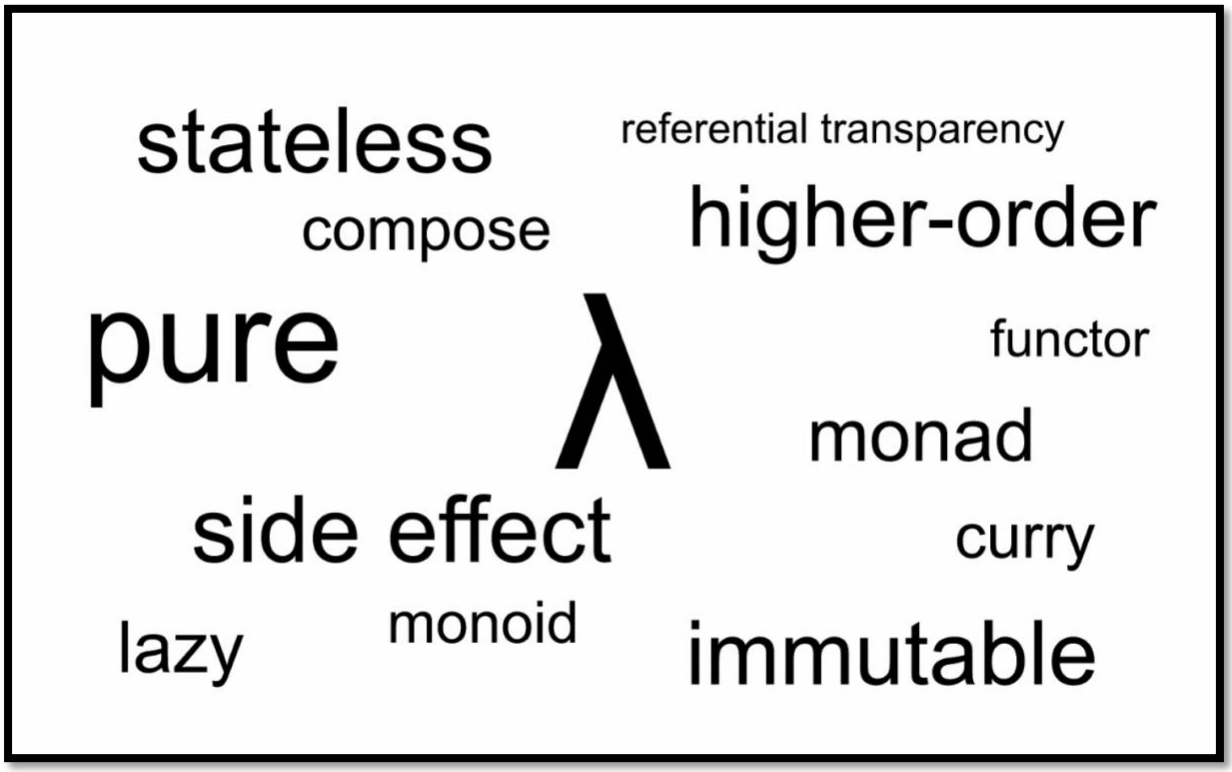
Functional Programming



Functional programming is classified as a **declarative** paradigm where the program description is separated from the calculations. The emphasis here is on the use of expressions to describe program logic. This is the opposite of **imperative** programming, where the code is executed step by step and tells the computer in detail how to get the job done.

Declarative vs Imperative

Imperative

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];

const results = []

for(let i = 0; i < numbers.length; i++)
{
    const secondPower = Math.pow(numbers[i], 2)

    if(secondPower & 1)
    {
        // or % 2 but operations on bits are faster
        results.push(secondPower);
    }
}

console.log(results) // [1, 9, 25, 49, 81]
```

As for the **imperative** solution, the focus on implementation details is clearly visible. In the loop, you can see the array index based on the need to control the number of elements. Due to the large number of details in the code, it is harder to focus on what it is doing.

Declarative

```
const risesToSecondPower = (num) => Math.pow(num, 2)
const isOdd = (num) => num & 1;

const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

const results = numbers.map(risesToSecondPower).filter(isOdd);

console.log(results) // [1, 9, 25, 49, 81]
```

In this solution, the implementing has been separated from the invocation by taking the logic to separate functions. Thanks to this solution, we can only focus on the names of functions that describe what is happening in them. Additionally, the level of abstraction has been raised and the logic now can be reusable. Now, let's focus on the call. You can't see any details in it, just a description that tells you what this code does, step by step:

1. map(risesToSecondPower) - take each element of an array and raise it to the second power,
2. filter(isOdd) - filter and select odd elements.

Benefits

Functional programming has many benefits. When it comes to **JavaScript**, the use of functions is natural because it is a functional language. Even the classes in this language are "syntactic sugar" and they are underneath composed of functions.

Readability

When it comes to readability, in the imperative approach, the code usually becomes a list with names of functions that can be read sequentially without delving into their logic. As a result, we do not focus on the implementation details.

Immutability

Another advantage is sticking to the convention of immutable objects. Thanks to this approach, the code becomes safer because the references in **JavaScript** are very strong and it is easy to modify the unwanted object.

Abstraction

In functional programming, the code is broken down into small functions that can easily be thought of as reusable abstract code.

Safer

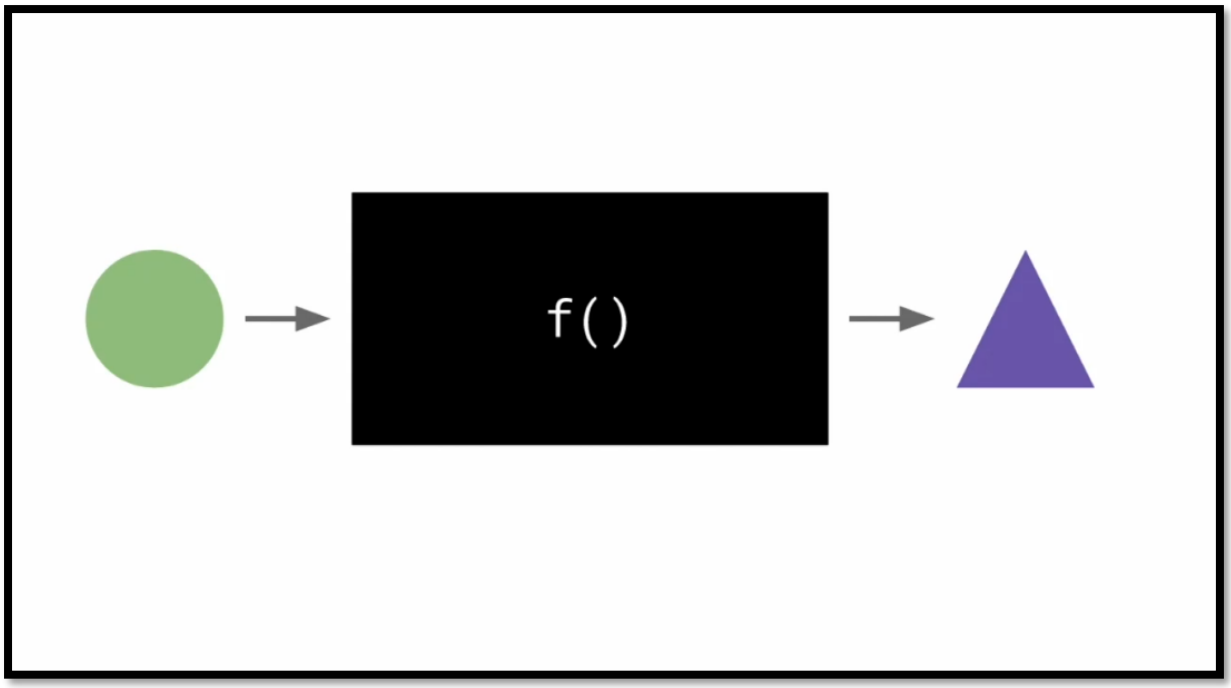
When functions are deterministic, they are more predictable and safer.

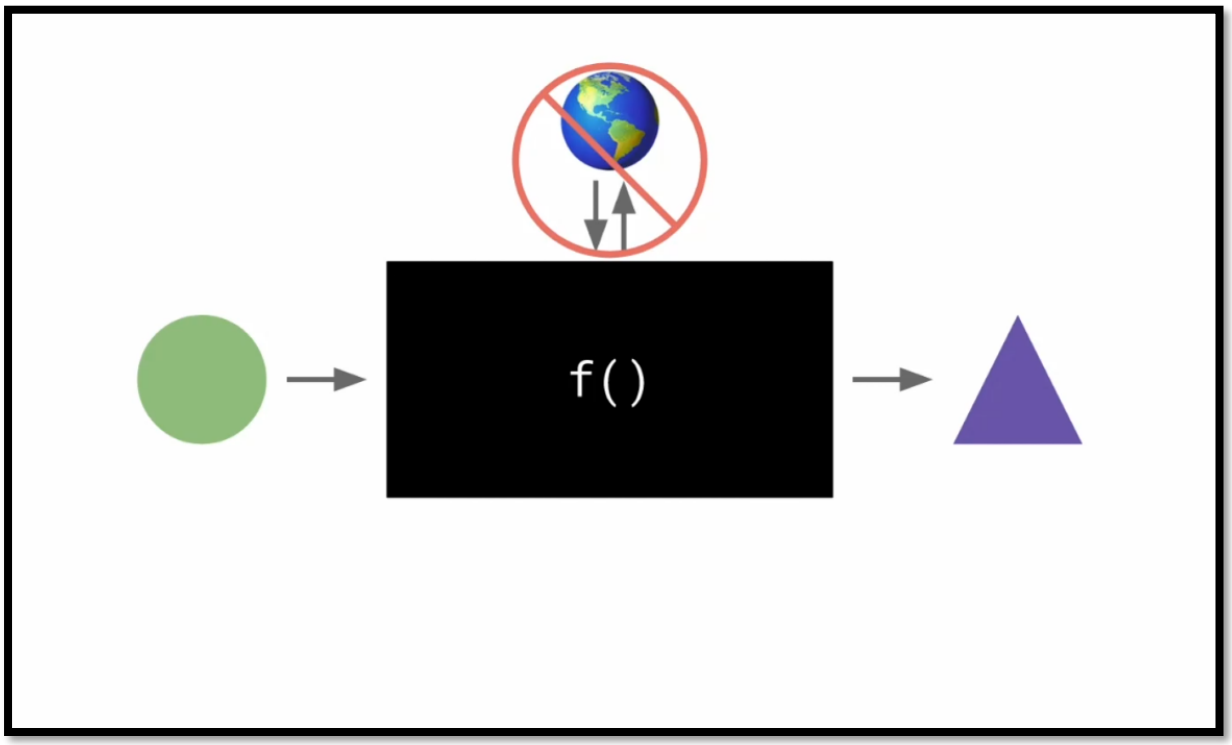
Easier to debug.

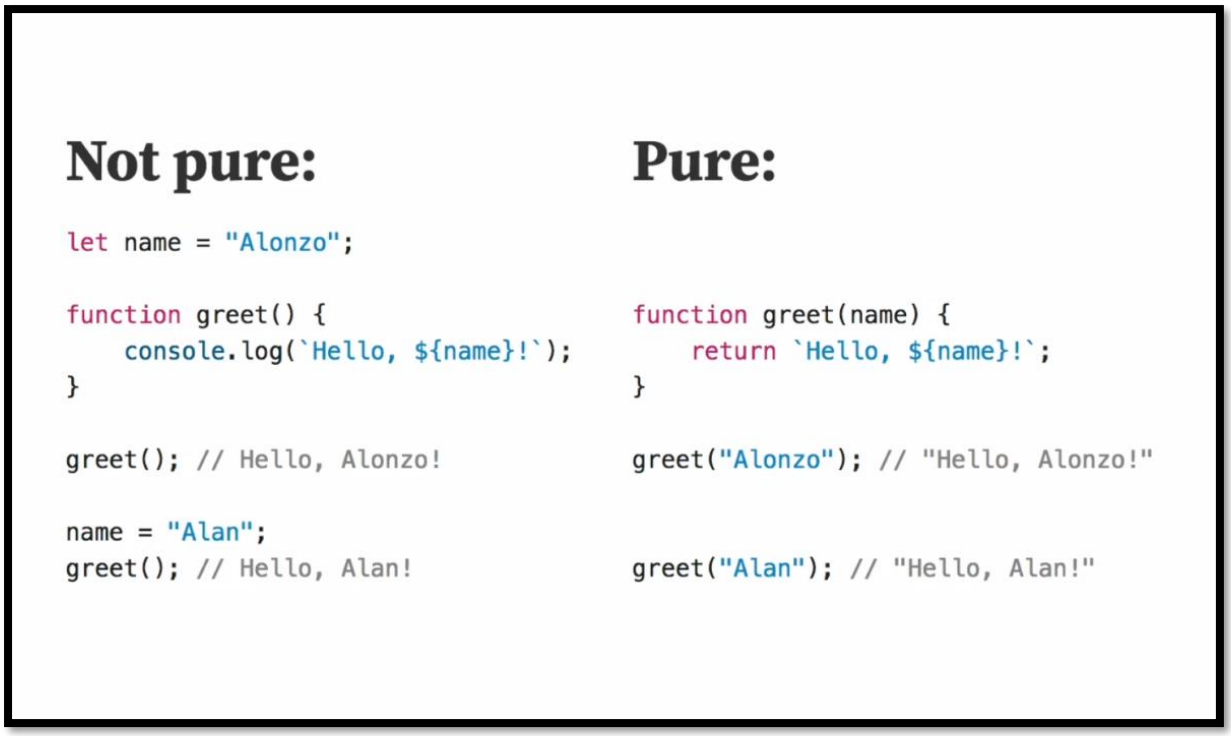
Pure Functions

One of the important considerations in functional programming is pure functions. To create such a function, you need to remember a few rules:

1. The result that the function returns depends only on the input parameters,
2. Do not use global variables and variables that are beyond your own range,
3. Do not change the state of the parameters,
4. Pure functions do not have any "side-effects" (usually modifying object properties),
5. For the indicated input parameters, they will always return one and the same result,
6. A pure function always takes a parameter and always returns a parameter.


f()


f()



In pure functions if we pass the same argument, we will get the same output because pure functions are not dependent on outside world. But in not pure functions, we might get two different values. Because it is very probable that someone might come and change the global variables.

So pure functions are deterministic and their output is totally determined by their input.

```
Impure function
let counter = 5;

const multipleCounter = (multiplier) =>
{
   counter = counter * multiplier
}

multiplyCounter(2) // -> ? the result depends on the initial value

Pure function
const multiplyBy = (multiplier) => (value) => value * multiplier
const multipleByTwo = multiplyBy(2)

const counter = multiplyByTwo(5) // -> 10
```

The first function is unpredictable because it depends on an external parameter that can change. The second function is transparent, it depends only on input parameters, does not modify them, and does not use out-of-range variables. It is transparent because it depends on parameters, does not modify them, does not use variables outside the range, and returns a new value.

Side Effects

Imperative:

```
let name = "Alonzo";
let greeting = "Hi";

console.log(`${greeting}, ${name}!`);
// Hi, Alonzo!

greeting = "Howdy";
console.log(`${greeting}, ${name}!`);
// Howdy, Alonzo!
```

Functional:

```
function greet(greeting, name) {
  return `${greeting}, ${name}!`;
}

greet("Hi", "Alonzo");
// "Hi, Alonzo!"

greet("Howdy", "Alan");
// "Howdy, Alan!
```

Side effects:

```
let thesis = {name: "Church's", date: 1936};

function renameThesis(newName) {
    thesis.name = newName;
    console.log("Renamed!");
}

renameThesis("Church-Turing"); // Renamed!
thesis; //{name: "Church-Turing", date: 1936}
```

1. Logging to console.
2. Global variable thesis.

No side effects:

```
const thesis = {name: "Church's", date: 1936};

function renameThesis(oldThesis, newName) {
  return {
    name: newName, date: oldThesis.date
  }
}

const thesis2 = renameThesis(thesis, "Church-Turing");
thesis; // {name: "Church's", date: 1936}
thesis2; // {name: "Church-Turing", date: 1936}
```