Functional programming is declarative

Functional programming falls under the umbrella of declarative programming paradigms: it's a paradigm that expresses a set of operations without revealing how they're implemented or how data flows through them.

The more popular models used today, though, are imperative or procedural, and are supported in most structured and object-oriented languages like

- Java
- C#
- C++
- and others.

Imperative programming treats a computer program as merely a sequence of top-to-bottom statements that changes the state of the system in order to compute a result.

Let's look at a simple imperative example. Suppose you need to square all the numbers in an array. An imperative program follows these steps:

```
const array = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
  for(let i = 0; i < array.length; i++) {
  array[i] = Math.pow(array[i], 2);
}
array; //-> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Imperative programming tells the computer, in great detail, how to perform a certain task (looping through and applying the square formula to each number, in this case).

This is the most common way of writing this code and will most likely be your first approach to tackling this problem.

Declarative programming, on the other hand, separates program description from evaluation.

It focuses on the use of expressions to describe what the logic of a program is without necessarily specifying its control flow or state changes.

An example of declarative programming is found in SQL statements. SQL queries are composed of statements that describe what the outcome of a query should look like, abstracting the internal mechanism for data retrieval.

Later, we'll show an example of using a SQL-like overlay over your functional code to give meaning to both your application and the data that runs through it.

Shifting to a functional approach to tackle this same task, we only need to be concerned with applying the

right behavior at each element and give up control of looping to other parts of the system. You can let ***Array.map()*** do most of the heavy lifting:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9].map(
function(num) {
  return Math.pow(num, 2);
});
//-> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Map takes a functionn that computes the square of each number

Compared with the previous example, you see that this code frees you from the responsibility of properly managing a loop counter and array index access; put simply, the more code we have, the more places there are for bugs to occur.

Also, standard loops aren't reusable artifacts unless they're abstracted with functions. And that's precisely what we'll do. I'll share my knowledge on how to remove manual loops completely from our code in favor of first-class, higher-order functions like map, reduce, and filter, which accept functions as parameters so that our code is more reusable, extensible, and declarative. Previous months this year, this was an on going topic I had with Noah from the ***Storyline*** team. ***Storyline*** has good examples of re-usable artifacts in use.

Studying how the redux's compose and lodash's flow function works, lead us to what I did with the magical run function in S-1.1 and S-1.2.

Abstracting loops with functions lets you take advantage of lambda expressions or arrow functions, introduced in ES6 JavaScript. Lambda expressions provide a succinct alternative to anonymous functions that can be passed in as a function argument, in the spirit of writing less:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9].map(num => Math.pow(num, 2));
//-> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Translating lambda notation to regular function notation

Lambda expressions provide an enormous syntactical advantage over regular function notations because they reduce the structure of a function call down to the most important pieces.

This ES6 lambda expression

```
num => Math.pow(num, 2);
```

is equivalent to the following function:

```
function(num) {
  return Math.pow(num, 2);
}
```

Why remove loops from your code?

A loop is an imperative control structure that’s hard to reuse and difficult to plug in to other operations.

In addition, it implies code that’s constantly changing or mutating in response to new iterations. We need to understand that functional programs aim for statelessness and immutability as much as possible.

Stateless code has zero chance of changing or breaking global state. To achieve this, we'll use functions that avoid side effects and changes of state, known as pure functions.