1.Give a write up on Difference between copy by value and copy by reference.

Armaan.K

Javascript has 5 data types that are passed by ***value***: `Boolean`, `null`, `undefined`, `String`, and `Number`. We'll call these **primitive types**.

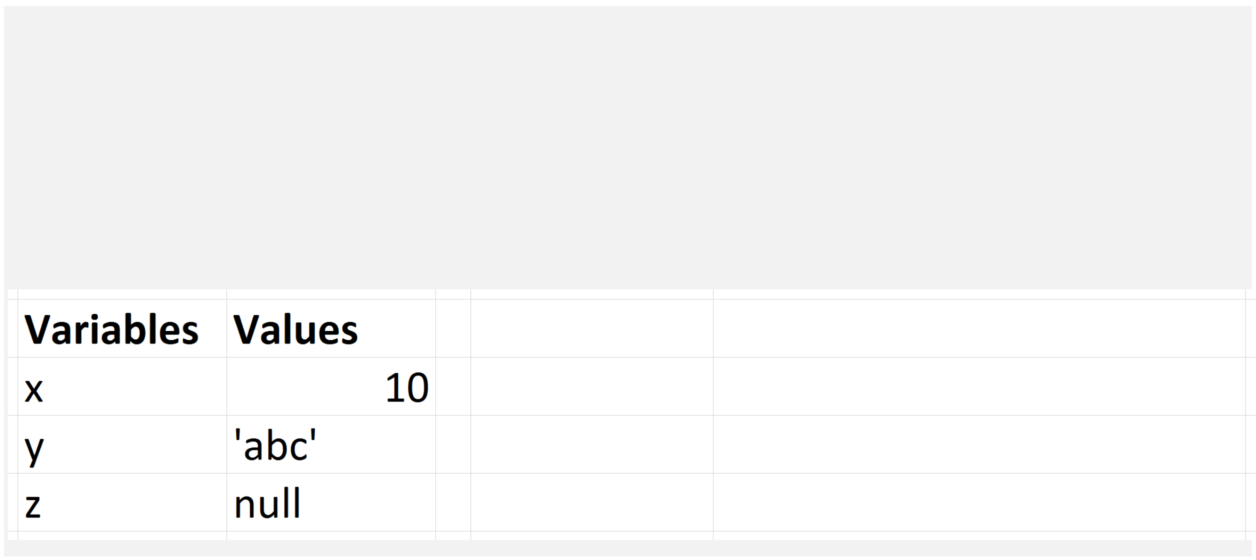
Javascript has 3 data types that are passed by ***reference***: `Array`, `Function`, and `Object`. These are all technically Objects, so we'll refer to them collectively as **Objects**.

Primitives

If a primitive type is assigned to a variable, we can think of that variable as *containing* the primitive value.

```
var x = 10;

var y = 'abc';

var z = null;
```

`x` *contains* `10`. `y` *contains* `'abc'`. To cement this idea, we'll maintain an image of what these variables and their respective values look like in memory.



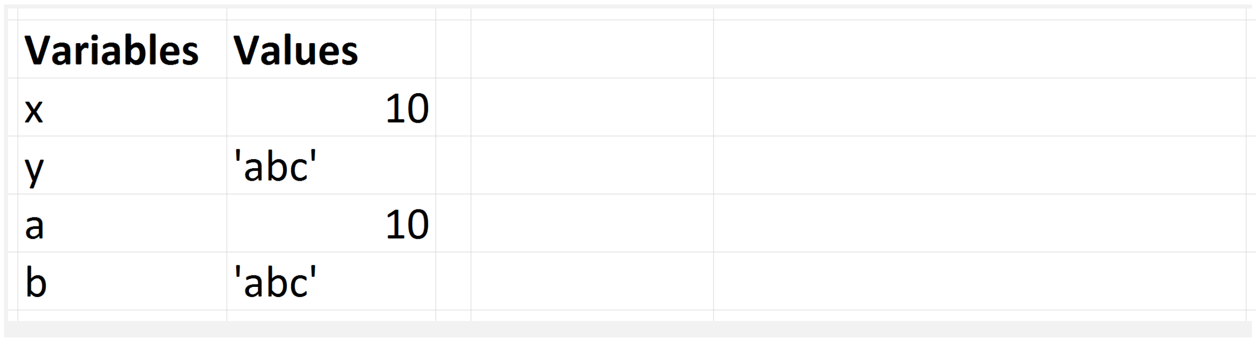
Variables	Values
x	10
y	'abc'
z	null

When we assign these variables to other variables using `=`, we **copy** the value to the new variable. They are copied by value.

```
var x = 10;

var y = 'abc';

var a = x;

var b = y;

console.log(x, y, a, b); // -> 10, 'abc', 10, 'abc'
```

Both `a` and `x` now contain `10`. Both `b` and `y` now contain `'abc'`.

They’re separate, as the values themselves were copied.



Variables	Values
x	10
y	'abc'
a	10
b	'abc'

Changing one does not change the other. Think of the variables as having no relationship to each other.

```
var x = 10;

var y = 'abc';

var a = x;

var b = y;

a = 5;
```

```
b = 'def';
```

```
console.log(x, y, a, b); // -> 10, 'abc', 5, 'def'
```

Objects

This will feel confusing, but bear with me and read through it.

Once you get through it, it'll seem easy.

Variables that are assigned a non-primitive value are given a *reference* to that value. That reference points to the object's location in memory. The variables don't actually contain the value.

Objects are created at some location in your computer's memory. When we write `arr = []`, we've created an array in memory. What the variable `arr` receives is the address, the location, of that array.

Let’s pretend that `address` is a new data type that is passed by value, just like `number` or `string`. An `address` points to the location, in memory, of a value that is passed by reference. Just like a string is denoted by quotation marks (`''` or `""`), an `address` will be denoted by arrow brackets, `<>`.

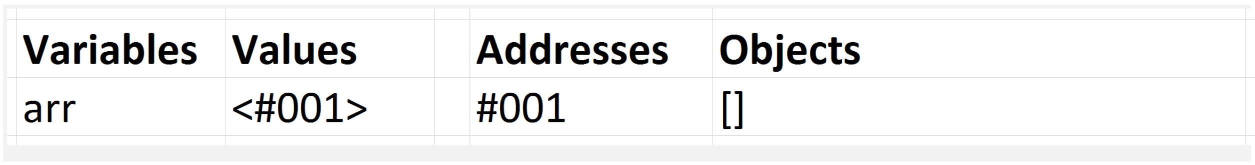
When we assign and use a reference-type variable, what we write and see is:

```
1) var arr = [];
```

```
2) arr.push(1);
```

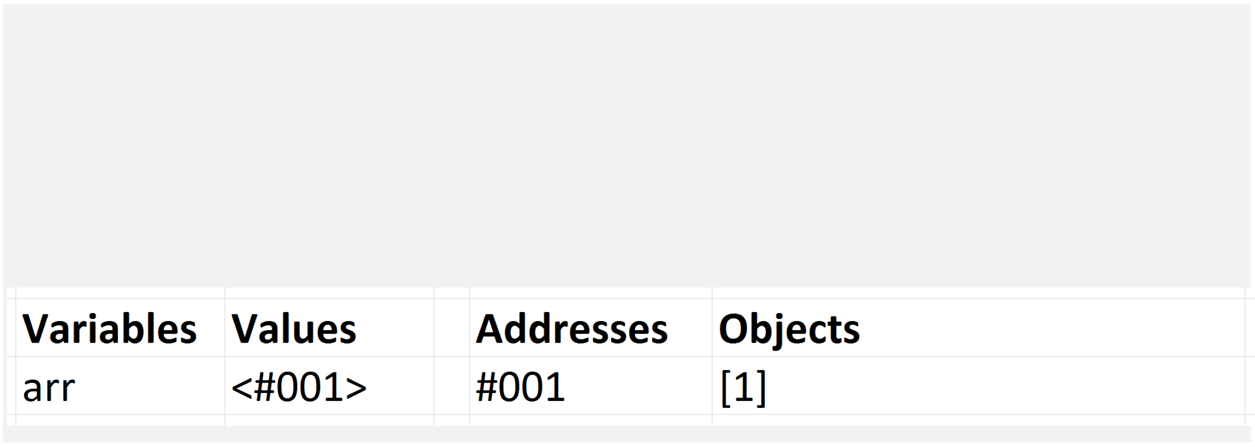
A representation of lines 1 and 2 above in memory is:

1.



Variables	Values		Addresses	Objects
arr	<#001>		#001	[]

2.



Variables	Values		Addresses	Objects
arr	<#001>		#001	[1]

Notice that the value, the address, contained by the variable `arr` is static. The array in memory is what changes. When we use `arr` to do something, such as pushing a value, the Javascript engine goes to the location of `arr` in memory and works with the information stored there.

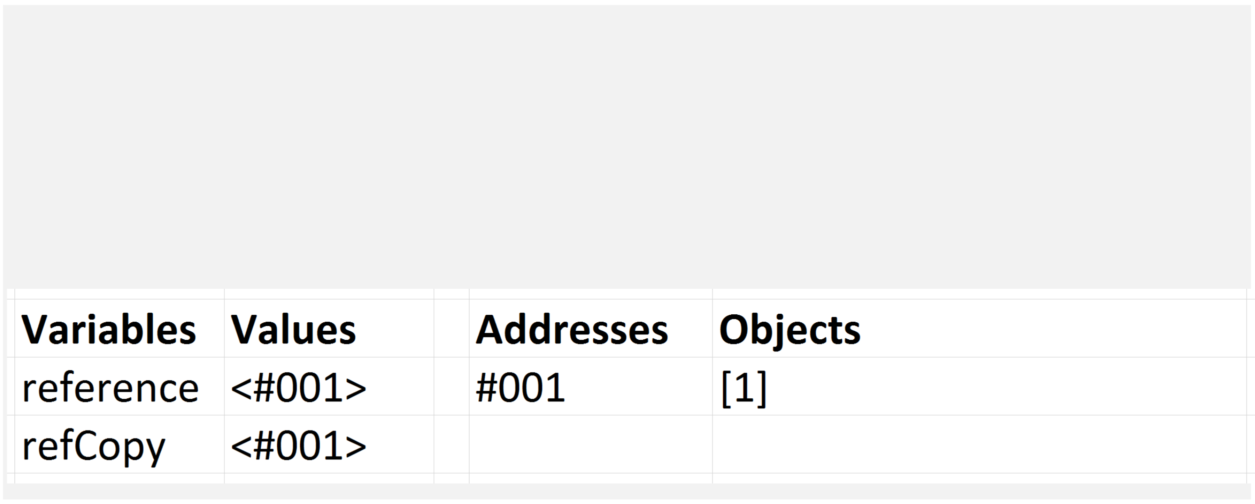
Assigning by Reference

When a reference type value, an object, is copied to another variable using `=`, the address of that value is what's actually copied over as if it were a primitive. **Objects are copied by reference** instead of by value.

```
var reference = [1];
```

```
var refCopy = reference;
```

The code above looks like this in memory.

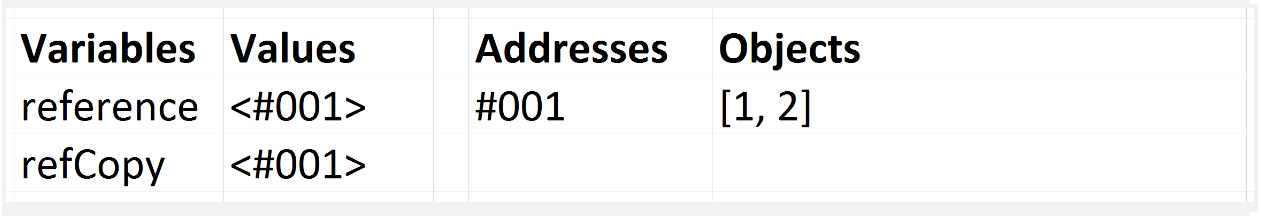


Variables	Values		Addresses	Objects
reference	<#001>		#001	[1]
refCopy	<#001>			

Each variable now contains a reference to the *same array*. That means that if we alter `reference`, `refCopy` will see those changes:

```
reference.push(2);
```

```
console.log(reference, refCopy); // -> [1, 2], [1, 2]
```



Variables	Values		Addresses	Objects
reference	<#001>		#001	[1, 2]
refCopy	<#001>			

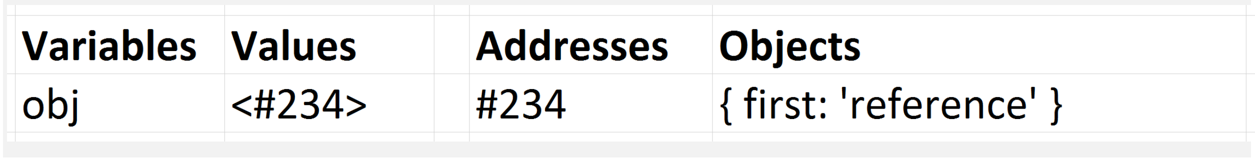
We’ve pushed `2` into the array in memory. When we use `reference` and `refCopy`, we’re pointing to that same array.

Reassigning a Reference

Reassigning a reference variable replaces the old reference.

```
var obj = { first: 'reference' };
```

In memory:



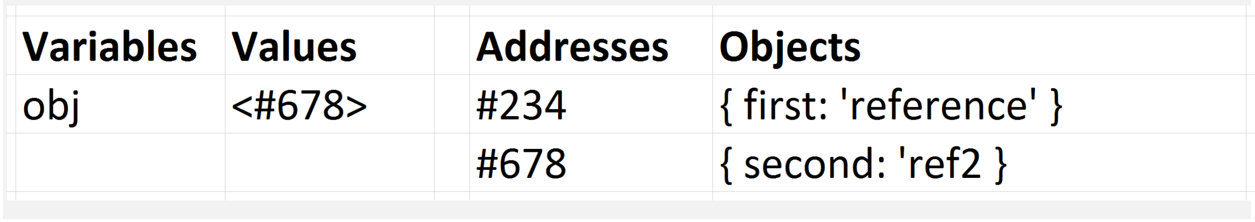
Variables	Values		Addresses	Objects
obj	<#234>		#234	{ first: 'reference' }

When we have a second line:

```
var obj = { first: 'reference' };
```

```
obj = { second: 'ref2' }
```

The address stored in `obj` changes. The first object is still present in memory, and so is the next object:



Variables	Values		Addresses	Objects
obj	<#678>		#234	{ first: 'reference' }
			#678	{ second: 'ref2 }

When there are no references to an object remaining, as we see for the address `#234` above, the Javascript engine can perform garbage collection. This just means that the programmer has lost all references to the object and can't use the object any more, so the engine can go ahead and safely delete it from memory. In this case, the object `{ first: 'reference' }` is no longer accessible and is available to the engine for garbage collection.

== and ===

When the equality operators, `==` and `===`, are used on reference-type variables, they check the reference. If the variables contain a reference to the same item, the comparison will result in `true`.

```
var arrRef = ['Hi!'];
```

```
var arrRef2 = arrRef;
```

```
console.log(arrRef === arrRef2); // -> true
```

If they're distinct objects, even if they contain identical properties, the comparison will result in `false`.

```
var arr1 = ['Hi!'];

var arr2 = ['Hi!'];


console.log(arr1 === arr2); // -> false
```

If we have two distinct objects and want to see if their properties are the same, the easiest way to do so is to turn them both into strings and then compare the strings. When the equality operators are comparing primitives, they simply check if the values are the same.

```
var arr1str = JSON.stringify(arr1);
var arr2str = JSON.stringify(arr2);
```

```
console.log(arr1str === arr2str); // true
```

Another option would be to recursively loop through the objects and make sure each of the properties are the same.

Passing Parameters through Functions

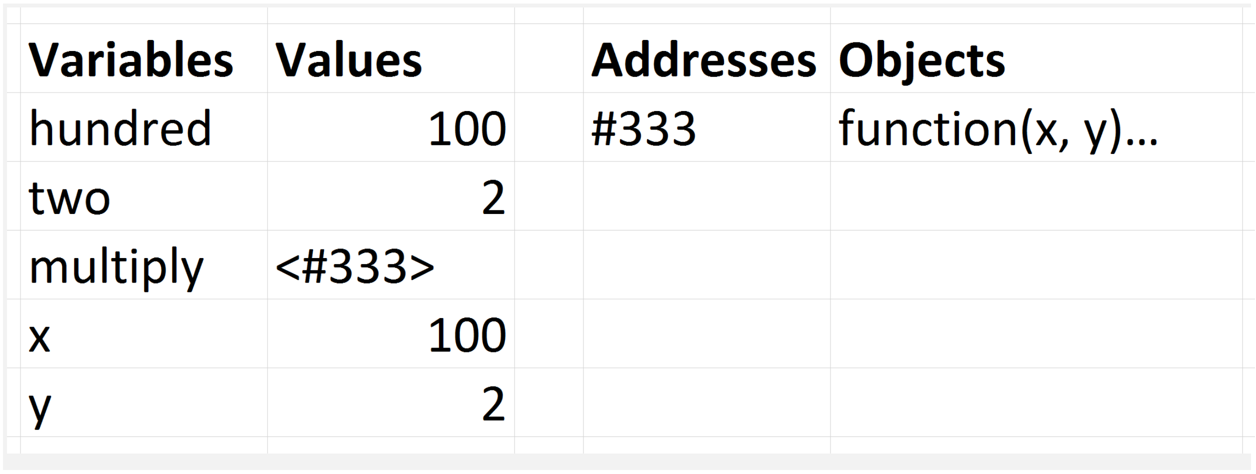
When we pass primitive values into a function, the function copies the values into its parameters. It’s effectively the same as using `=`.

```
var hundred = 100;
```

```
var two = 2;
```

```
function multiply(x, y) {
    // PAUSE

    return x * y;

}

var twoHundred = multiply(hundred, two);
```

In the example above, we give `hundred` the value `100`. When we pass it into `multiply`, the variable `x` gets that value, `100`. The value is copied over as if we used an `=` assignment. Again, the value of `hundred` is not affected. Here is a snapshot of what the memory looks like right at the PAUSE comment line in `multiply`.



Variables	Values		Addresses	Objects
hundred	100		#333	function(x, y)…
two	2			
multiply	<#333>			
x	100			
y	2			

Pure Functions

We refer to functions that don’t affect anything in the outside scope as *pure functions*. As long as a function only takes primitive values as parameters and doesn’t use any variables in its surrounding scope, it is automatically pure, as it can’t affect anything in the outside scope. All variables created inside are garbage-collected as soon as the function returns.

A function that takes in an Object, however, can mutate the state of its surrounding scope. If a function takes in an array reference and alters the array that it points to, perhaps by pushing to it, variables in the surrounding scope that reference that array see that change. After the function returns, the changes it makes persist in the outer scope. This can cause undesired side effects that can be difficult to track down.

Many native array functions, including `Array.map` and `Array.filter`, are therefore written as pure functions. They take in an array reference and internally, they copy the array and work with the copy instead of the original. This makes it so the original is untouched, the outer scope is unaffected, and we're returned a reference to a brand new array.

Let's go into an example of a pure vs. impure function.

```
function changeAgeImpure(person) {

    person.age = 25;
    return person;
}

var alex = {
    name: 'Alex',
    age: 30
};

var changedAlex = changeAgeImpure(alex);

console.log(alex); // -> { name: 'Alex', age: 25 }

console.log(changedAlex); // -> { name: 'Alex', age: 25 }
```

This impure function takes in an object and changes the property `age` on that object to be 25. Because it acts on the reference it was given, it directly changes the object `alex`. Note that when it returns the `person` object, it is returning the exact same object that was passed in. `alex` and `alexChanged` contain the same reference. It's redundant to return the `person` variable and to store the reference in a new variable.

Let's look at a pure function.

```
function changeAgePure(person) {
    var newPersonObj = JSON.parse(JSON.stringify(person));
    newPersonObj.age = 25;
    return newPersonObj;
}
```

```
var alex = {
    name: 'Alex',
    age: 30
};

var alexChanged = changeAgePure(alex);

console.log(alex); // -> { name: 'Alex', age: 30 }

console.log(alexChanged); // -> { name: 'Alex', age: 25 }
```

In this function, we use `JSON.stringify` to transform the object we're passed into a string, and then parse it back into an object with `JSON.parse`. By performing this transformation and storing the result in a new variable, we've created a new object. There are other ways to do the same thing such as looping through the original object and assigning each of its properties to a new object, but this way is simplest. The new object has the same properties as the original but it is a distinctly separate object in memory.

When we change the `age` property on this new object, the original is unaffected. This function is now pure. It can't affect any object outside its own scope, not even the object that was passed in. The new object needs to be returned and stored in a new variable or else it gets garbage collected once the function completes, as the object is no longer in scope.

2.How to copy by value a composite datatype (array+objects).

1. Spread Operator (Shallow copy)

Ever since ES6 dropped, this has been the most popular method. It's a brief syntax and you'll find it incredibly useful when using libraries like React and Redux.

```
numbers = [1, 2, 3];
numbersCopy = [...numbers];
```

Note: This doesn't safely copy multi-dimensional arrays. Array/object values are copied by *reference* instead of by *value*.

This is fine

```
numbersCopy.push(4);
console.log(numbers, numbersCopy);
// [1, 2, 3] and [1, 2, 3, 4]
// numbers is left alone
```

This is not fine

```
nestedNumbers = [[1], [2]];
numbersCopy = [...nestedNumbers];

numbersCopy[0].push(300);
console.log(nestedNumbers, numbersCopy);
// [[1, 300], [2]]
// [[1, 300], [2]]
// They've both been changed because they share references
```

2. Good Old for() Loop (Shallow copy)

I imagine this approach is the *least* popular, given how trendy functional programming's become in our circles.

Pure or impure, declarative or imperative, it gets the job done!

```
numbers = [1, 2, 3];
numbersCopy = [];

for (i = 0; i < numbers.length; i++) {
  numbersCopy[i] = numbers[i];
}
```

Note: This doesn't safely copy multi-dimensional arrays. Since you're using the = operator, it'll assign objects/arrays by *reference* instead of by *value*.

This is fine

```
numbersCopy.push(4);
console.log(numbers, numbersCopy);
// [1, 2, 3] and [1, 2, 3, 4]
// numbers is left alone
```

This is not fine

```
nestedNumbers = [[1], [2]];
numbersCopy = [];
```

```
for (i = 0; i < nestedNumbers.length; i++) {
  numbersCopy[i] = nestedNumbers[i];
}

numbersCopy[0].push(300);
console.log(nestedNumbers, numbersCopy);
// [[1, 300], [2]]
// [[1, 300], [2]]
// They've both been changed because they share references
```

3. Good Old while() Loop (Shallow copy)

Same as `for`—impure, imperative, blah, blah, blah...it works! ?

```
numbers = [1, 2, 3];
numbersCopy = [];
i = -1;

while (++i < numbers.length) {
  numbersCopy[i] = numbers[i];
}
```

Note: This also assigns objects/arrays by *reference* instead of by *value.*

This is fine

```
numbersCopy.push(4);
console.log(numbers, numbersCopy);
// [1, 2, 3] and [1, 2, 3, 4]
// numbers is left alone
```

This is not fine

```
nestedNumbers = [[1], [2]];
numbersCopy = [];

i = -1;

while (++i < nestedNumbers.length) {
  numbersCopy[i] = nestedNumbers[i];
}

numbersCopy[0].push(300);
console.log(nestedNumbers, numbersCopy);
// [[1, 300], [2]]
// [[1, 300], [2]]
// They've both been changed because they share references
```

4. Array.map (Shallow copy)

Back in modern territory, we'll find the `map` function. Rooted in mathematics, `map` is the concept of transforming a set into another type of set, while preserving structure.

In English, that means `Array.map` returns an array of the same length every single time.

To double a list of numbers, use `map` with a `double` function.

```
numbers = [1, 2, 3];
double = (x) => x * 2;

numbers.map(double);
```

What about cloning??

True, this article's about cloning arrays. To duplicate an array, just return the element in your `map` call.

```
numbers = [1, 2, 3];
numbersCopy = numbers.map((x) => x);
```

If you'd like to be a bit more mathematical, `(x) => x` is called *identity*. It returns whatever parameter it's been given.

`map(identity)` clones a list.

```
identity = (x) => x;
numbers.map(identity);
// [1, 2, 3]
```

Note: This also assigns objects/arrays by *reference* instead of by *value*.

5. Array.filter (Shallow copy)

This function returns an array, just like `map`, but it's not guaranteed to be the same length.

What if you're filtering for even numbers?

```
[1, 2, 3].filter((x) => x % 2 === 0);
```

```
// [2]
```

The input array length was 3, but the resulting length is 1.

If your `filter`'s predicate always returns `true`, however, you get a duplicate!

```
numbers = [1, 2, 3];
numbersCopy = numbers.filter(() => true);
```

Every element passes the test, so it gets returned.

Note: This also assigns objects/arrays by *reference* instead of by *value*.

6. Array.reduce (Shallow copy)

I almost feel bad using `reduce` to clone an array, because it's so much more powerful than that. But here we go…

```
numbers = [1, 2, 3];

numbersCopy = numbers.reduce((newArray, element) => {
  newArray.push(element);

  return newArray;
}, []);
```

`reduce` transforms an initial value as it loops through a list.

Here the initial value is an empty array, and we're filling it with each element as we go. That array must be returned from the function to be used in the next iteration.

Note: This also assigns objects/arrays by *reference* instead of by *value*.

7. Array.slice (Shallow copy)

`slice` returns a *shallow* copy of an array based on the provided start/end index you provide.

If we want the first 3 elements:

```
[1, 2, 3, 4, 5].slice(0, 3);
// [1, 2, 3]
// Starts at index 0, stops at index 3
```

If we want all the elements, don't give any parameters

```
numbers = [1, 2, 3, 4, 5];
numbersCopy = numbers.slice();
// [1, 2, 3, 4, 5]
```

Note: This is a *shallow* copy, so it also assigns objects/arrays by *reference* instead of by *value*.

8. JSON.parse and JSON.stringify (Deep copy)

`JSON.stringify` turns an object into a string.

`JSON.parse` turns a string into an object.

Combining them can turn an object into a string, and then reverse the process to create a brand new data structure.

Note: This one safely copies deeply nested objects/arrays!

```
nestedNumbers = [[1], [2]];
numbersCopy = JSON.parse(JSON.stringify(nestedNumbers));

numbersCopy[0].push(300);
console.log(nestedNumbers, numbersCopy);

// [[1], [2]]
// [[1, 300], [2]]
// These two arrays are completely separate!
```

9. Array.concat (Shallow copy)

`concat` combines arrays with values or other arrays.

```
[1, 2, 3].concat(4); // [1, 2, 3, 4]
[1, 2, 3].concat([4, 5]); // [1, 2, 3, 4, 5]
```

If you give nothing or an empty array, a shallow copy's returned.

```
[1, 2, 3].concat(); // [1, 2, 3]
[1, 2, 3].concat([]); // [1, 2, 3]
```

Note: This also assigns objects/arrays by *reference* instead of by *value.*

10. Array.from (Shallow copy)

This can turn any iterable object into an array. Giving an array returns a shallow copy.

```
numbers = [1, 2, 3];
numbersCopy = Array.from(numbers);
// [1, 2, 3]
```

Note: This also assigns objects/arrays by *reference* instead of by *value.*

Conclusion

Well, this was fun ?

I tried to clone using just 1 step. You'll find many more ways if you employ multiple methods and techniques.