Functional Programming 101

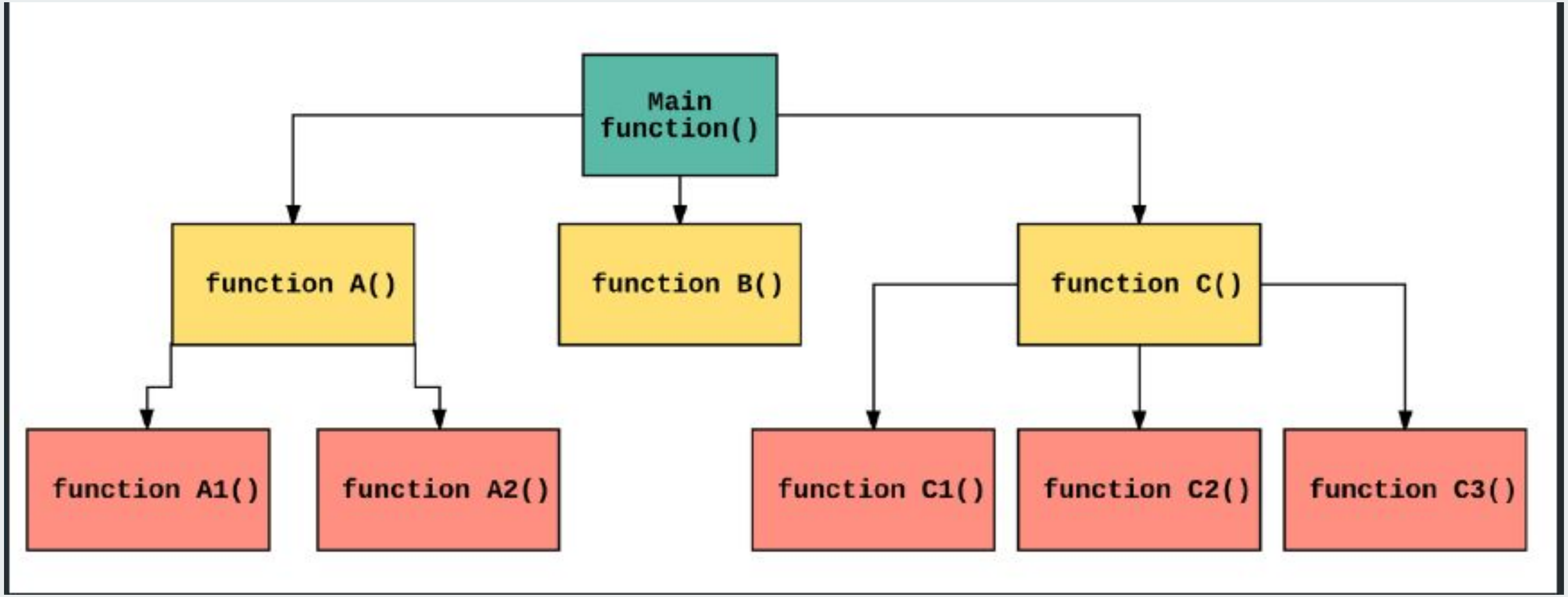
Agenda

What is FP
Basics of FP
Pure Functions
Immutability
Referential Transparency
Higher Order Functions
Partial Application and Currying

What is FP ?

In functional programming, we place a major emphasis on writing code using functions as building blocks. Your program is defined in terms of one main function. This main function is defined in terms of other functions, which are in turn defined in terms of still more functions, until at the bottom level the functions are just language primitives like number or string.

What is FP ?


Main function()
function A()
function B()
function C()
function A1()
function A2()
function C1()
function C2()
function C3()

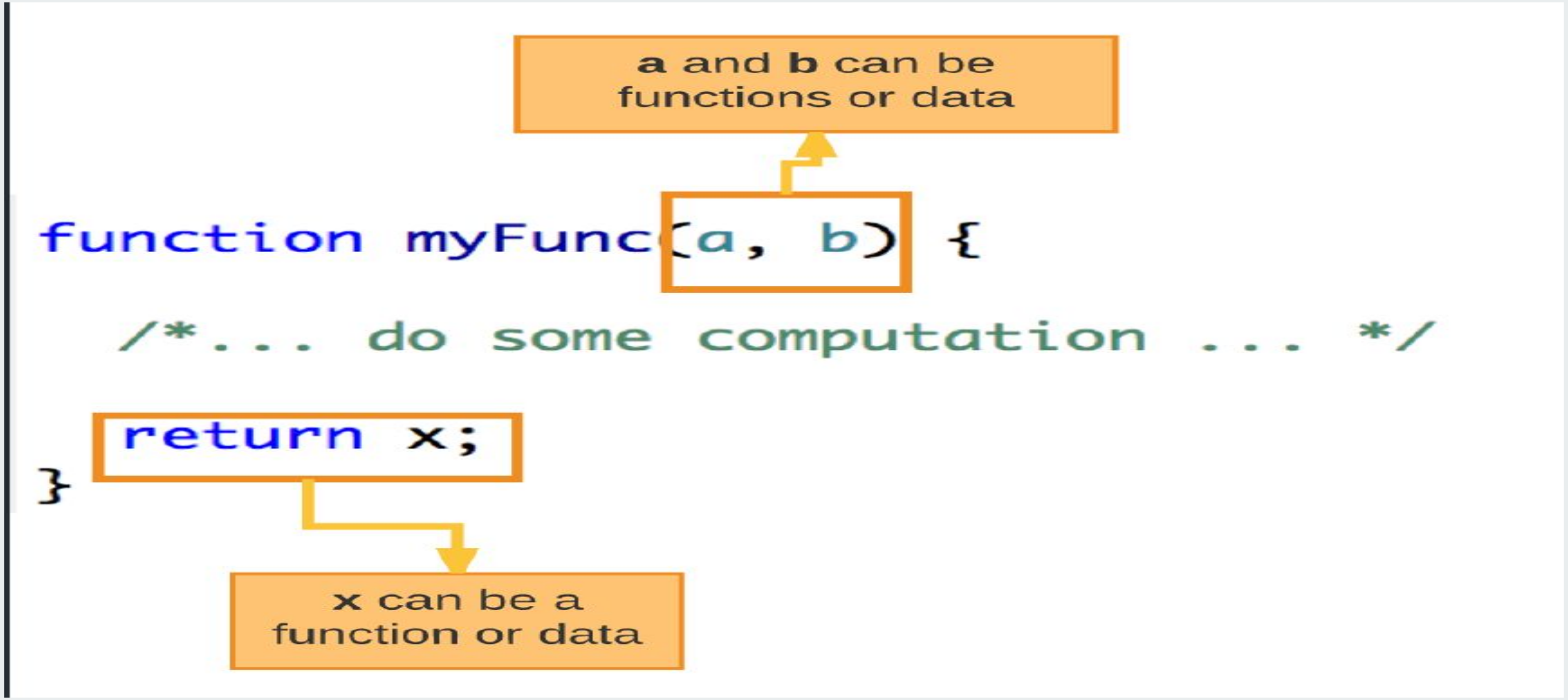
Basics

Every Software program has two things

- Behavior(What the program does)
- Data

In FP , behavior is handled only using functions and data is immutable. Rather than changing data they take in, functions in functional programming take in data as input and produce new values as output(always). Also functions in FP can be passed around just like data.

Basics



```
function myFunc(a, b) {
  /*... do some computation ... */
  return x;
}
```

a and **b** can be functions or data
x can be a function or data

Functions

- Functions are pure
- Functions use immutable data
- Functions guarantee referential transparency
- Functions are first-class entities

Pure Functions

Qualities of Pure Functions

- The function depends only on the input provided to it to produce a result (and not on any external state).
- The function does not cause any observable side effects, such as modifying a global object or modifying a parameter passed by reference

Pure Functions

```
var PI = 3.14;
const calculateArea  = (radius) => radius * radius * PI;

let count = 1;
const  increment = (val) => count = count + val;

const reverseArray = (array) => array.reverse();
```

Pure Functions

```
async function amountExceedsCreditLimit(amount) {
  const limit = await getCreditLimit() // assume api call
  if (amount > limit) {
        return true;
  }
  return false;
}

console.log(“Hello World!”);
```

Pure Functions benefits

- Pure functions are predictable
- Pure functions are easy to test
- Easier to refactor

Immutability

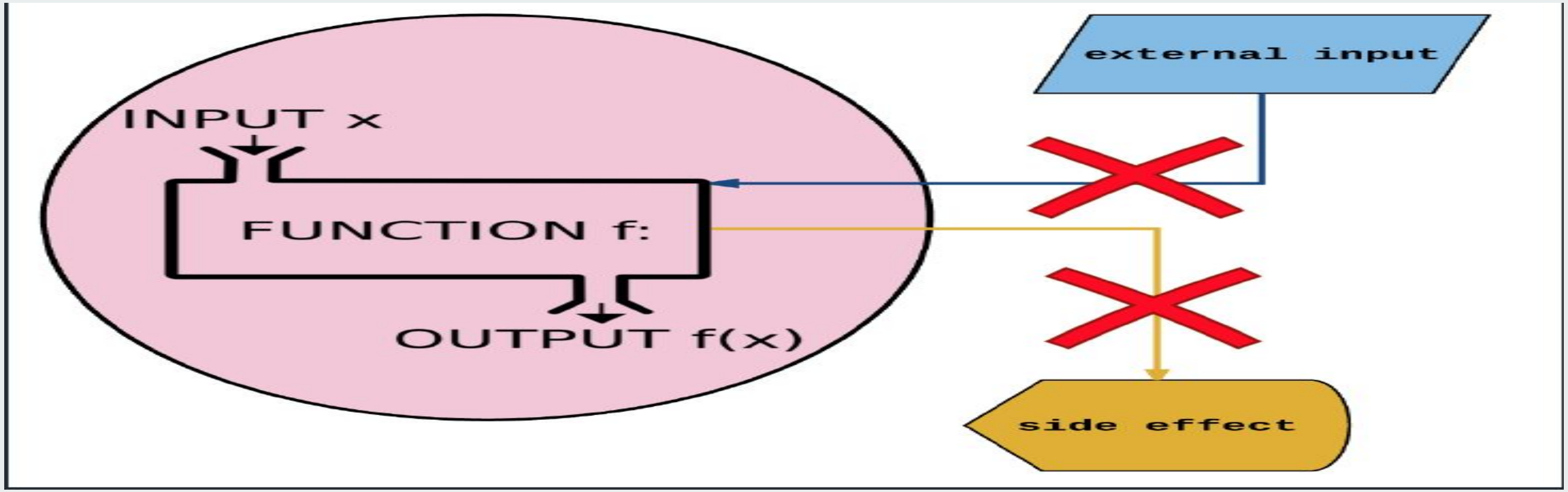
```
var firstName = “Maneesh”
firstName[3] = “i”;
console.log(firstName[3]);
console.log(firstName);

firstName = firstName.slice(0,3);
console.log(firstName);
```

Immutability

Mutation modifies the underlying object without affecting the link between the variable and the object, whereas assignment changes the link to a different object, but doesn’t modify any objects.

Functions -Referential Transparency



Functions -Referential Transparency

If a function consistently yields the same result for the same input , it is referentially transparent. Referential transparency gives the ability to freely replace an expression with its value and not change the behavior of the program.

Functions -Referential Transparency

Pure functions + immutable data = referential transparency i.e a function that doesn't rely on external states or mutable data will always return the same output for a given input.

Functions are first class Citizens

- Refer to it from constants and variables
- Pass it as a parameter to other functions
- Return it as result from other functions

Higher Order Functions

A higher order function is a function that meets at least one of the following requirements: Note most functions which take a callback are higher order functions.

- It accepts a function as an argument
- It returns a new function

Higher Order Functions

Benefits of higher order functions

- Hide Implementation Details

```
numbers.stream().map(s -> Integer.valueOf(s))
.filter(number -> number % 2 == 0)
.collect(Collectors.toList());
```

Higher Order Functions

Benefits of higher order functions

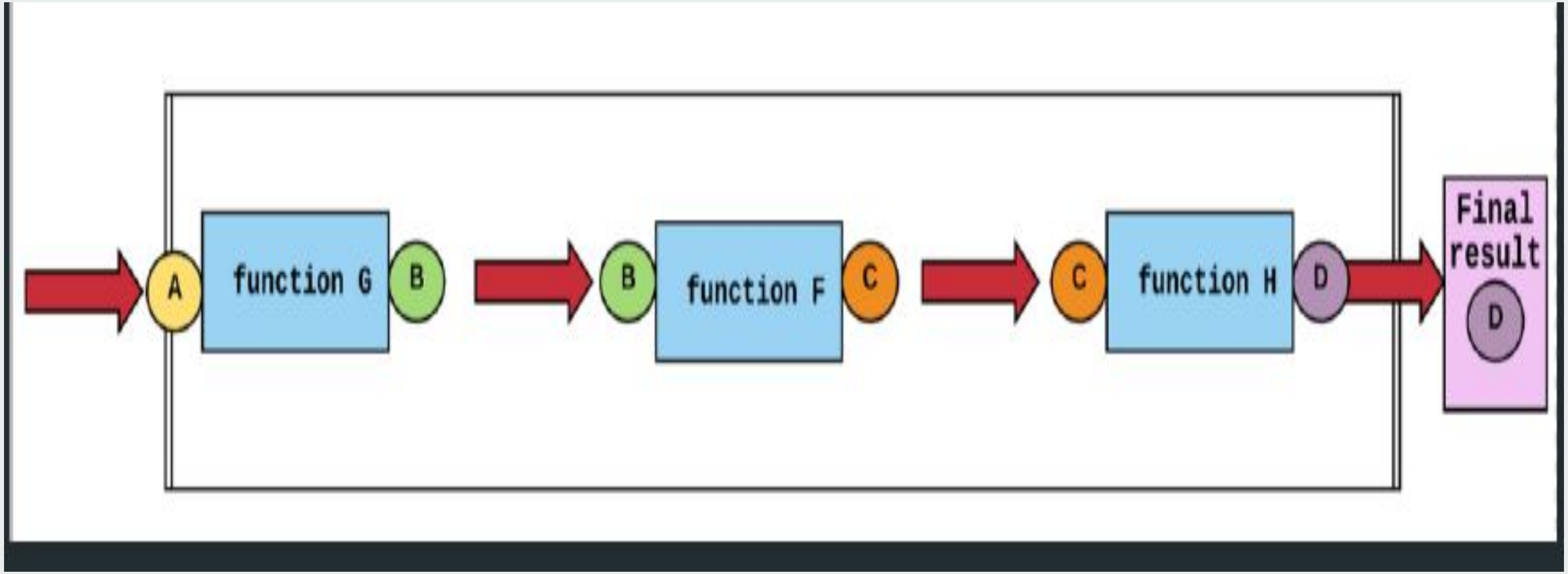
- Adding Functionality to an Existing Function

```
function logResult(fn) {
return (...args) => { const result = fn(...args)
console.log(result)
return result
}}
```

Higher Order Functions

```
const findAndMultiply = (index,multiplier) => {
    return (array) => {
        if (array && typeof(array[index] == ‘number’) {
            return array[index] * multiplier ;
        else{
            Return “Bhaiya, ye na ho pai””
        }
    };
};
```

Function Composition



Function Composition

Create small reusable functions that you can combine to compose new functions.

```
BiFunction<String, List<Article>, List<Article>> byAuthor =
  (name, articles) -> articles.stream()
    .filter(a -> a.getAuthor().equals(name))
    .collect(Collectors.toList());
```

Function Composition

```
BiFunction<String, List<Article>, List<Article>> byTag =
  (tag, articles) -> articles.stream()
    .filter(a -> a.getTags().contains(tag))
    .collect(Collectors.toList());


Function<List<Article>, List<Article>> sortByDate =
  articles -> articles.stream()
    .sorted((x, y) -> y.published().compareTo(x.published()))
    .collect(Collectors.toList());
```

Function Composition

```
Function<List<Article>, Optional<Article>> first =
   a -> a.stream().findFirst();

Function<List<Article>, Optional<Article>> newest =
   first.compose(sortByDate);

BiFunction<String, List<Article>, List<Article>>
byAuthorSorted =  byAuthor.andThen(sortByDate);
```

Function Composition

The *compose* method uses a parameter named, *before*, and the *andThen* method uses a parameter named, *after*. These names indicate the order that the functions will be evaluated on.

$f(x) = (2+x)*3$
$g(x) = 2 +(x*3)$

Use *compose* and *andThen* to form f(x) and g(x).

Function Composition

```
Function<Integer, Integer> baseFunction = t -> t + 2;

Function<Integer, Integer> afterFunction =
baseFunction.andThen(t -> t * 3);
System.out.println(afterFunction.apply(5));
Function<Integer, Integer> beforeFunction =
        baseFunction.compose(t -> t * 3);
System.out.println(beforeFunction.apply(5));
```

when the apply method is executed, the corresponding lambda expression is executed

Fluent Interfaces

```
int hoursWorked[] = {8, 12, 8, 6, 6, 5, 6, 0};
   int totalHoursWorked = Arrays.stream(hoursWorked)
         .filter(n -> n > 6)
         .sum();

LocalDateTime timeInstance = LocalDateTime.now()
         .plusDays(3)
         .minusHours(4)
         .plusWeeks(1)
         .plusYears(2);
```

Partial Application

Partial application is a way to turn a function that expects multiple parameters into one that will keep returning a new function until it receives all its arguments

```
const f = (a,b,c) => a * b *c
const g = partial(f,[2,3])
const h = g(4)
```

Partial Application

```
const applyTax = (amount,tax) => amount + (amount *  tax)

applyTax(0.08) =>

applyTaxOfEightPercent = partial(applyTax,[0.08]);
applyTaxOfFivePercent = partial(applyTax,[0.05]);
applyTaxofEightPercent(100);
applyTaxofFivePercent(100);
```

Exercise - Template design pattern. Try implementing it using partial application.

Currying

Currying is the process of taking a function that accepts N arguments and turning it into a chained series of N functions that each take one argument. The difference between partial application and currying is that with currying, you can never provide more than one argument — it has to be either one or zero. If a function takes 5 arguments, we can partially apply 3 of them. But with currying, we only pass one argument at a time. So if a function takes 5 arguments, we have to curry 5 times before we get the resulting value.
Future and Promise are an application of currying.