**JavaScript Functional programming**

Functional programming is a subset of the Declarative programming paradigm.

Functional programming technique makes our code more concise, readable, and predictable. It is easy to test and maintain code developed through functional programming.

Functional programming involves crucial concepts such as immutable states, referential transparency, method references, and high-order, and pure functions. It involves programming techniques such as functional composition, recursion, currying and functional interfaces.

**Imperative vs declarative**

Imperative programming specifies and directs the control flow of the program. Declarative programming specifies the expected result and core logic without directing the program's control flow.

**-> imperative**

// print some of array values

var arr = [10,30,50,60,90];

var total= 0;

for(var i=0;i<arr.size;i++){

total = total+ arr[i];

}

**-> declarative**

var arr = [10,30,50,60,90];

()=>{} //

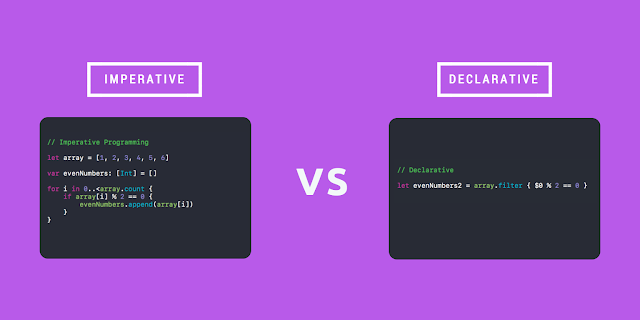
function(){

}

var total = arr.reduce((acc, ele)=>{

acc = acc+ ele;

},0);



Here are some key advantages of functional programming:

1. **Predictable and Reliable Code:**

Functional programming emphasizes immutability and pure functions. This leads to more predictable code behavior since pure functions produce the same output for the same input, and they don't have side effects. This predictability makes it easier to reason about and test your code.

1. **Modular and Reusable Code:**

Functional programming encourages breaking down your code into smaller, composable functions. These functions can be easily reused in different parts of your application, leading to a more modular and maintainable codebase.

1. **Reduced Side Effects:**

By minimizing mutable state and side effects, functional programming reduces the chances of unexpected interactions between different parts of your code. This can lead to fewer bugs and easier debugging.

1. **Parallel and Concurrent Programming:**

Pure functions can be safely executed in parallel since they don't depend on shared state. This can lead to improved performance, especially in multi-core environments.

1. **Easier Testing:**

Pure functions are easy to test because they don't rely on external state or context. You can test them in isolation, making unit testing more straightforward and effective.

1. **Better Code Maintainability:**

Functional programming promotes declarative code, which describes what the code should do rather than how to do it. This makes the code easier to read and understand, enhancing its maintainability.

1. **Code Clarity and Expressiveness:**

Functional programming emphasizes higher-order functions, function composition, and other techniques that can lead to more concise and expressive code. This makes the intent of the code clearer and reduces unnecessary boilerplate.

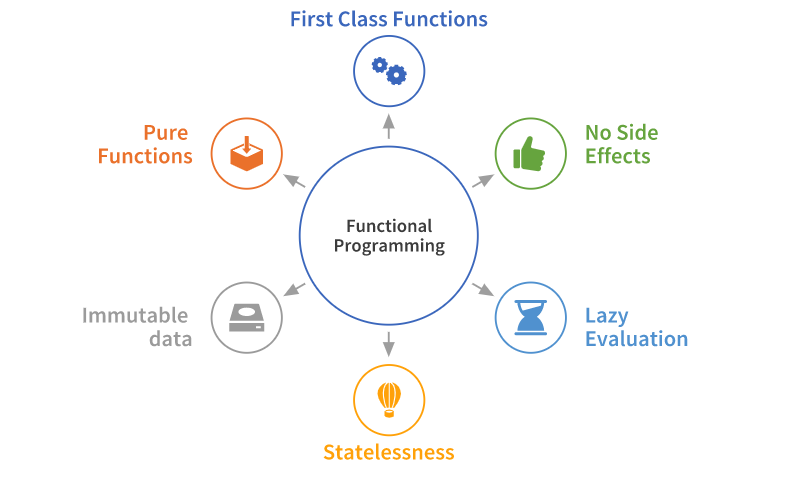
1. **Avoidance of Callback Hell:**

Functional programming techniques, such as using Promises or async/await, help avoid callback hell—a situation where nested callbacks become hard to manage and understand.

1. **Easier Debugging and Troubleshooting:**

Since functional programming minimizes side effects and state changes, it's often easier to locate the source of bugs and unexpected behavior. Debugging can be more straightforward due to the limited scope of functions.

Here's an overview of some key concepts in functional programming using JavaScript:



1. **Higher-Order Functions:**

Higher-order functions are functions that take one or more functions as arguments or return a function as their result. They allow for the composition of functions and help in abstracting common patterns.

// Callback function, passed as a parameter in the higher order function

function callbackFunction(){

console.log('I am a callback function');

}

// higher order function

function higherOrderFunction(func){

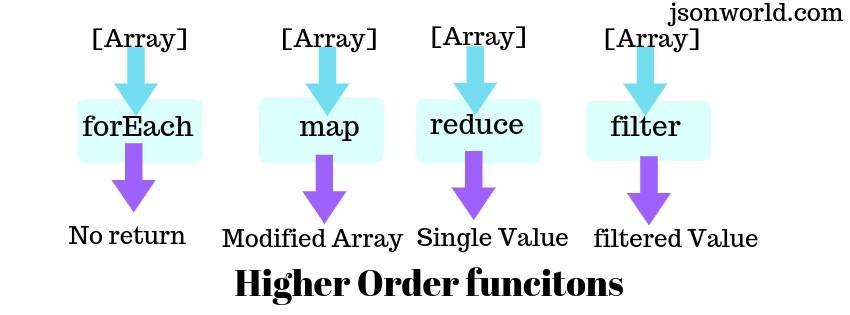
console.log('I am higher order function')

func()

}

higherOrderFunction(callbackFunction);

Few Examples:



1. **Pure Functions:**

A pure function in JavaScript is a function that returns the same result if the same arguments(input) are passed in the function. Let's see what makes a function pure in detail:

The return value of the function on the function call should only be dependent on the input function arguments.

It should not modify any non-local state. It means the function should not manipulate anything other than the data stored in the local variables declared within the function.

The function should not have any side effects, such as reassigning non-local variables, mutating the state of any part of code that is not inside the function, or calling any non-pure functions inside it.

A function that follows all the above conditions is a pure function in javascript.

// A pure function adding two integers passed in it.

function operationAdd(a, b){

return a+b;

}

// Pure function to divide two integers passed in it.

function operationDivide(a, b){

return a/b;

}

// Pure function to multiple two integers passed in it.

function operationMulti(a, b){

return a\*b;

}

// Calling all the pure functions

console.log(

operationAdd(2,5),

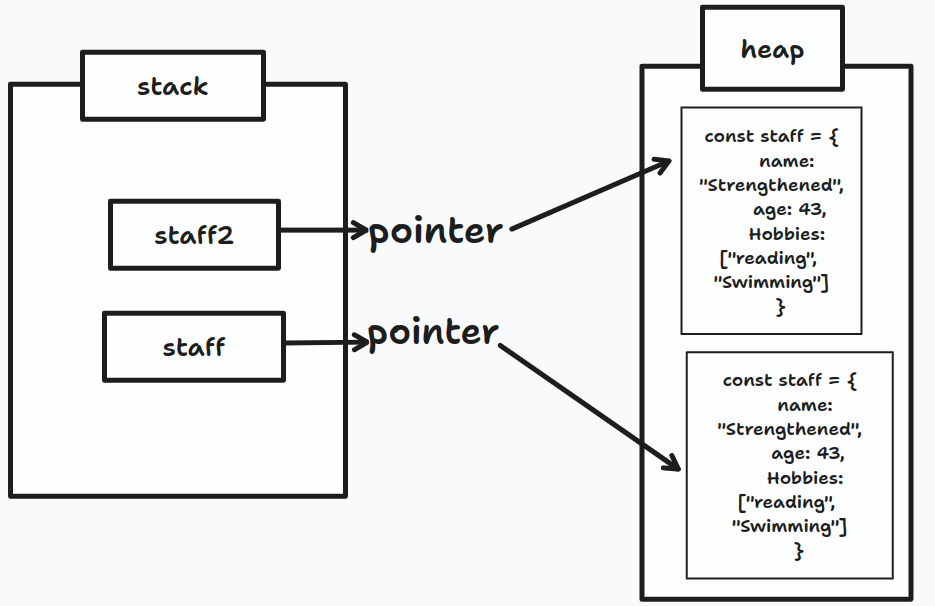
operationMulti(3,2),

operationDivide(20,5)

);

1. **Immutability:**

Immutability refers to the practice of not changing the state of data once it's created. Instead of modifying data in-place, you create new instances with the desired changes. This leads to safer and more predictable code.



1. **Function Composition:**

Function composition is the practice of combining multiple functions to create more complex functions. It's a key concept in functional programming for building modular and reusable code.

const add = (x) => x + 2;

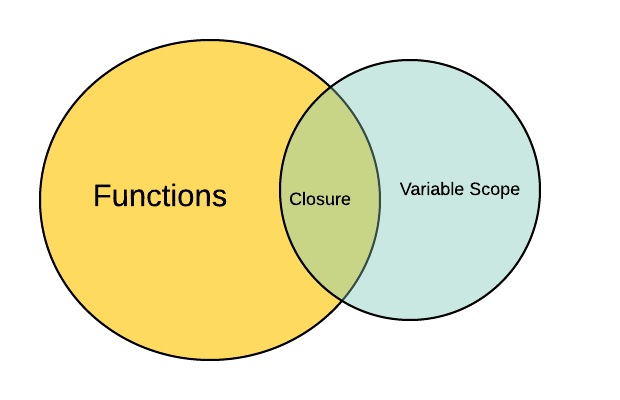
const multiply = (x) => x \* 3;

const subtract = (x) => x - 1;

const composedFunction = (x) => subtract(multiply(add(x)));

1. **Closure:**

Closures allow functions to remember the scope in which they were created, even after that scope has finished executing. They are useful for creating private variables and data encapsulation.



function outerFunction(number) {

let initial = 1;

function innerFunction(newNumber) {

initial = initial \* newNumber+number

return initial;

}

return innerFunction;

}

let returnedFunction = outerFunction(2); // returnedFunction here is same as innerFunction.

console.log(returnedFunction(1)); // 3

console.log(returnedFunction(3)) // 11

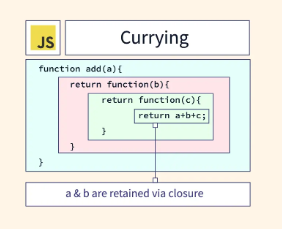
1. **Currying:**

Currying involves transforming a function that takes multiple arguments into a series of functions that each take a single argument.

Currying takes a function that receives more than one parameter and breaks it into a series of unray (one parameter) functions. Hence, the currying function takes only one parameter at a time.

**Uses of Currying Function**

* Currying in JavaScript can be for the following reasons:
* Currying is helpful in Event handling.
* By using the currying function, we can avoid passing the same variable many times.
* Currying in JavaScript can be used to make a higher-order function.



function sum(x) { // x is taken as argument here

return (y) => {

return (z) => { // this function will return the sum

return x + y + z

}

}

}

console.log(sum(1)(2)(3)) // 6