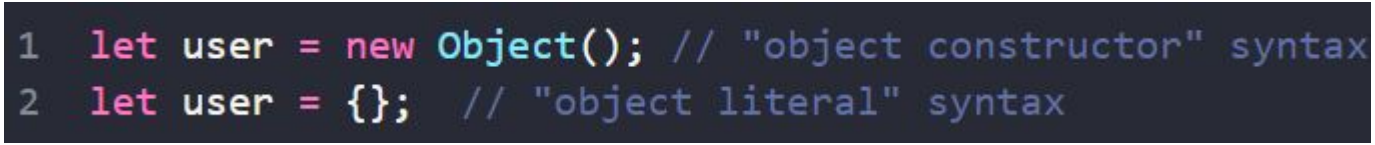
Javascript Day – 6

Objects

What are objects ?

An object can be created with figure brackets {…} with an optional list of properties. A property is a “key: value” pair, where key is a string (also called a “property name”), and value can be anything.

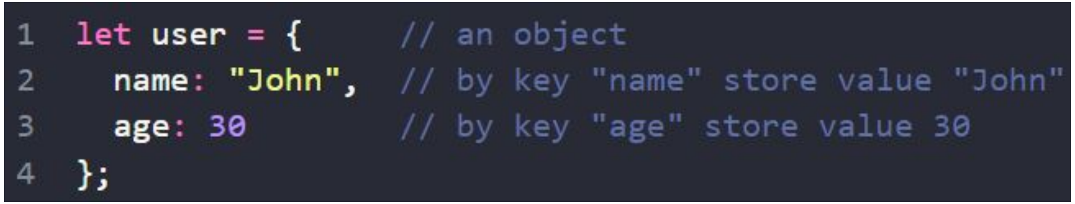
An empty object (“empty cabinet”) can be created using one of two syntaxes:



```
let user = new Object(); // "object constructor" syntax
let user = {};  // "object literal" syntax
```

Object Properties

We can immediately put some properties into {...} as “key: value” pairs:



```
let user = {     // an object
  name: "John",  // by key "name" store value "John"
  age: 30        // by key "age" store value 30
};
```

A property has a key (also known as “name” or “identifier”) before the colon ":" and a value to the right of it.

In the user object, there are two properties:

The first property has the name "name" and the value "John".

The second one has the name "age" and the value 30.

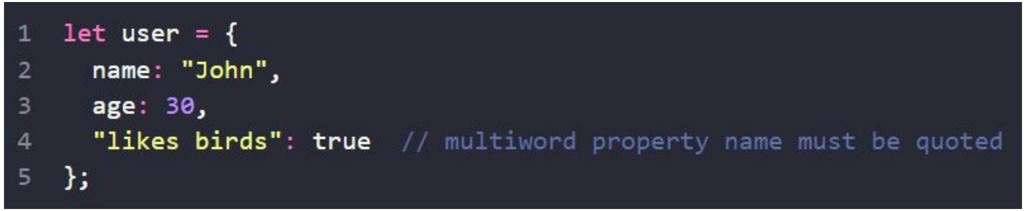
Accessing Object Properties

We can access object properties using two methods.

1. dot notation
2. Square bracket notation

To remove a property from an object we can use the **delete operator**

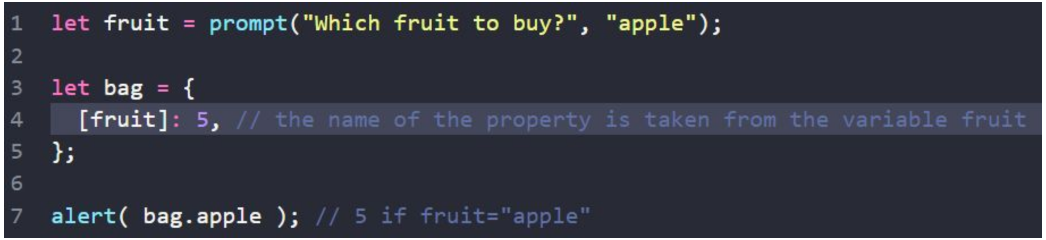
If our property name is multiword, we will have to enclose it in quotes



```
let user = {
  name: "John",
  age: 30,
  "likes birds": true  // multiword property name must be quoted
};
```

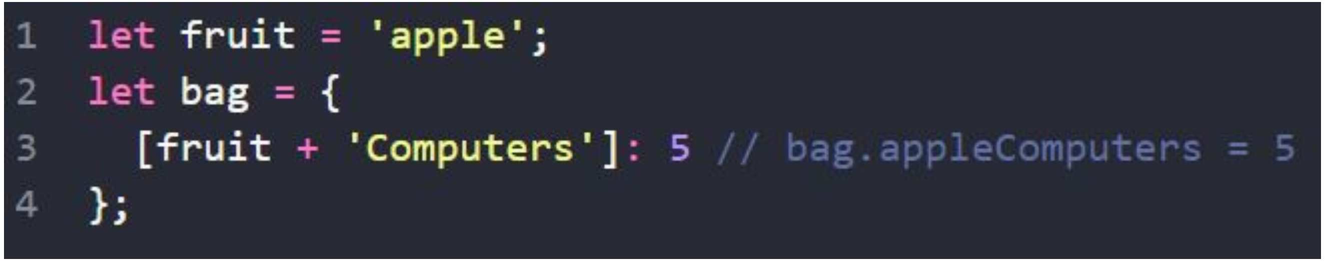
Computed Object Properties

We can use square brackets in an object literal, when creating an object. That’s called computed properties.



```
let fruit = prompt("Which fruit to buy?", "apple");

let bag = {
  [fruit]: 5, // the name of the property is taken from the variable fruit
};

alert( bag.apple ); // 5 if fruit="apple"
```

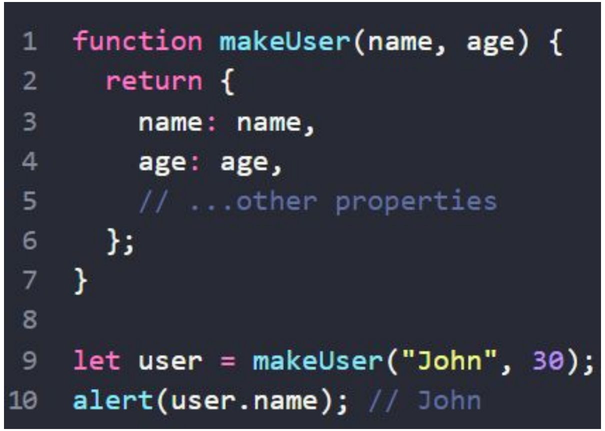
We can use more complex expressions inside square brackets:



```
let fruit = 'apple';
let bag = {
  [fruit + 'Computers']: 5 // bag.appleComputers = 5
};
```

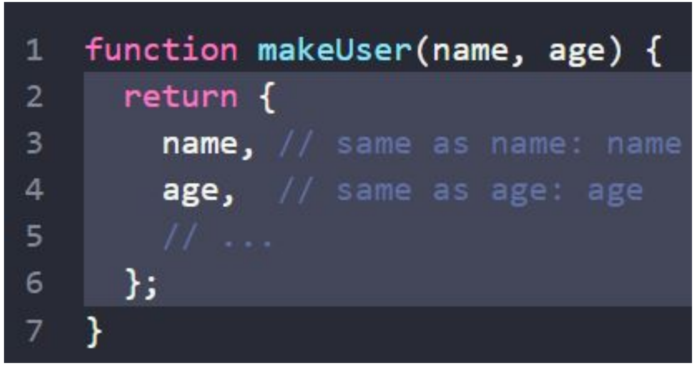
Property value shorthand

In real code, we often use existing variables as values for property names. ➳



```
function makeUser(name, age) {
  return {
    name: name,
    age: age,
    // ...other properties
  };
}

let user = makeUser("John", 30);
alert(user.name); // John
```

Instead of name:name we can just write name, like this: ➳

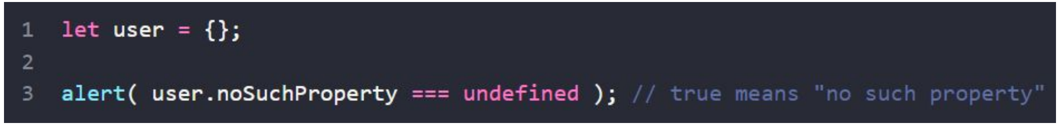


```
function makeUser(name, age) {
  return {
    name, // same as name: name
    age,  // same as age: age
    // ...
  };
}
```

Check if object property exists ?

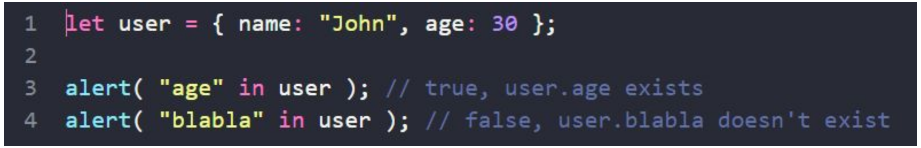
A notable feature of objects in JavaScript, compared to many other languages, is that it’s possible to access any property. There will be no error if the property doesn’t exist!

Reading a non-existing property just returns undefined. So we can easily test whether the property exists:



```
let user = {};

alert( user.noSuchProperty === undefined ); // true means "no such property"
```

There’s also a special operator "in" for that.

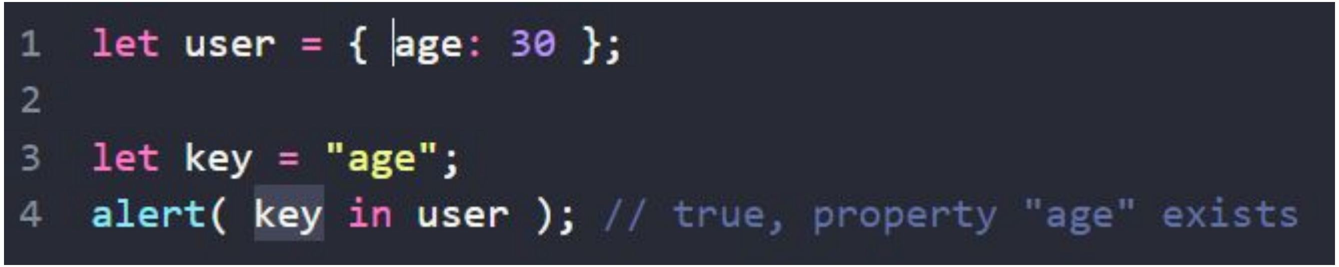


```
let user = { name: "John", age: 30 };

alert( "age" in user ); // true, user.age exists
alert( "blabla" in user ); // false, user.blabla doesn't exist
```

Check if object property exists ? (contd.)

Please note that on the left side of in there must be a property name. That’s usually a quoted string.

If we omit quotes, that means a variable should contain the actual name to be tested. For instance:



```
let user = { age: 30 };

let key = "age";
alert( key in user ); // true, property "age" exists
```

Check if object property exists ? (contd.)

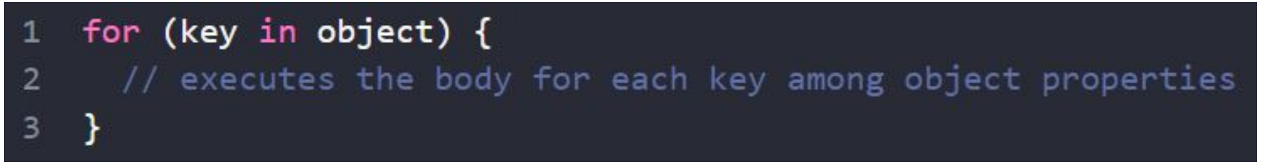
Why does the in operator exist? Isn’t it enough to compare against undefined?

Well, most of the time the comparison with undefined works fine. But there’s a special case when it fails, but "in" works correctly.

It’s when an object property exists, but stores undefined:

For In Loop

To walk over all keys of an object, there exists a special form of the loop: for..in. This is a completely different thing from the for(;;) construct that we studied before.



```
for (key in object) {
  // executes the body for each key among object properties
}
```

Summary

Objects are associative arrays with several special features.

They store properties (key-value pairs), where:

- Property keys must be strings or symbols (usually strings).
- Values can be of any type.

To access a property, we can use:

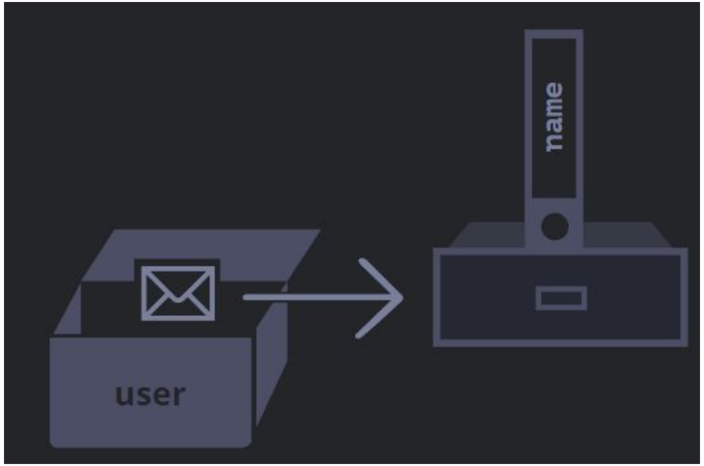
- The dot notation: obj.property.
- Square brackets notation obj["property"]. Square brackets allow taking the key from a variable, like obj[varWithKey].

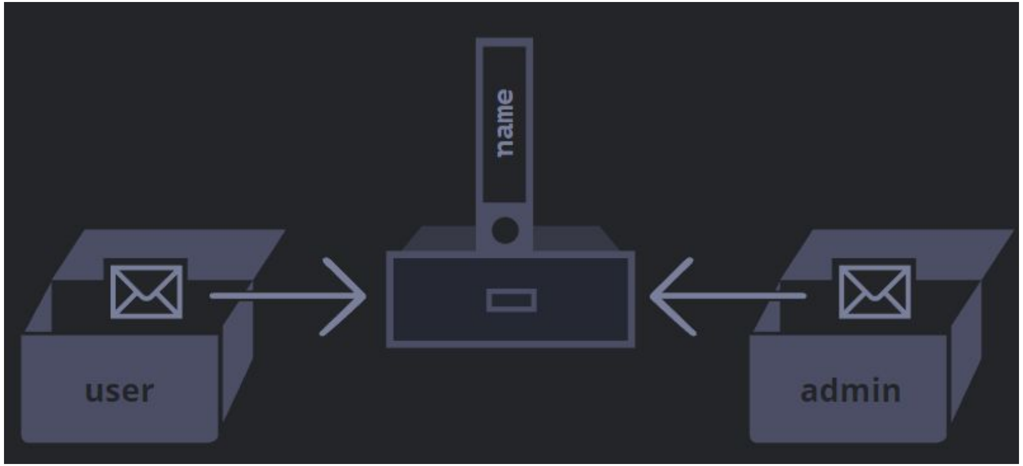
Additional operators:

- To delete a property: delete obj.prop.
- To check if a property with the given key exists: "key" in obj.
- To iterate over an object: for (let key in obj) loop.

Object reference and copying

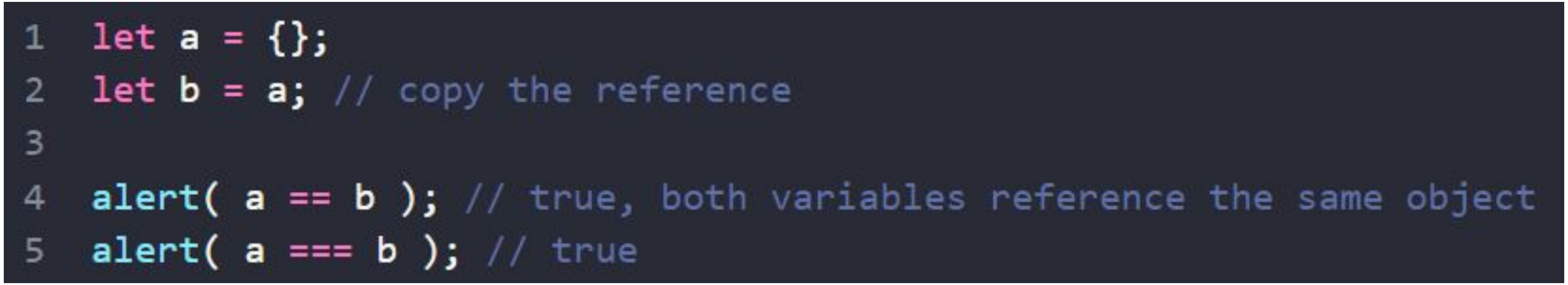
One of the fundamental differences of objects versus primitives is that objects are stored and copied “by reference”, whereas primitive values: strings, numbers, booleans, etc – are always copied “as a whole value”.





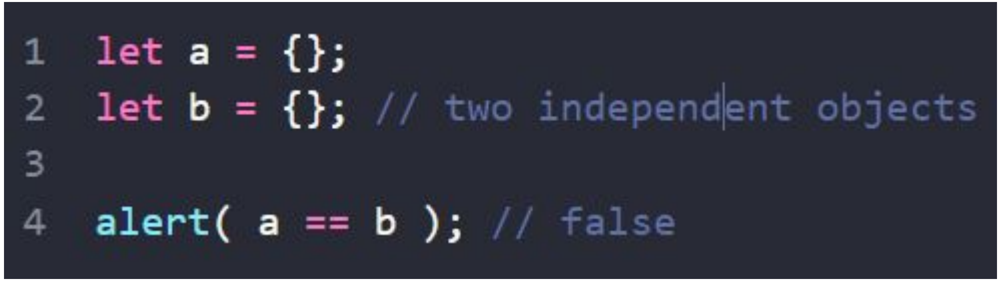
Comparison by reference

Two objects are equal only if they are the same object.



```
let a = {};
let b = a; // copy the reference

alert( a == b ); // true, both variables reference the same object
alert( a === b ); // true
```

And here two independent objects are not equal, even though they look alike (both are empty):



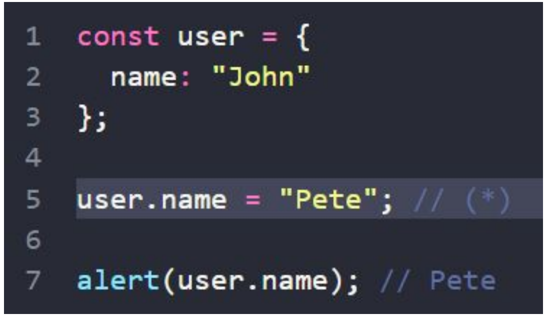
```
let a = {};
let b = {}; // two independent objects

alert( a == b ); // false
```

Const object can be modified

An important side effect of storing objects as references is that an object declared as const can be modified.

It might seem that the line (*) would cause an error, but it does not. The value of user is constant, it must always reference the same object, but properties of that object are free to change.

In other words, the const user gives an error only if we try to set user=... as a whole.



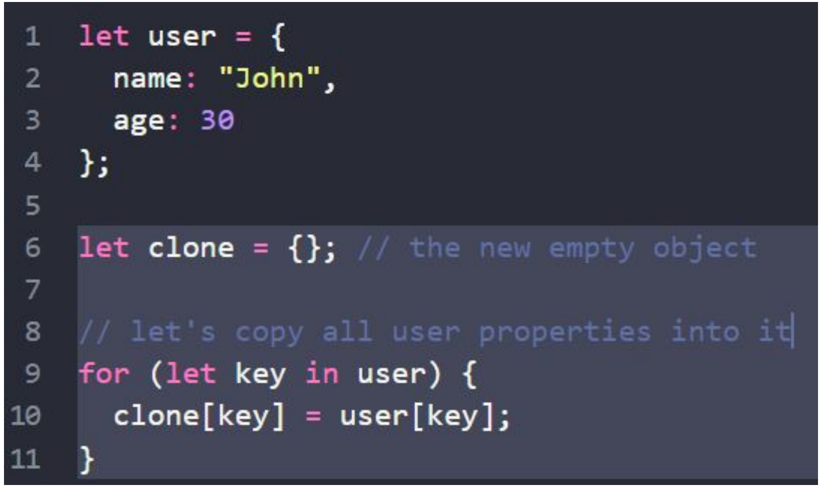
```
const user = {
  name: "John"
};

user.name = "Pete"; // (*)

alert(user.name); // Pete
```

Cloning and merging, Object.assign

Copying an object variable creates one more reference to the same object.

But what if we need to duplicate an object?

We can create a new object and replicate the structure of the existing one, by iterating over its properties and copying them on the primitive level.

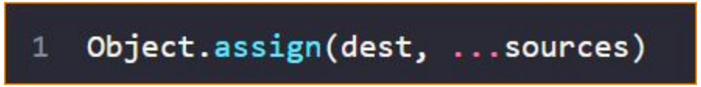


```
let user = {
  name: "John",
  age: 30
};

let clone = {}; // the new empty object

// let's copy all user properties into it
for (let key in user) {
  clone[key] = user[key];
}
```

Cloning and merging, Object.assign

We can also use the method Object.assign.

The syntax is ↣



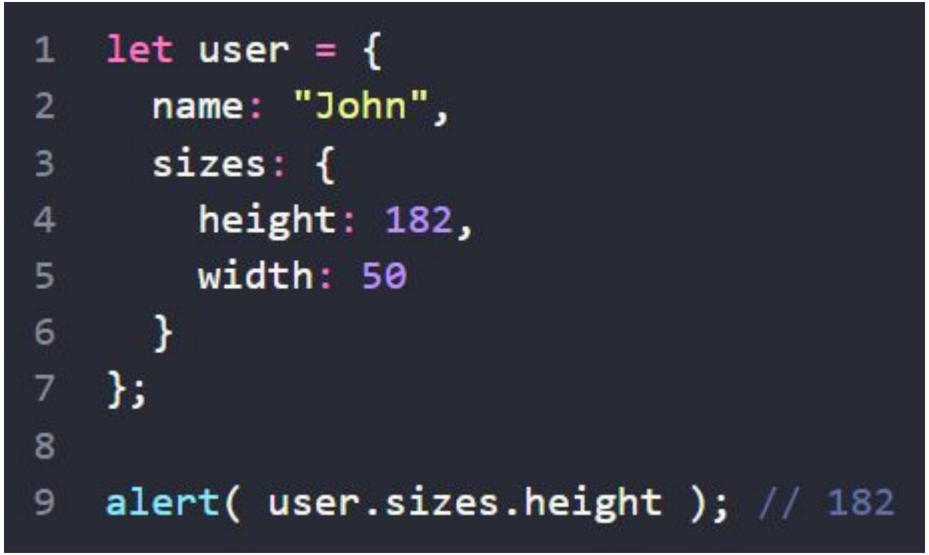
```
Object.assign(dest, ...sources)
```

- The first argument dest is a target object.
- Further arguments is a list of source objects.

It copies the properties of all source objects into the target dest, and then returns it as the result.

Nested cloning

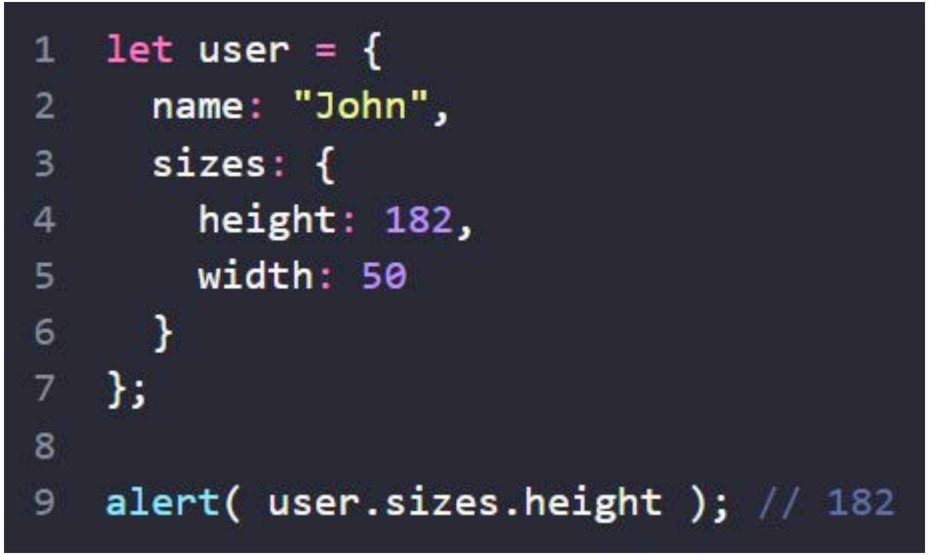
Until now we assumed that all properties of objects are primitive. But properties can be references to other objects.



```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

alert( user.sizes.height ); // 182
```

Nested cloning

Now it’s not enough to copy clone.sizes = user.sizes, because user.sizes is an object, and will be copied by reference, so clone and user will share the same sizes:



```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

alert( user.sizes.height ); // 182
```

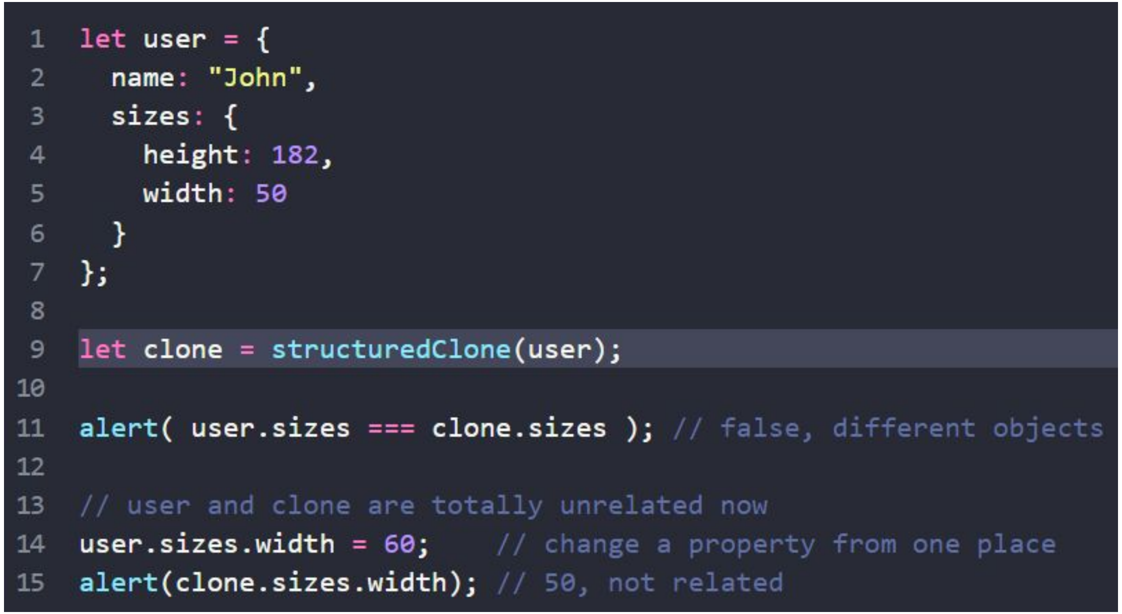
Nested cloning

To fix that and make user and clone truly separate objects, we should use a cloning loop that examines each value of user[key] and, if it’s an object, then replicate its structure as well. That is called a “deep cloning” or “structured cloning”.

There’s **structuredClone** method that implements deep cloning.

structuredClone

The call structuredClone(object) clones the object with all nested properties.



```javascript
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

let clone = structuredClone(user);

alert( user.sizes === clone.sizes ); // false, different objects

// user and clone are totally unrelated now
user.sizes.width = 60;    // change a property from one place
alert(clone.sizes.width); // 50, not related
```