# Programming

A programming paradigm is a way to classify a programming language based on some features. Basically, a programming paradigm is a way to write programs.

Some languages use only 1 paradigm, while others can use multiple paradigms. For example, Haskell uses a single paradigm: functional programming. JavaScript uses multiple paradigms, as we can write programs using procedural programming, "object-oriented" programming (OOP for short), functional programming, or even reactive programming.

For single paradigm languages, since there's only 1 way of writing programs, it's easy to "stay on the track". For multi-paradigms languages, it's harder to stay on the same track, as the frontier between 2 paradigms depends solely on the developer writing the program.

**Types of programming languages**

1. **Procedural programming languages:**

A procedural language follows a sequence of statements or commands in order to achieve a desired output. Each series of steps is called a procedure, and a program written in one of these languages will have one or more procedures within it. Example - C and C++, Java, Pascal, BASIC.

1. **Functional programming languages:**

Rather than focusing on the execution of statements, functional languages focus on the output of mathematical functions and evaluations. Each function–a reusable module of code–performs a specific task and returns a result. The result will vary depending on what data you input into the function. Example – Scala, Erlang, Haskell, Elixir, F#

1. **Object-oriented programming languages:**

This type of language treats a program as a group of objects composed of data and program elements, known as attributes and methods. Objects can be reused within a program or in other programs. This makes it a popular language type for complex programs, as code is easier to reuse and scale. Example – Java, Python, PHP, C++, Ruby.

1. **Scripting languages:**

Programmers use scripting languages to automate repetitive tasks, manage dynamic web content, or support processes in larger applications. Example- PHP, Ruby, Python, bash

1. **Logic programming languages:**

Instead of telling a computer what to do, a logic programming language expresses a series of facts and rules to instruct the computer on how to make decisions. Example- Prolog, Absys, Data log.

# Functional Programming

**Side effect (computer science):**

In computer science, an operation, function, or expression is said to have a side effect if it modifies some state variable value(s) outside its local environment, which is to say if it has any observable effect other than its primary effect of returning a value to the invoker of the operation. Example side effects include modifying a non-local variable, modifying a static local variable, modifying a mutable argument passed by reference, performing I/O or calling other functions with side-effects.

**What is Functional Programming?**

Functional programming is a paradigm through which developers write programs using a combination of pure functions, which are developed in such a way that they don’t have side effects (more on that later). Functional languages emphasis on expressions and declarations rather than execution of statements. Therefore, unlike other procedures which depend on a local or global state, value output in FP depends only on the arguments passed to the function.

**Characteristics of Functional Programming:**

1. Functional programming method focuses on results, not the process.
2. Emphasis is on what is to be computed.
3. Data is immutable.
4. Functional programming Decompose the problem into ‘functions.
5. It is built on the concept of mathematical functions which uses conditional expressions and recursion to do perform the calculation.
6. It does not support iteration like loop statements and conditional statements like If-Else.

**History of Functional programming:**

1. The foundation for Functional Programming is Lambda Calculus. It was developed in the 1930s for the functional application, definition, and recursion
2. LISP was the first functional programming language. McCarthy designed it in 1960
3. In the late 70’s researchers at the University of Edinburgh defined the ML(Meta Language)
4. In the early 80’s Hope language adds algebraic data types for recursion and equational reasoning
5. In the year 2004 Innovation of Functional language ‘Scala.’

**Type of functional language :**

1. **Pure Functional Languages** − These types of functional languages support only the functional paradigms. For example − Haskell.
2. **Impure Functional Languages** − These types of functional languages support the functional paradigms and imperative style programming. For example − LISP.

## Pure function

Pure functions have two important properties:

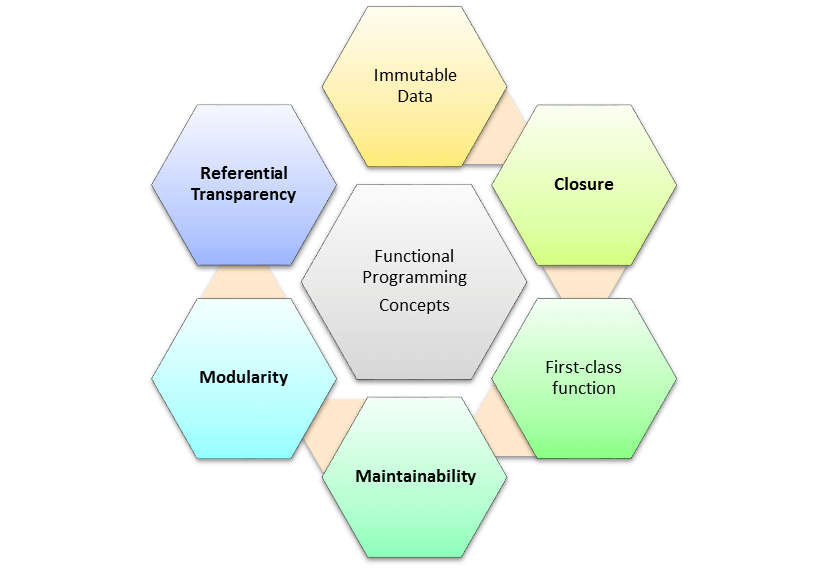
1. They always produce the same output with the same arguments irrespective of other factors.
2. They are deterministic. Pure functions either give some output or modify any argument or global variables i.e., they have no side effects.

Because pure functions have no side effects or hidden I/O, programs built using functional paradigms are easy to debug. Moreover, pure functions make writing concurrent applications easier.

When the code is written using the functional programming style, a capable compiler can:

1. Memorize the results
2. Parallelize the instructions
3. Wait for evaluating results

## Basic Functional Programming Terminology and Concepts



1. **First-Class Functions:** A first-class citizen is an entity of the language that supports the following operations:
2. Storing the entity in a variable
3. Passing the entity as an argument of a function
4. Returning the entity when calling a function

JavaScript’s functions are first-class citizens-

1. They can be stored in variables.
2. They be passed as arguments of other functions.
3. They be returned when calling other functions.

Therefore, functions are first-class citizens in JavaScript, meaning we can write FP programs using this language.

1. **Higher-Order Functions:**
2. Higher-order functions take functions as an argument or return the functions.
3. Higher-order functions are also essential for building programs in functional programming.
4. Higher-order functions allow partial applications or currying.
5. This technique applies a function to its arguments one at a time, as each application returning a new function which accepts the next argument.

Code:

Function show(a,b){

Return function(){

Console.log(a + b);

}

}

1. **Variables are Immutable:**
2. In functional programming, we can’t modify a variable after it’s been initialized.
3. We can create new variables – but we can’t modify existing variables, and this really helps to maintain state throughout the runtime of a program.
4. Once we create a variable and set its value, we can have full confidence knowing that the value of that variable will never change.
5. **Recursion:** There are no “for” or “while” loop in functional languages. Iteration in functional languages is implemented through recursion. Recursive functions repeatedly call themselves, until it reaches the base case.   
   example of the recursive function:

fib(n){

if (n <= 1)

return 1;

else

return fib(n - 1) + fib(n - 2);

}

**Function Composition:** Function composition allows you to combine pure functions to create more complicated ones.

Code:

Const trim = str => str.trim();

Const wrap = str => `<div> ${str} </div>`;

Const result = wrap(trim(“ .. shuvo … “));

**Referential Transparency:** Functional programs should perform operations just like as if it is for the first time. So, you will know what may or may not have happened during the program’s execution, and its side effects. In FP term it is called Referential transparency.

**Object-Oriented Programming vs Functional Programming**

1. OOP uses the imperative programming model, meaning functions are invariably coded in every step needed to solve a problem. You code each operation with the code itself specifying how to solve the problem. This model requires the programmer to know which functions are necessary to solve a problem instead of relying on models that can solve the problems.
2. FP uses the declarative programming model, meaning it relies on the underlying concepts of a programming language to execute the necessary steps to reach the predetermined outcome.
3. Imperative programs focus on the step-by-step process of solving a problem, whereas declarative programs focus on the result of solving a problem.
4. Another critical difference is mutability: OOP uses mutable data while FP uses immutable data. You can alter (or mutate) mutable objects after creation, whereas you can’t for immutable objects. In FP, you’ll need to make a copy of the object and use that copy to write the rest of your code.
5. Overall, immutable code is easier to update, more efficient to manage, and easier to test and debug. And because variables are constant, the resulting code is easier to understand and reason about. Many programmers and software developers prefer to work with FP models.
6. Ultimately, the right programming paradigm for you will depend on your intended application. OOP works best for standardized and straightforward projects, whereas FP works best for projects that require scalability and flexibility.

# Currying

Currying is the transformation of a function with multiple arguments into a sequence of single-argument functions. That means converting a function like this f(a, b, c, ...) into a function like this f(a)(b)(c)... . Currying is helpful when you have to frequently call a function with a fixed argument.

Normal Function :

function isGreaterThan(a, b) {

return b > a;

}

isGreaterThan(2, 5)

Currying function :

function isGreaterThan(a) {

return function(b) {

return b > a;

}

}

isGreaterThan(2)(5);

const isGreaterThan = a => b => b > a;

# object Update

1. In JavaScript we can copy object and update the object without mutable the value of principal object with object.assign() method.

Syntax: Object.assign(target, sources);

Code:

const obj = { a: 1 };

const copy = Object.assign({}, obj);

console.log(copy);

1. but now we use spread operator for update the object.

# redux with JavaScript

1. **component:** First we create a html screen where work with state.

Code:

<html lang="en">

<head>

</head>

<body>

    <div class="container">

        <p>Home</p>

        <h1 id="counter"></h1>

            <button id="inc"> increment </button>

            <button id="dec"> decrement </button>

</div>

    <script src="react.js"></script>

</body>

</html>

1. **initial the value:** For redux use we first create initial state. This state we pass to redux for use state change and UI interaction.

const initialState = {

    value: 0

}

1. **create action/reducer function:**

function counterReducer(state = initialState, action) {

  if (action.type == "increment") {

    return {

      ...state,

      value: state.value + 1,

    };

  } else if (action.type == "decrement") {

    return {

      ...state,

      value: state.value - 1,

    };

  } else {

    return state;

  }

}

1. **store create:** we use createStore() method for create store. Here we save state and action.

const store = Redux.createStore(counterReducer);

1. **dispatch method:**

inc.addEventListener("click", () => {

  store.dispatch({

    type: "increment",

  });

});

dec.addEventListener("click", () => {

store.dispatch({

    type: "decrement",

});

});

1. **Subscribe:**

store.subscribe(rendar);

1. **update UI :**

const rendar = () => {

  const state = store.getState();

  console.log(state);

  counter.innerText = state.value.toString();

};

# React-Redux

**Normal React-Redux:**

1. **install:** we should install “redux” then “react- redux”

npm i redux

npm I react-redux

1. **value initiate:**

const initialState = {

    value: 0

}

1. **action create:**

function counterReducer (state = initialState, action) {

switch(action.type){

case increment:

    return {

      ...state,

      value: state.value + 1,

    };

case decrement:

    return {

      ...state,

      value: state.value - 1,

    };

  }

default:

    return state;

}

}

1. **store create:** in react-redux we import createStore from redux. Then pass reducer functions.

Import {createStore} from “redux”

const store = createStore(counterReducer);

1. **provider:** now we provide store in component by wrapping the whole app. Then pass the store.

Import {Provider} from “redux”;

Import store from “. /app”;

<Provider store={store}>

<Counter />

</Provider>

**react-Redux hook:**

useSelector():

# redux toolkit

1. **install redux toolkit:** we install redux toolkit by command in terminal.

Npm install @reduxjs/toolkit

1. **createSliec:** A function that accepts an initial state, an object of reducer functions, and a "slice name", and automatically generates action creators and action types that correspond to the reducers and state.

**Syntax:**

function createSlice ({name: string, initialState: any, reducers: Object})

* 1. initialState: The initial state value for this slice of state.
  2. Name: A string name for this slice of state. Generated action type constants will use this as a prefix.
  3. Reducers: An object containing Redux "case reducer" functions.

Code:

import { createSlice } from '@reduxjs/toolkit'  
const counterSlice = createSlice({  
 name: 'counter',  
 initialState: 0,  
 reducers: {  
 increment: (state) => state + 1,  
 },  
})

* 1. **Return Value**[**​**](https://redux-toolkit.js.org/api/createslice#return-value)**:** createSlice will return an object that looks like:

{  
 name : string,  
 reducer : ReducerFunction,  
 actions : Record<string, ActionCreator>,  
 caseReducers: Record<string, CaseReducer>.  
 getInitialState: () => State  
}

1. Redux Store: