Pure and impure functions

A basic principle of functional programming is that it avoids changing the application state (statelessness) and variables outside its scope (immutability).

**Pure Function**

Pure functions are a pillar of [functional programming](http://en.wikipedia.org/wiki/Functional_programming).

Pure functions are functions that:

* with a given input, always return the same output.
* they do not rely on any variable outside their scope
* they do not modify the state of the application (no side effects)
* Pure functions must not change any variable outside their scope

A pure function **doesn’t depend on** and **doesn’t modify** the states of variables out of its scope.

Pure functions do not have any observable side effects, such as network or database calls. The pure functions just calculate the new value. You can be confident that if you call the pure function with the same set of arguments, you're going to get the same returned value. They are **predictable**.

Libraries such as [ReactJS](https://facebook.github.io/react/docs/components-and-props.html) and [Redux](http://redux.js.org/docs/introduction/ThreePrinciples.html) require the use of pure functions

**What Are Observable Side Effects?**

An observable side effect is **any** interaction with the outside world from within a function. That could be anything from changing a variable that exists outside the function, to calling another method from within a function.

**Note:** If a pure function calls a pure function this isn’t a side effect and the calling function is still pure.

Side effects include, but are not limited to:

* Making a HTTP request
* Mutating data
* Printing to a screen or console
* DOM Query/Manipulation
* Math.random()
* Getting the current time

Example of pure function:

function priceAfterTax(productPrice) {  
return (productPrice \* 0.20) + productPrice;  
}

**Impure Function**

On the opposite, impure functions may call the database or the network, they may have side effects, they may operate on the DOM, and **they may override the values that you pass to them**. This is going to be an important distinction because some of the functions that you're going to write in Redux have to be pure, and you need to be mindful of that.

Impure function Example:

let a = 1;

let impure = (input) => {

// Multiply with variable outside function scope

let output = input \* a;

return output;

}

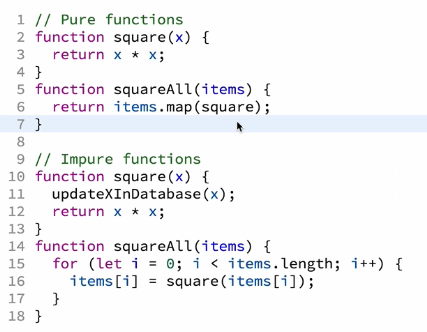
console.log(impure(2)) // Logs 2

a++; // a becomes equal to 2

console.log(impure(2)) // Logs 4

This impure function rely on variable a that is defined outside its scope. So, if a is modified, impure's function result will be different.

Example:



Advantages of pure functions

1. The main advantage of a pure function is that it **doesn’t have any side effect**. It doesn’t modify the state of the system outside of their scope. Then, they just simplify and clarify the code: when you call a pure function, you just need to focus on the return value as you know you didn’t broke anything elsewhere doing so.
2. A pure function is also robust. Its order of execution **doesn’t have any impact on the system**. Operations with pure functions could be parallelized.
3. Also, it’s very **easy to unit test** a pure function since there is no context to consider. Just focus on inputs / outputs.
4. Finally, maximizing the use of pure functions **makes your code simpler, more flexible**.

“Idea here is simple:

**simplify your code by limiting the number of functions that has an impact on the system**.”