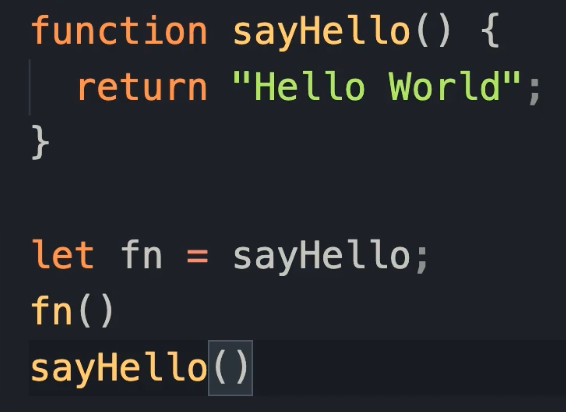
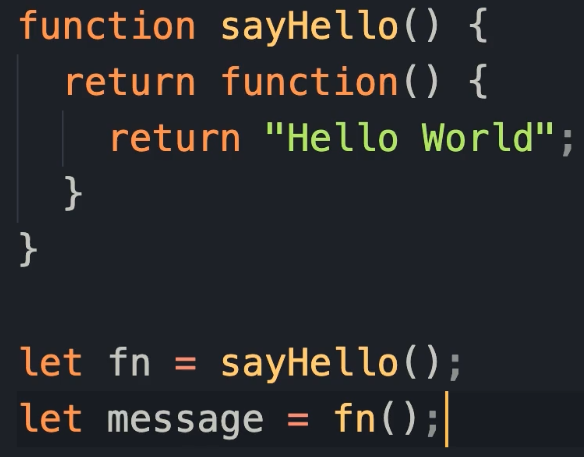
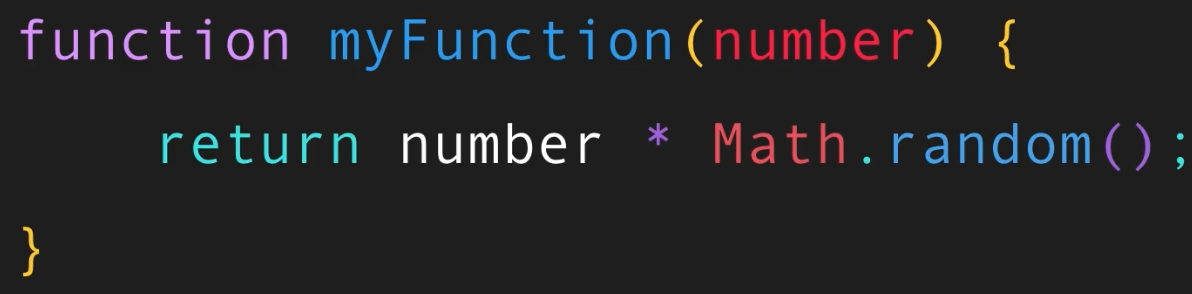
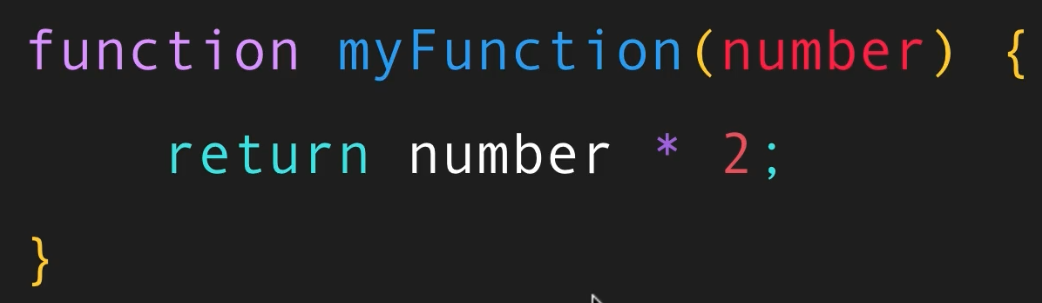
* **Functional programming** is a programming paradigm where programs are constructed by applying and composing functions. When a pure function is called with some given arguments, it will always return the same result, and cannot be affected by any mutable state or other side effects.
* **Functions Are first class citizen in JavaScript.** This means we can treat them as any other variables (assign them to a variable, pass as an argument of a function, return them from a function). The examples are given in order below

* **Higher Order Function** is a function that takes a function as an argument, or returns a function. The 2nd and 3rd image above is an example.
* **Pure functions:** The function return values are identical for identical arguments. The function has no side effects. Example,

 not pure as it generates random value each time is called.

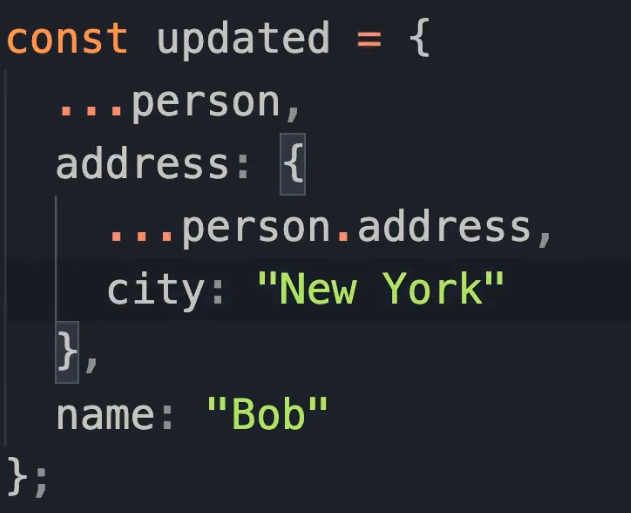
 pure as it returns same value for same input.

They must not have any random values, current date/time, global state (DOM, files, db etc) and no mutation of parameters.

* **Immutability:** Immutable means can’t change. In JavaScript objects and arrays are not immutable. We can make them immutable (kind of) by declaring them as constants. We can still edit their values, but we can’t assign them to new objects.

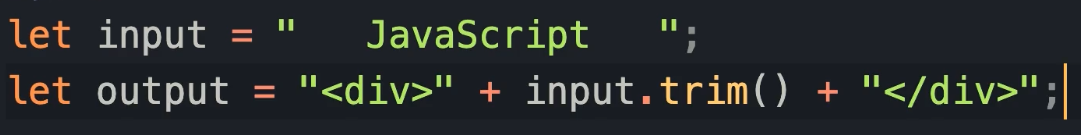
We can copy objects in JavaScript in two ways, using **spread operators** and using **object.assign**. One issue with this is, it makes a shallow copy (like both variables pointing to same memory location object). So, changing the updated one will change the original one as well **in case of nested objects**.

Example, first image, the person.city is changed too when we update the updated.city. We can solve this the way like the 2nd image.

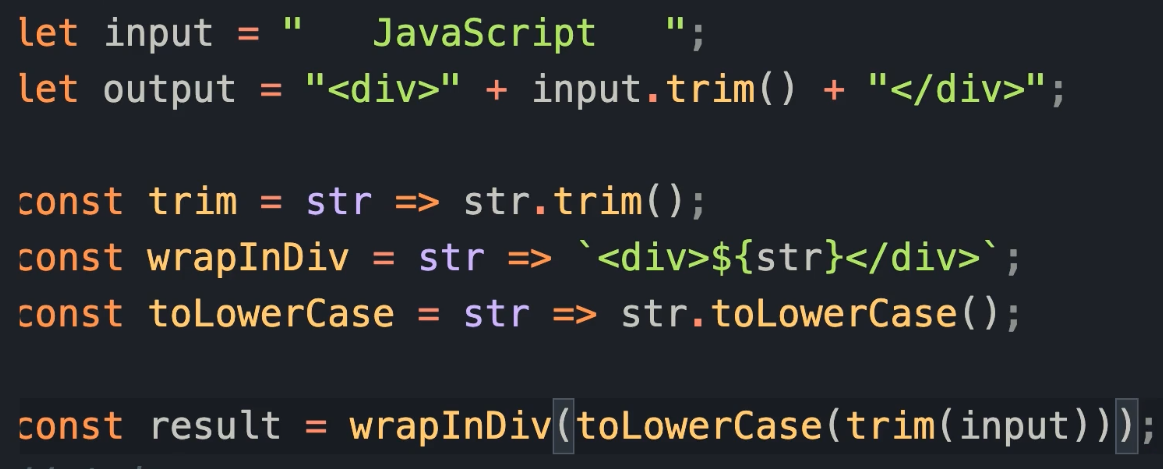


**Function Composition:** Sometimes we need to compose small functions to solve a real-life problem.

For example, this is non-functional way,



The functional way with composition is like this,



**Problem 1:** since we read it from right to left. The parenthesis can be a lot depending on our chain. Lodash/fp Library helps us out here with two functions **compose** and **pipe**.

* **Compose:** Compose **takes** bunch of functions as argument and **returns** a new function which is composition of all these functions. The composition happens from right to left (trim then toLowerCase then wrapInDive). Example of the above problem is below

****

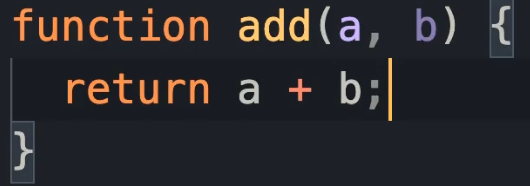
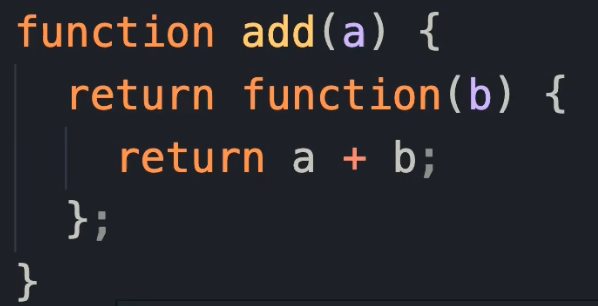
* **Pipe:** Similar to compose except order is from left to right.

****

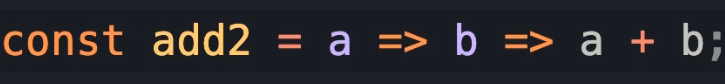
**Problem 2:** In case we need more than one argument in the functions we composite, we will run into problem. For example, if the wrap function took two arguments like this.



**Currying:** The solution is currying technique.

is same as  it is called like 

If we convert them in arrow functions,

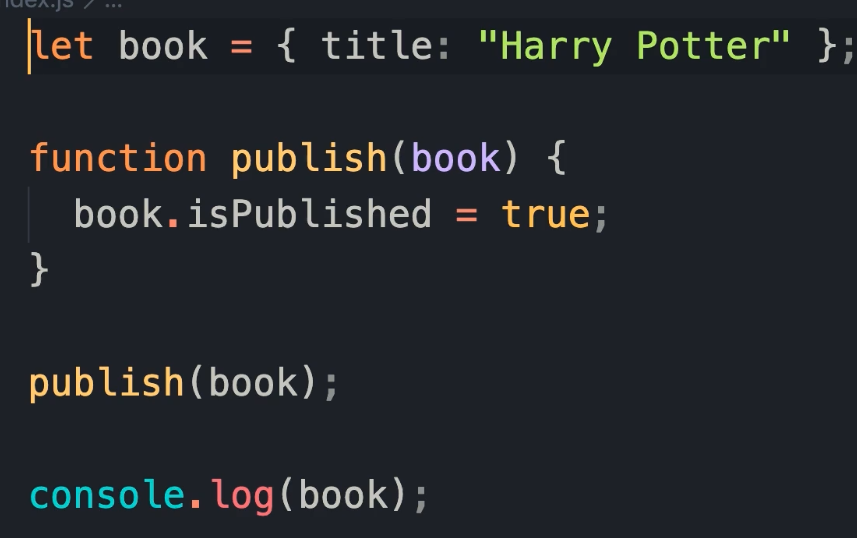
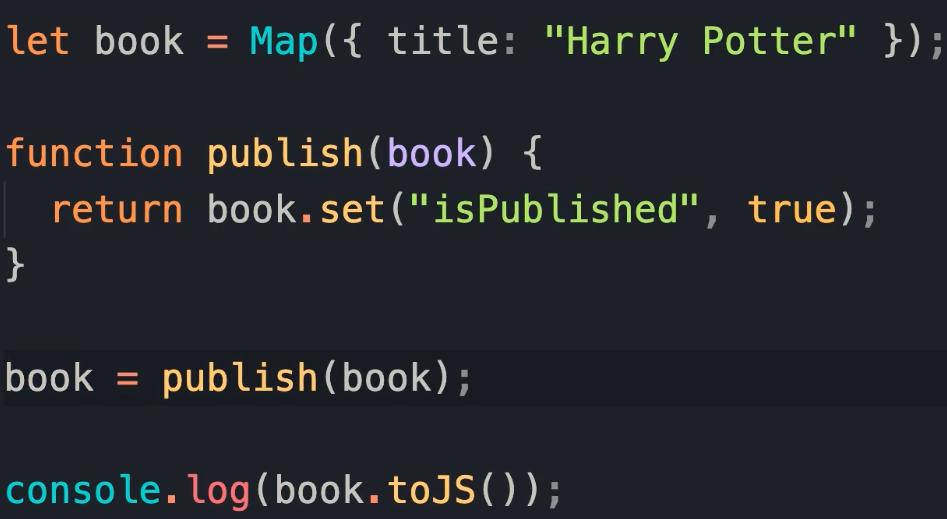


So, the solution to our problem of the wrap can be solved by using currying,



**Problem 3:** As we saw in the nested object copy section during shallow copy (in first page), the more nested objects we have more messed up our code will be. We have libraries to solve this problem - Immutable, Immer, Mori and others.

* **Immutable.js:** This library has bunch of defined data structures that are immutable and can be used to work as immutable objects. However, each time we use this, it returns an immutable object and we have to convert it into JavaScript object. Which is why at this point I’ll stick to Immer library.

Using Immutable.js => 

* **Immer.js:** This library has produce function which does the work for us, we don’t have convert from immutable to mutable JavaScript object. Example is below, Here the produce function returns a new copy of book with the modification we define in the function.

