Imperative vs Declarative Programming in JavaScript

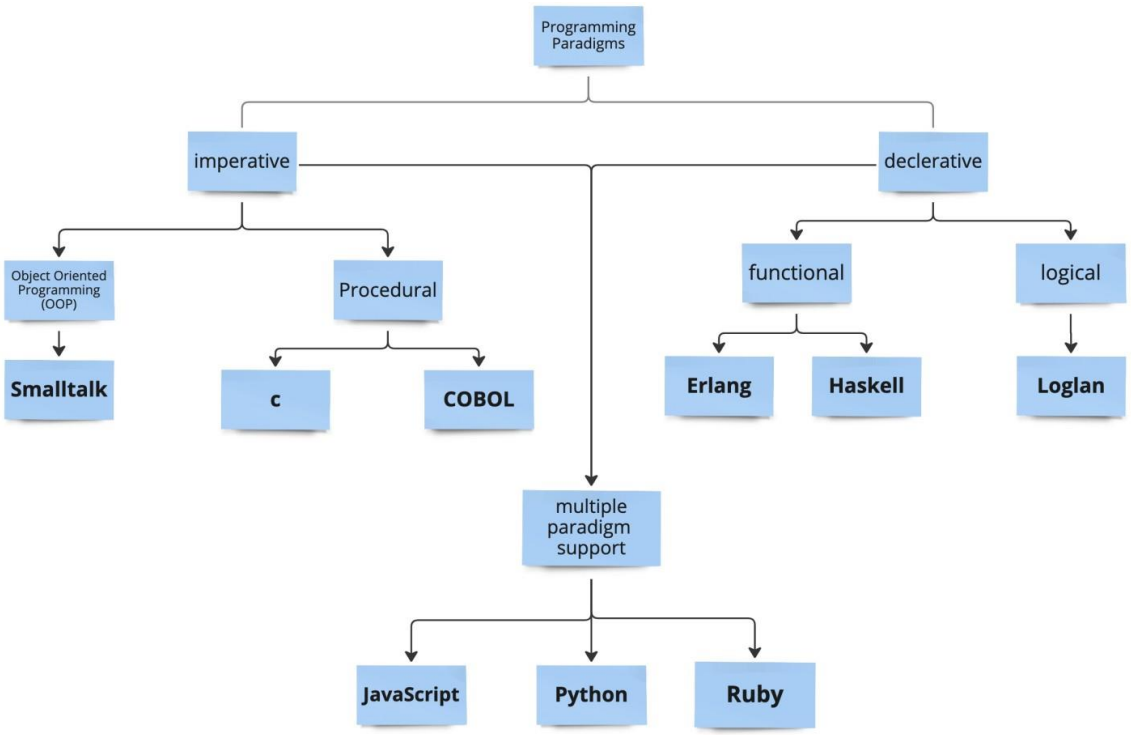
Programming paradigms

JavaScript is an outstanding programming language that allows you to write code in different paradigms or combine them into a 'multi-paradigmatic' approach.

A programming paradigm can be termed as an approach or a style of writing code. Most of the modern programming languages fall into two general paradigms: **imperative** (procedural, OOP etc.) and **declarative**(functional etc.).

Popular examples of declarative languages are database query languages such as SQL, markup languages like HTML and CSS, and *functional* and *logic* programming languages.

Although Some languages are designed to support one paradigm, Many modern programming languages are designed to be versatile and flexible, and thus support multiple paradigms. For example, Java is an object-oriented programming language that also supports procedural programming and functional programming constructs. Similarly, Python is a multi-paradigm language that supports procedural, object-oriented, and functional programming styles.



The easiest way to explain the difference between declarative and imperative code, would be that ***imperative* code focuses on writing an explicit sequence of commands to describe *how* you want the computer to do things, and declarative code focuses on specifying the result of *what* you want.**

This explanation may sound a little confusing at first, but with a super easy example it'll become much clearer.

This would be the implementation of taking an array of numbers as an input and returning a new array with each number multiplied by 2 in both declarative and imperative ways:

```
//IMPERATIVE
const doubleMapImperative = numbers => {
    const doubled = [];
    for (let i = 0; i < numbers.length; i++) {
        doubled.push(numbers[i] * 2);
    }
    return doubled;
};

//DECLARATIVE
const doubleMapDeclarative = numbers => numbers.map(n => n * 2);


console.log(doubleMapImperative( numbers: [2, 3, 4])); // [4, 6, 8]
console.log(doubleMapDeclarative( numbers: [2, 3, 4])); // [4, 6, 8]
```

As you can see, the declarative mapping does the same thing, but abstracts the flow control away using the functional **Array.prototype.map()** utility, which allows you to express the flow of data more clearly without any comparisons or state keeping (even though the map() method does the state-keeping itself). This is enabled due to JavaScript's support for Higher-Order functions and lambda expressions.

Generally, built in Object and Array methods (map, filter, reduce etc.) are becoming more and more popular in modern JavaScript, because generic utilities that act on many data types are quite suitable for a weakly typed language like JavaScript.

Imperative vs Declarative control flow

Imperative procedural and object-oriented languages use reserved words that act on blocks, such as *if*, *while*, and *for*, to implement control flow. That replaced the 'goto' statements and 'branch tables' from Assembly.

If I would have a task to go through an array of vegetables and return an array with all peppers, imperatively I could write it by using this common approach:

```
const myVeggies = ['potatoes', 'peppers','hot-peppers'];

const getPeppers = function () {
    let veggies = [];
    for(let i = 0; i < myVeggies.length; i++) {
        if(myVeggies[i].toLowerCase().includes('peppers')) {
            veggies.push(myVeggies[i]);
        }
    }
    return veggies;
}

getPeppers();
// => ['peppers','hot-peppers']
```

Now lets see what that would look like if we were to use a more declarative approach:

```
const myVeggies = ['potatoes', 'peppers','hot-peppers'];
const includesPeppers =veggie=>veggie.toLowerCase().includes('peppers')
const getPeppers=veggies=>veggies.filter(includesPeppers)
getPeppers(myVeggies);
// => ['peppers','hot-peppers']
```

Pros and Cons

Imperative:

pros

- The syntax is easier to grasp for programmers coming from most languages.

- Using the imperative code flow enables more control over what occurs in the blocks, including stopping the loops when needed.
- Using imperative code *can* work a little quicker sometimes according to this performance analysis.

cons:

- The code is usually longer.
- Code is harder to read.

Declarative:

pros:

- The code is usually shorter.
- Code is clearer and easier to read.
- Declarative code is very appropriate for many JavaScript frameworks and state management systems such as React and Redux.

cons:

- Code may work a little slower sometimes (this would not be a problem when dealing with smaller datasets)