Functional Programming: Side Effects

Author: *Mohammed shahid*

Functional programming is based on the simple premise that your functions should not have side effects; they are considered evil in this paradigm. If a function has side effects we call it a procedure, so functions do not have side effects.

What is side Effects ?

Any operation which modifies the state of the computer or which interacts with the outside world is said to have a side effect.

1. Changing the value of a variable;
2. Writing some data to disk;
3. Enabling or disabling a button in the User Interface.

No Side Effects Example:

```
int square(int x)
{
return x * x;
}
```

For example, this function has no side effects. Its result depends only on its input arguments, and nothing about the state of the program or its environment changes when it is called:

Side Effects Example:

```
int x = 0;

int next_x()
{
return x++,x;
}

void set_x(int new_x)
{
x = new_x;
}
```

calling these functions will give you different results depending upon the order in which you call them, because they change something about the state of the computer.

```
int Write(const char* s)
{
return printf("Output: %s\n", s);
}
```

This function has the side effect of writing data to output. You don't call the function because you want its return value; you call it because you want the effect that it has on the "outside world":

Note: Every operation actually makes changes to the computer's state (CPU registers, memory, power consumption, heat, etc.). "Side effect" usually refers to an imaginary idealized execution environment where nothing about that environment changes unless the program explicitly changes it. But if you are calling square(x) because you want that external computer state to change, you could consider that to be a side effect.

A function, in the mathematical sense, is a mapping from input to output. The intended effect of calling a function is for it to map the input to the output it returns. If the function does anything else, it doesn't matter what, but if it has any behavior that is not mapping the input to the output, that behavior is known to be a side effect.

A side-effect is when an operation has an effect on a variable/object that is outside the intended usage.

An effect is anything that affects an actor. If I call a function that sends my girlfriend a breakup text message,that affects a bunch of actors, me, her, the cell phone company's network, etc. The only intended effect of calling a side-effect free function, is for the function to return me a mapping from my input. So for:

```
public void SendBreakupTextMessage() {
   Messaging.send("I'm breaking up with you!")
  }
```

If this is intended to be a function, then the only thing it should do is return void. If it was side-effect free, it should not actually send the text message.

so when someone says this function has side-effect, they effectively mean, this construct does not behave like a mathematical function. And when someone says this function is side-effect free, they mean, this construct effectively behaves like a mathematical function.

Pure Functions

A function that returns always the same result for the same input is called a pure function.

A pure function, therefore, is a function with no observable side effects, if there are any side effects on a function the evaluation could return different results even if we invoke it with the same arguments. We can

substitute a pure function with its calculated value.

Pure Function Example:Python

```
def add(x,y):
    return x+y
```

A pure function is always side effect free, by definition. A pure function, is a way to say, this function, even though it's using a construct that allows more effects, only has as effect an equal one to that of a mathematical function.

Idempotence/Idempotent

> *Idempotence* means calling this function many times with the same inputs (doesn't matter where they come from) will always result in the same effects (side effect or not).

Keynotes

- Pure functions shouldn't modify anything outside of the function.
- Pure functions should always give the same *output* to the same *input(idempotence)**.

> ***f(x,y) = x+y***

- Pure functions are ***Anti Fragile***.
- Pure functions are also known as **Deterministic Functions** in ***Philosophy*** and ***Physics***.
- When a function has no **Side Effects** and **Deterministic** we call it **Pure Function**.

Pure functions are easier to,

- Read
- Debug
- Concurrency
- Test

So, why is this all important?

It's all about control and keeping it. If you call a function, and it does something else then return a value, it is hard to reason about its behavior. You will need to go look inside the function for the actual code to guess what it is doing and assert its correctness. The ideal situation, is that it is very clear and easy to know what is the input the function is using and that it isn't doing anything else then returning an output for it

Because the most helpful to understand, reason and control the behavior of a program is to have all input clearly grouped together and explicit, as well as have all side-effect be grouped together and explicit, this is generally what people talk about when they say side-effect, pure, etc.

Finally, I think a big problem with explaining side effects is that until you've used a language like Ocaml or Haskell,Elixir it can be very hard to reason about side-effect free (almost in any language!) programming.

Don't get me wrong, this paradigm ***(FP)*** doesn't try to get rid of side effects, just limit them when they are required. Side effects are needed because without them our programs will do only calculations. We often have to write to databases, integrate with external systems or write files

The reason why side effects are bad is because, if you had them, a function can be unpredictable depending on the state of the system ***(Non Deterministic Function)***. When a function has no side effects we can execute it anytime, it will always return the same result, given the same input.