```

```
console.log({name, name2});
>> { name: 'Vinicius', name2: 'Marina' }
```



Passed by value.

When the variable `name` is assigned, a space in the memory with an address of `0x001` is reserved to store that value. The variable `name` then points to that address. The variable `name2` is then set to equal `name`. A new space in the memory, with a new address `0x002` is allocated and stores a copy of the value stored in the address the `name` points to.

So, whenever we want to modify the value of `name`, the value stored by `name2` won't be changed, since its a copy, stored in a different location.

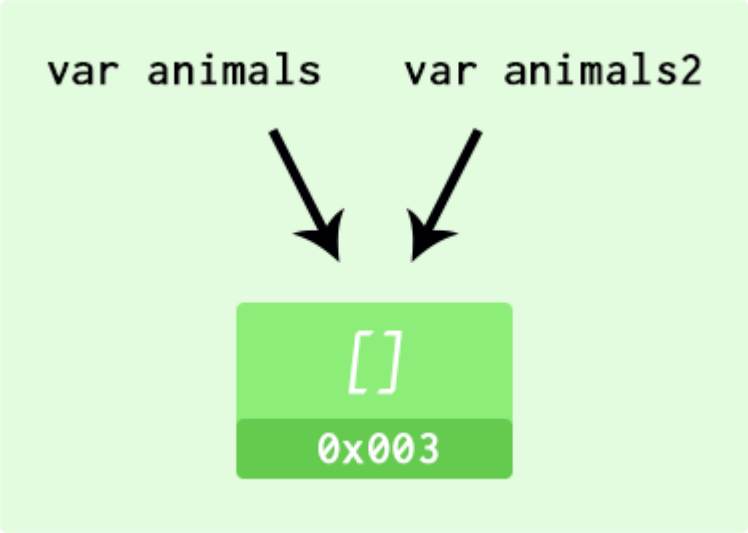
Objects and Arrays

Objects in JavaScript are passed by reference. When more than one variable is set to store either an `object`, `array` or `function`, those variables will point to the same allocated space in the memory.

```
const animals = ['Cat', 'Dog', 'Horse', 'Snake'];
```

```
let animals2 = animals;
console.log({animals, animals2});
>>
{
  animals: ['Cat', 'Dog', 'Horse', 'Snake'],
  animals2: ['Cat', 'Dog', 'Horse', 'Snake']
}
```

```
animals2[3] = 'Wale';
console.log(animals, animals2);
>>
{
  animals: ['Cat', 'Dog', 'Horse', 'Wale'],
  animals2: ['Cat', 'Dog', 'Horse', 'Wale']
}
```



Passed by reference.

When `animals` is set to store an array, memory is allocated and an address is associated to that variable. Then `animals2` is set to equal `animals`. Since `animals` stores an array, instead of creating a copy of that array and a new address in the memory, `animals2` is simply pointed to the same object in the existing address. That way any changes made to `animals2` will reflect on `animals`, because they point to the same location.

You'll see the same behavior for objects:

```
const person = {
  name: 'Marina',
  age: 29
};
```

```
let femme = person;
femme.age = 18;
```

```
console.log({person, femme});
>>
{
  person: { name: 'Marina', age: 18 },
  femme: { name: 'Marina', age: 18 }
}
```

Copying Objects and Arrays

Since a simple assignment is not enough to produce a copy of an object, that can be achieved by other approaches:

Arrays

slice()

```
let animals2 = animals.slice();
animals2[3] = 'Shark';
```

concat()

```
let animals3 = [].concat(animals);
animals3[3] = 'Tiger';
```

spread (ES6)

```
let animals4 = [...animals];
animals4[3] = 'Lion';
```

Changes will affect only the object modified:

```
console.log({animals, animals2, animals3, animals4});
>>
{
  animals: ['Cat', 'Dog', 'Horse', 'Snake'],
  animals2: ['Cat', 'Dog', 'Horse', 'Shark'],
  animals3: ['Cat', 'Dog', 'Horse', 'Tiger'],
  animals4: ['Cat', 'Dog', 'Horse', 'Lion']
}
```

Objects

assign()

```
let human = Object.assign({}, person, { age: 20 });
```

```
console.log(person, human);
>>
{
  person: { name: 'Marina', age: 29 },
  human: { name: 'Marina', age: 20 }
}
```

Deep Clone

It’s important to note that those methods are just one level deep. For deep clones there is a frowned upon method. Use carefully.

```
let femme3 = JSON.parse(JSON.stringify(person));
femme3.name = 'Leslie';
```

```
console.log(person, femme3);
>>
{
  person: { name: 'Marina', age: 29 },
  femme3: { name: 'Leslie', age: 29 }
}
```

Tags: variables, functions / methods, javascript