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

**Primitive Data Types**

**COPY BY VALUE:**

In JavaScript, we can divide data types into two different buckets, primitive data types and objects.

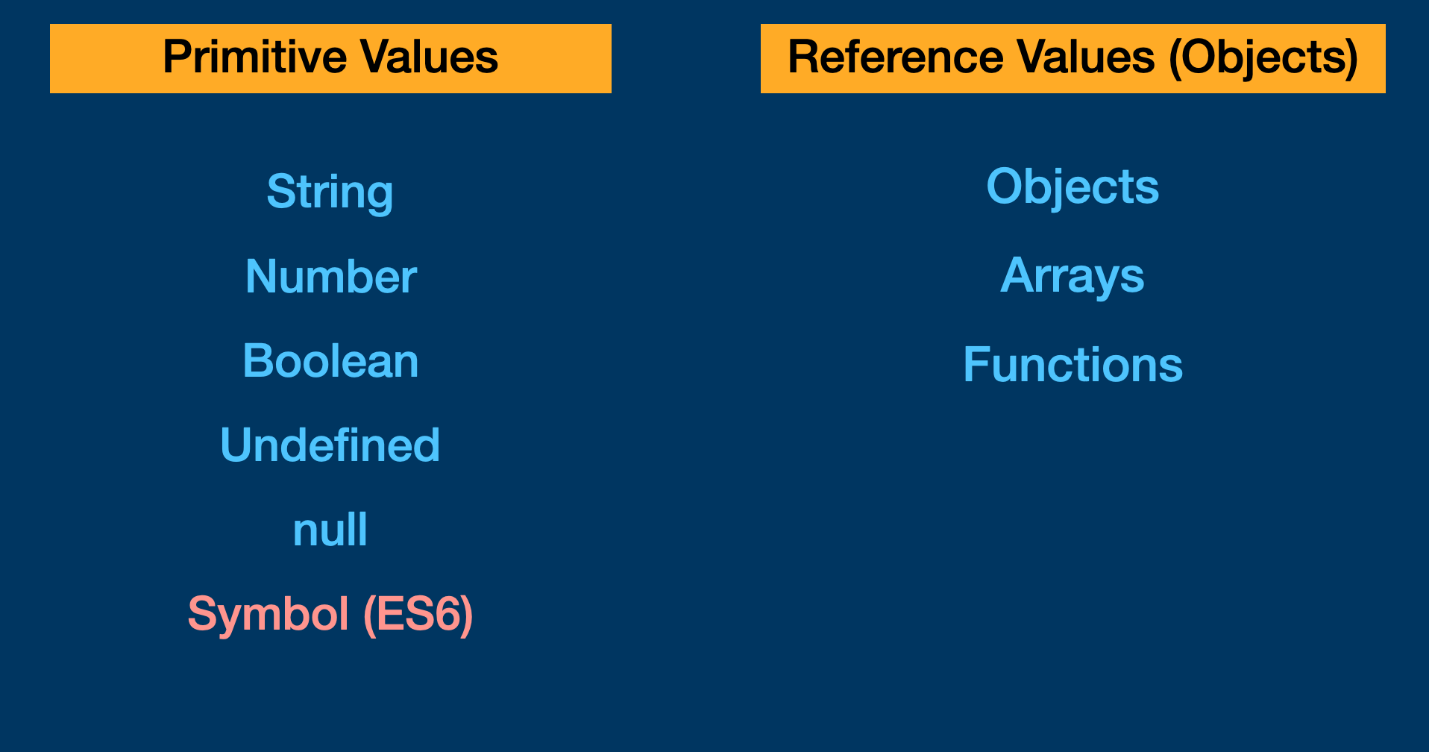


Chart illustrating the primitive data types, or values, and the object types, or reference values

There are six primitive data types in JavaScript: string, number, boolean, undefined, null, and symbol as of ES6.

Primitive data types are passed, or copied, **by value** and are immutable, meaning that the existing value cannot be altered the way an array or an object can. Let’s take a look at the code below to see this in action.

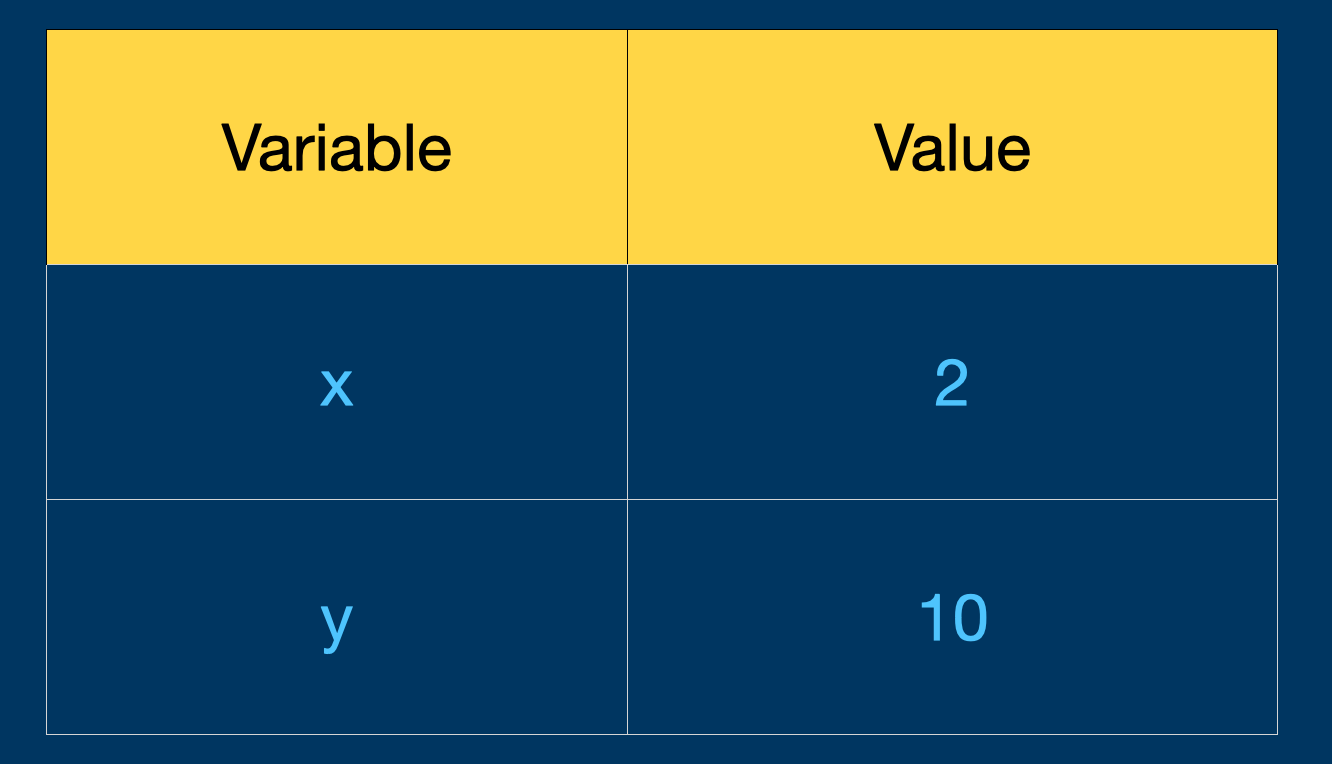


Here we have created two variables, x = 10 and y = x. Since 10 is a number and a primitive value, when we set y = x we are copying the value, i.e., 10, and assigning it to y. We can also visualize this using the chart below.



If we were to change the value of x, we would see that y retains its value of 10. Again, this is because primitive values are **copied**, so y's value is independent of x's value. Think of it as making a photocopy of a picture. After making the copy, you have two identical pictures: an original and a facsimile. If you were to cut the original in half, only the original would be altered, and the facsimile would remain exactly the same.



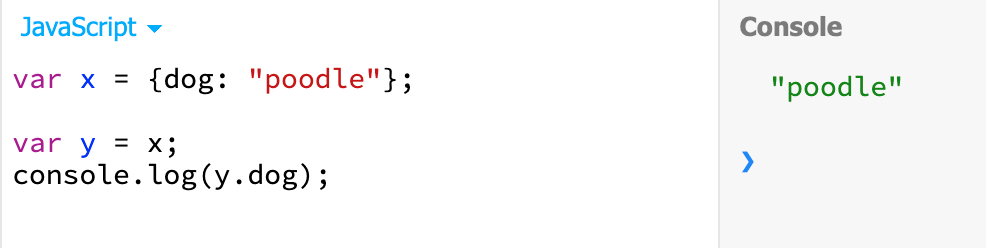


We overwrote the value of x but y remains the same

**Reference Objects**

**COPY BY REFERENCE:**

Objects, on the other hand, are passed**by reference** and point to a location in memory for the value, not the value itself. Let’s take a look at this in our code.



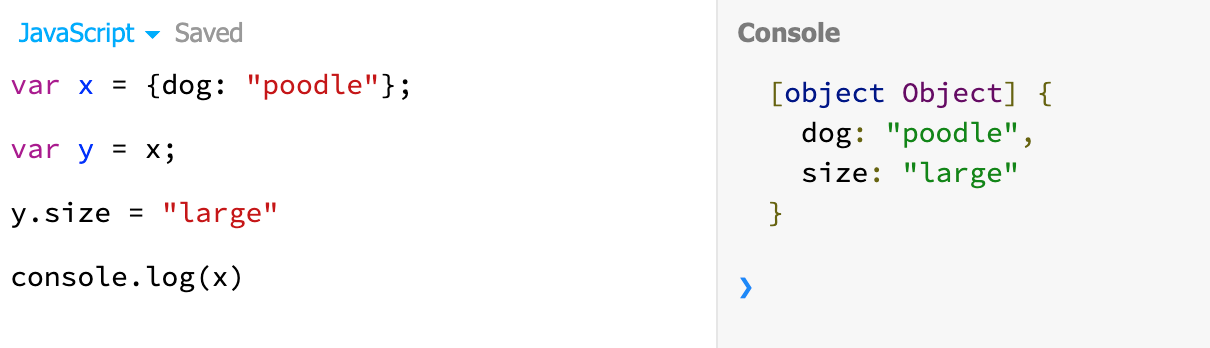
Variables x and y now point to objects rather than primitive data types.

In this example, x is now an object pointing to {dog: "poodle"}. When we create the variable y and assign it the value of x, we are now able to tap into the various properties of x which includes the value for dog, i.e., "poodle". This seems like the exact same logic used for primitive values, but let us take a look at our handy-dandy chart below to see the subtle, but important difference.



Both x and y point to a (made up) address in memory which stores a reference to the object

Now this chart looks a little bit different from when our variables x and y held primitive data types. In this version, we see that the values for both x and y are not data types but **references** to an address in memory, the same address in fact! Now let us take a look at what happens to x if we add a new property of size to y…



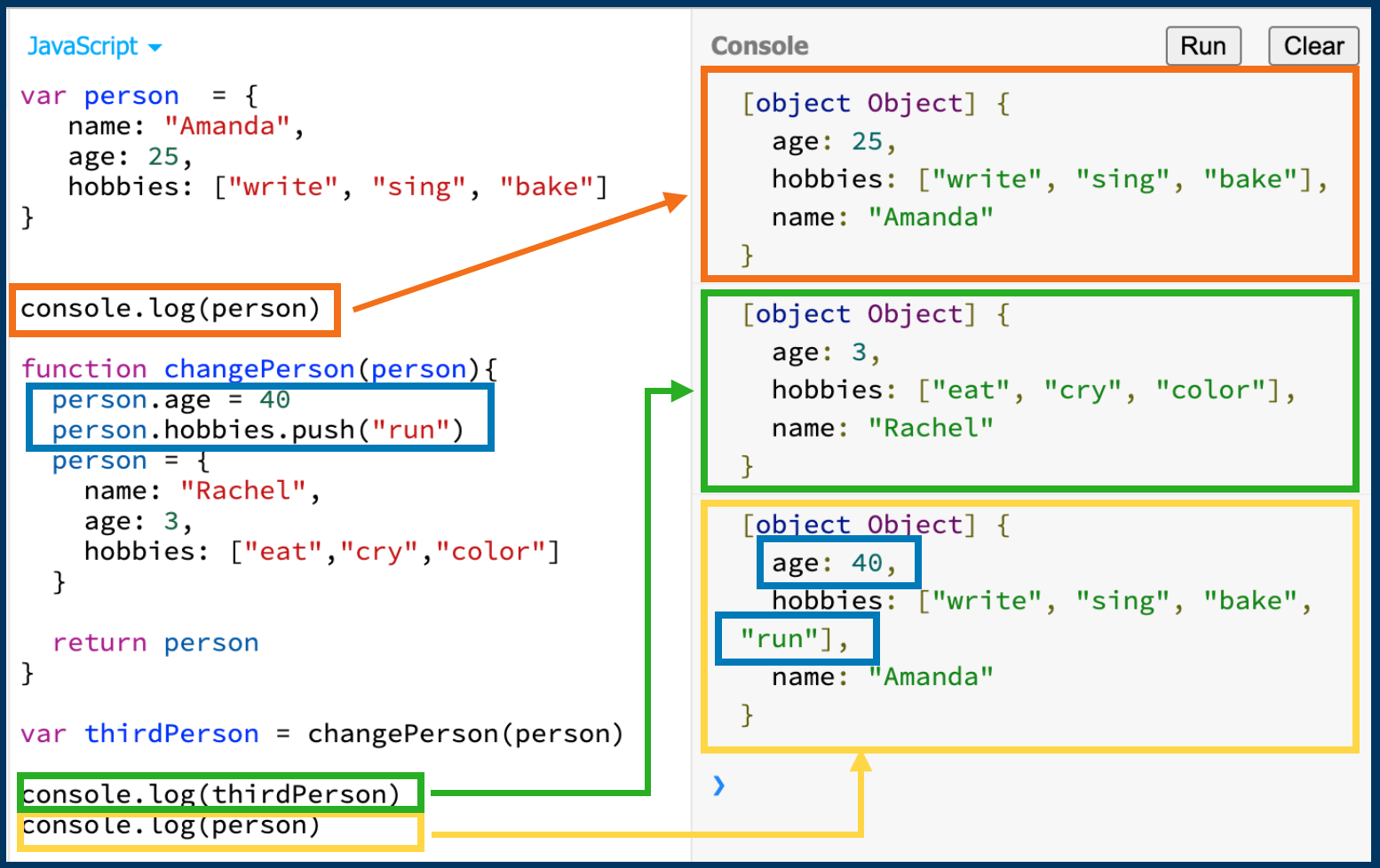
x still returns an object but now it has an additional property of size also! Again, this is because both x and y point to the same reference object, so any changes made to one variable, will be visible in the other.



Chart illustrates how a change in y updates the reference value that is shared with x

To help me remember this concept, I like to think of reference values as a house and the variables as people who live in that house. All the residents (variables) can say “I have a house” and point to the same house. If a single resident decides they want to paint the house yellow, then all the residents now have a yellow house because it is shared.

Let us take a look at one more example that contain a variety of reference objects.



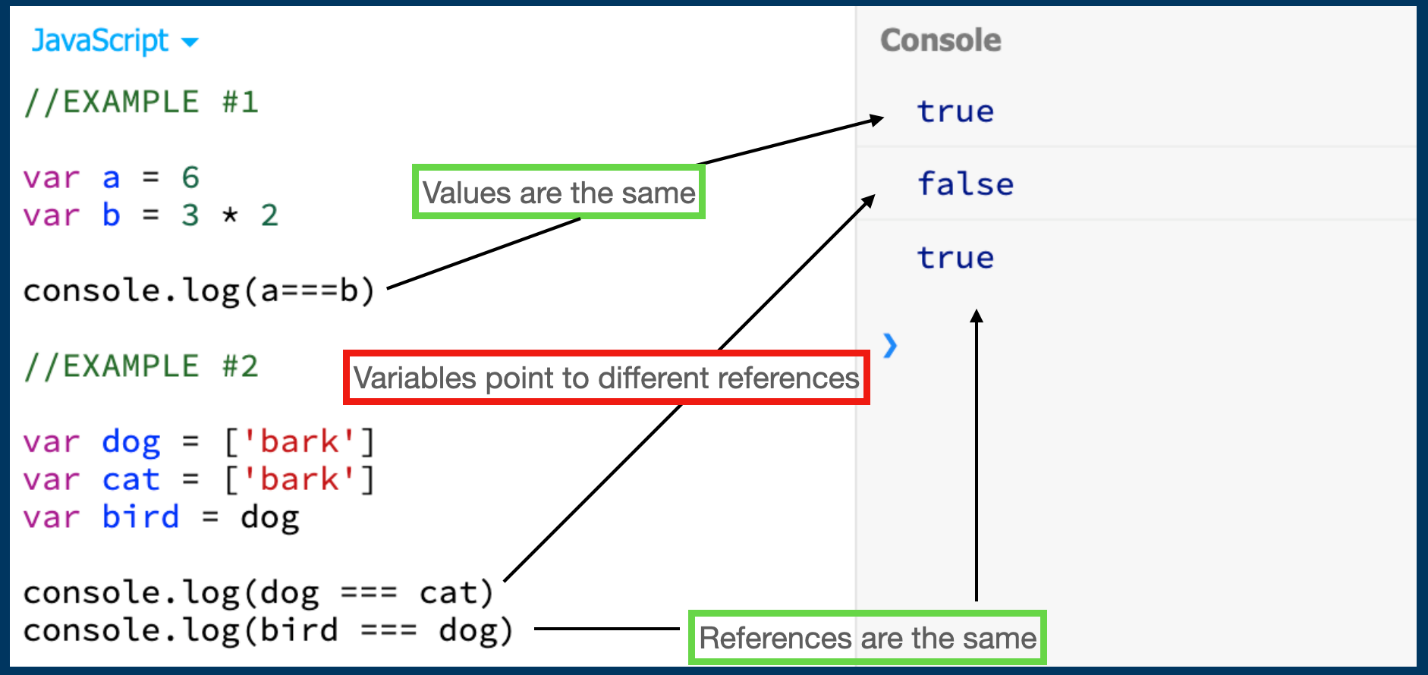
In this code, we start out with a variable person that contains properties of name, age, and hobbies. When we print this object to the console, we get exactly what we expect — the same object we just created.

Next, we have a function called changePerson that takes in an argument, makes a few changes, then returns an object. When we create the variable thirdPerson, we invoke thechangePerson function by passing our original object of person into it. The interesting bit is what happens when we print to the console thirdPerson and person again.

Notice that console.log(thirdPerson) returns a whole new object with new properties. Now look at what console.log(person) returns. This is similar to our original object, but it contains new property values that were introduced in our changePerson function.

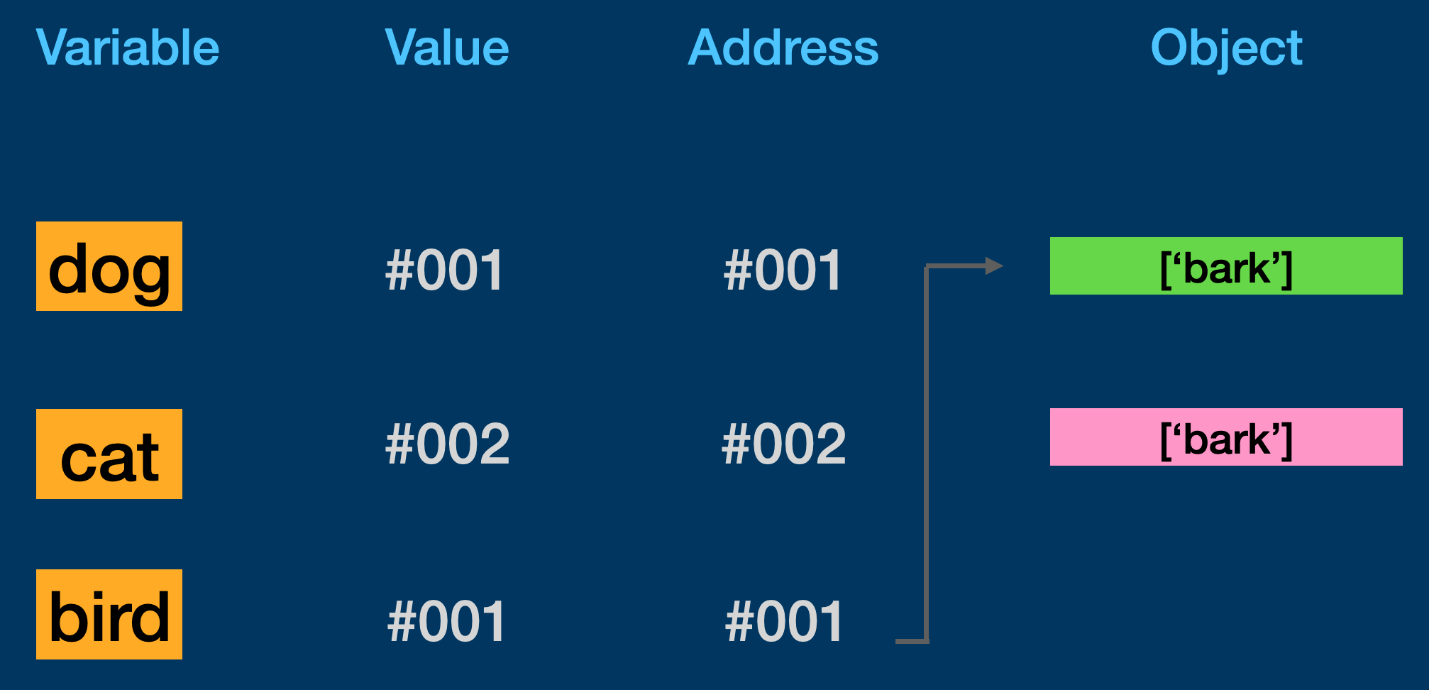
Checking for Equality

Finally, let us take a look at how primitive data types and reference objects behave with equality operators.



When it comes to primitive data types, it does not matter what is to the right of the = sign as long as the values are the same. We can see this above with variables a and b which are written differently but evaluate to the same value when we use the ===, the strict equality operator.

The opposite is true in the second example for dog and cat. While it may appear that they contain identical values, ['bark'] is an array, and a reference object, which means the === is checking to see if both dog and cat have the same reference to the value in memory. On the other hand, bird === dog is true because they share the same reference object.



dog and bird share the same reference while cat does not despite having an identical array.

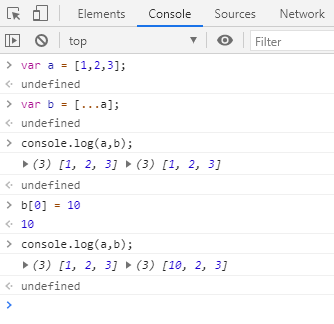
**3.** **How to copy by value a composite datatype (array + objects)?**

There are three ways to copy by value for composite data types.

1. Using the Spread operator (…) operator.
2. Using the Object.assign() method.
3. Using the JSON.stringify() and JSON.parse() methods

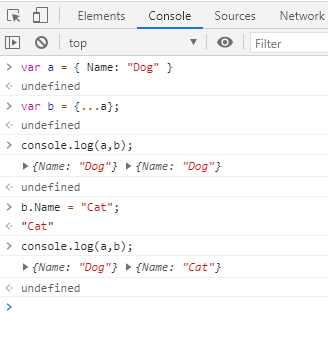
# **The Spread Operator**

The spread operator (…) does a shallow copy of the variable data. It copies the values present that is the non primitive data type.



For arrays

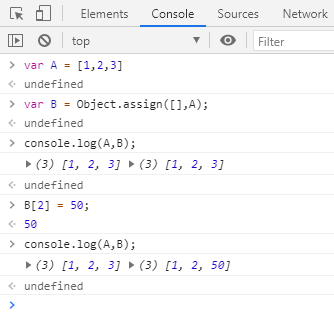
In the above image the variable “b” value was changed. But it did not affect the original variable “a”.



For Objects

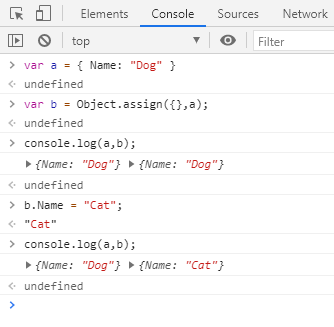
# **The Object.assign()**

The Object.assign() method copies all properties from one source object to a target object. It returns the target object. This does the shallow copy of the original object.



For Array

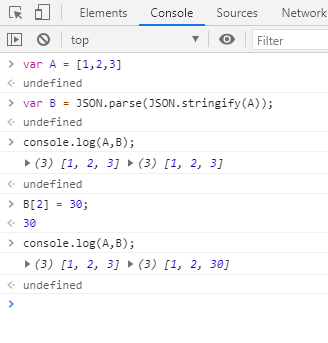
In case of Objects the Object.assign is specified with {}.



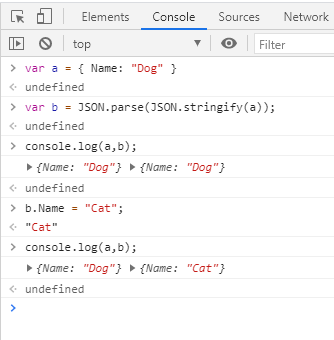
For Objects

# **The JSON.parse() and JSON.stringify()**

The JSON object is available in all the browsers, this has two methods for JSON data i.e the parse and the stringify. Parse takes a JSON string and transforms it into JS Object and Stringify takes JS Object and transforms into a JSON Object.



For Arrays



For Objects

These are the ways by which a composite datatype can be copied by value.