# *Building Large Distributed Systems Using Functional Programming*

*Jaideep Ganguly, Director of Software Development*

# What is wrong with Object Oriented Programming (OOP)?

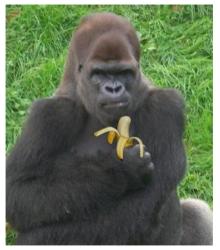
1. The vision of code reuse, through the "right" design as offered by object-oriented (OO) languages such as Java, is a myth. In practice OO languages tightly couple data with behavior, it is hard to extract the behavior into reusable code.

2. Design is a much misused word. Design is not about drawing boxes and arrows between classes and interfaces.

3. Design patterns offer suggestions for modeling code that solves common problems in a manner that is resilient to change. Correctly applied, these patterns are supposed to reduce the time to market in an environment of changing requirements.

4. But, design patterns are opinions based on experience. As programming languages and technical challenges evolve, there is no way of knowing how well they will stand the test of time. Example, the Singleton design pattern.

# What is wrong with Object Oriented Programming (OOP)?



The following quote says it all. "The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle". – Joe Armstrong, the creator of Erlang.

1. Lots of dependencies is a *code smell*. Lots of dependencies require lots of testing and testing in OOP is not easy.

2. Most software engineers look at unit and integration testing as a chore and do not quite look forward to it. Tests are often perfunctory resulting in illusory productivity with a mess of alpha/beta quality software that require a magnitude more of effort towards fixes.

3. *Mocking* is fairly ineffective with OOP because of multiple reasons. Firstly, test setup is slow because of the need to mock or stub the inputs along with any dependencies necessary for the code to execute a scenario.

4. Secondly, multiple happy path scenarios and error scenarios need to be considered. And finally, test suites need to be written efficiently so that it does not take hours to run in *continuous integration*.

# What is Functional Programming (FP)?

1. There is a need to move from object dependencies to functional dependencies and evolve APIs towards behavior instead of objects.

2. FP models behavior through a declarative paradigm, i.e., through expressions or declarations instead of statements. Functional programming brings in simplicity, are easier to test and eliminates the need for mocking.

3. But learning FP can be a daunting process as it brings with it unfamiliar terminology that may appear to be difficult. But the mathematically sound FP with concepts such as **immutability**, **referential transparency**, **functional composition** and **monads**, is the right choice.

4. In FP, behavior is a first-class citizen. Functions deterministically transform single inputs into single outputs. You will compose functions into data flows where providing data to an initial function, the output of that function becomes the input to the next function and so on.

# What is Functional Programming (FP)?

1. FP avoids changing state and mutable data. Programming is done with expressions or declarations instead of statements to ensure **referential transparency**.

2. FP uses recursion to do looping. **Immutability** creates simpler and safer code. In FP, the output of a function depends only on the arguments of the function and eliminates **side effects**.

3. A **function is a first-class citizen**, i.e., it is just another value. **Higher-order Functions** either take functions as parameters, return functions or both. Also, lambda expression or anonymous function can access its **closure**, i.e. the variables declared in the outer scope.

4. **Data flow** can be modeled using **function compositions**, a chaining process, is the pointwise application of one function to the result of another to produce a third function. Mathematically, an example of this concept is:

   *output = h ( g ( f ( input ) ) )*

5. The essence of all software development is the process of breaking a large problem down into smaller, independent pieces (decomposition) and composing the solutions together to form an application that solves the large problem (composition). But what if any of the functions fail to return a valid response? That leads us to the topic of Monads.

# Tail Recursion

1. Will excessive reliance on recursion cause stack overflows? That brings us to to the topic of **Tail Recursion**.


2. A recursive function is eligible for tail recursion if the function call to itself is the **last operation** it performs. Since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use.

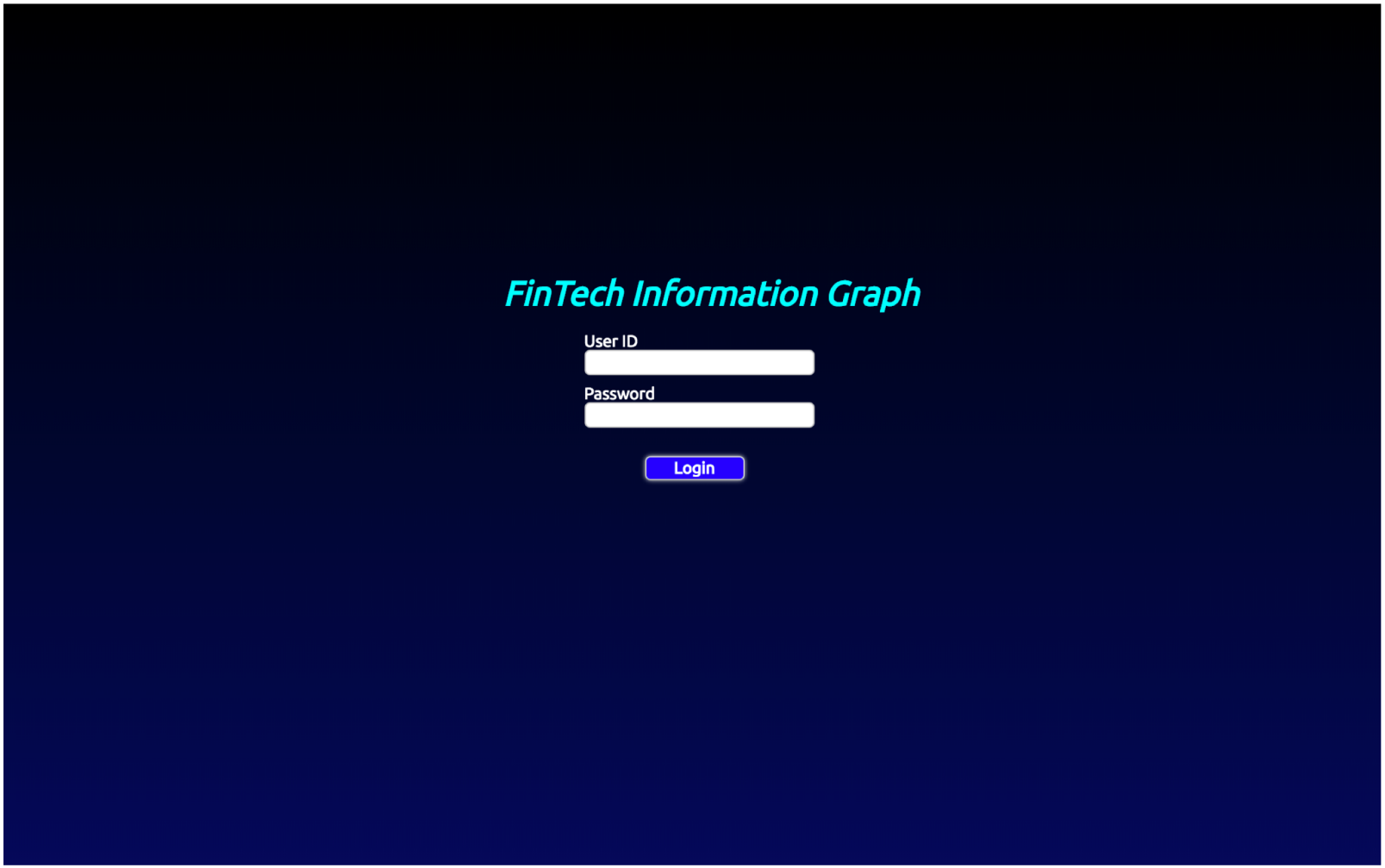
3. The Kotlin (as well as many others, **but not Java**) compiler is smart enough to recognize this and effectively converts into a while loop and jumps to the called function. The compiler optimization for **tail recursive** functions removes the possibility of stack overflow. You have to mark the function with tailrec for the Kotlin compiler to perform tail recursion optimization.

```
tailrec fun fibonacci(n: Int, a: Long, b: Long): Long {
    return if (n == 0) b else fibonacci(n-1, a+b, a)
}
```

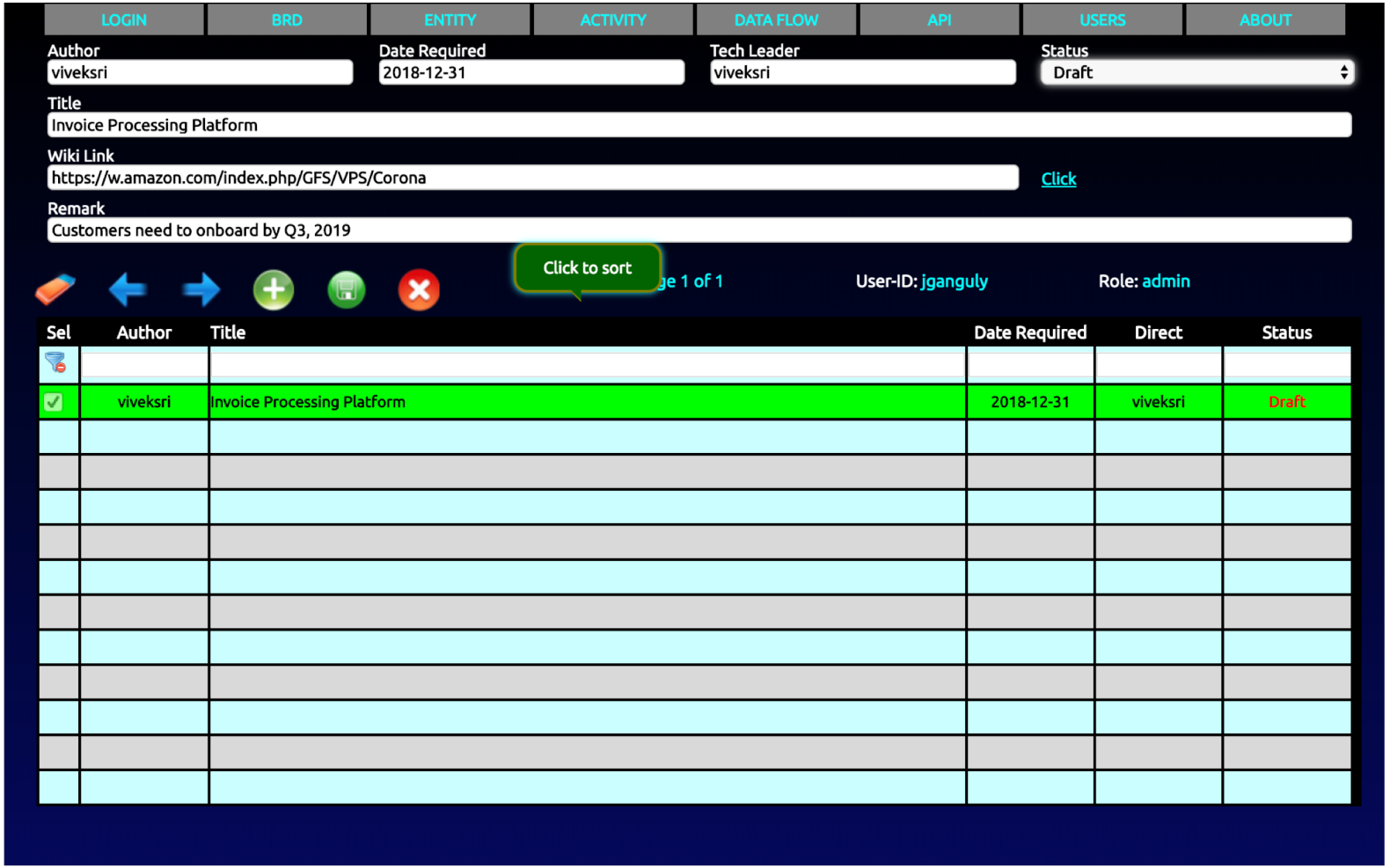
# Benefits of Functional Programming

| | | |
|---|---|---|
| No states<br>No side effects | Mathematically correct code via Composition | Easier Testing<br>No Mocks |
| Better parallelized | Better performance | Higher Productivity |

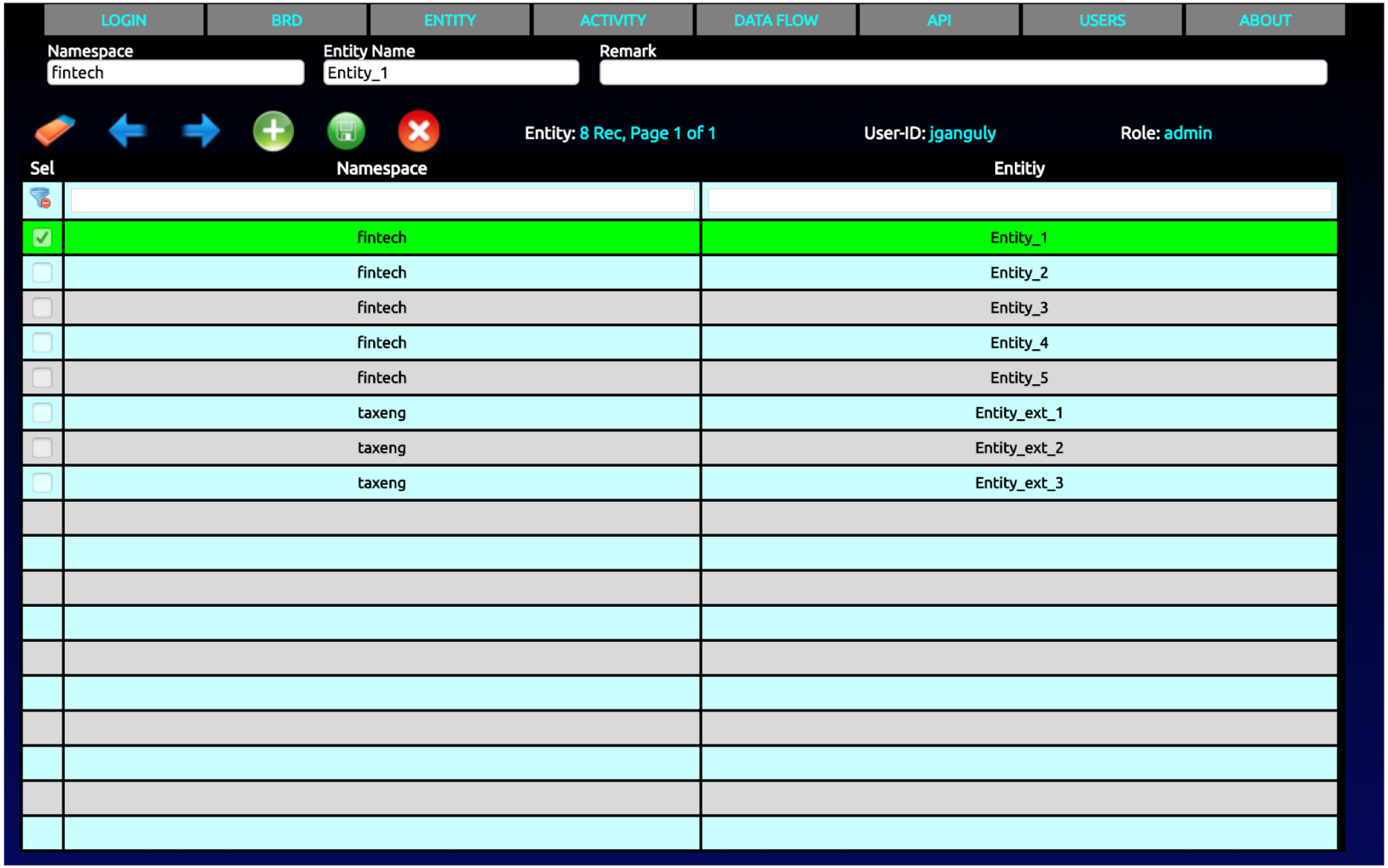
# FinTech Information Graph (FIG)
# https://fct.corp.amazon.com:8443/fig.php



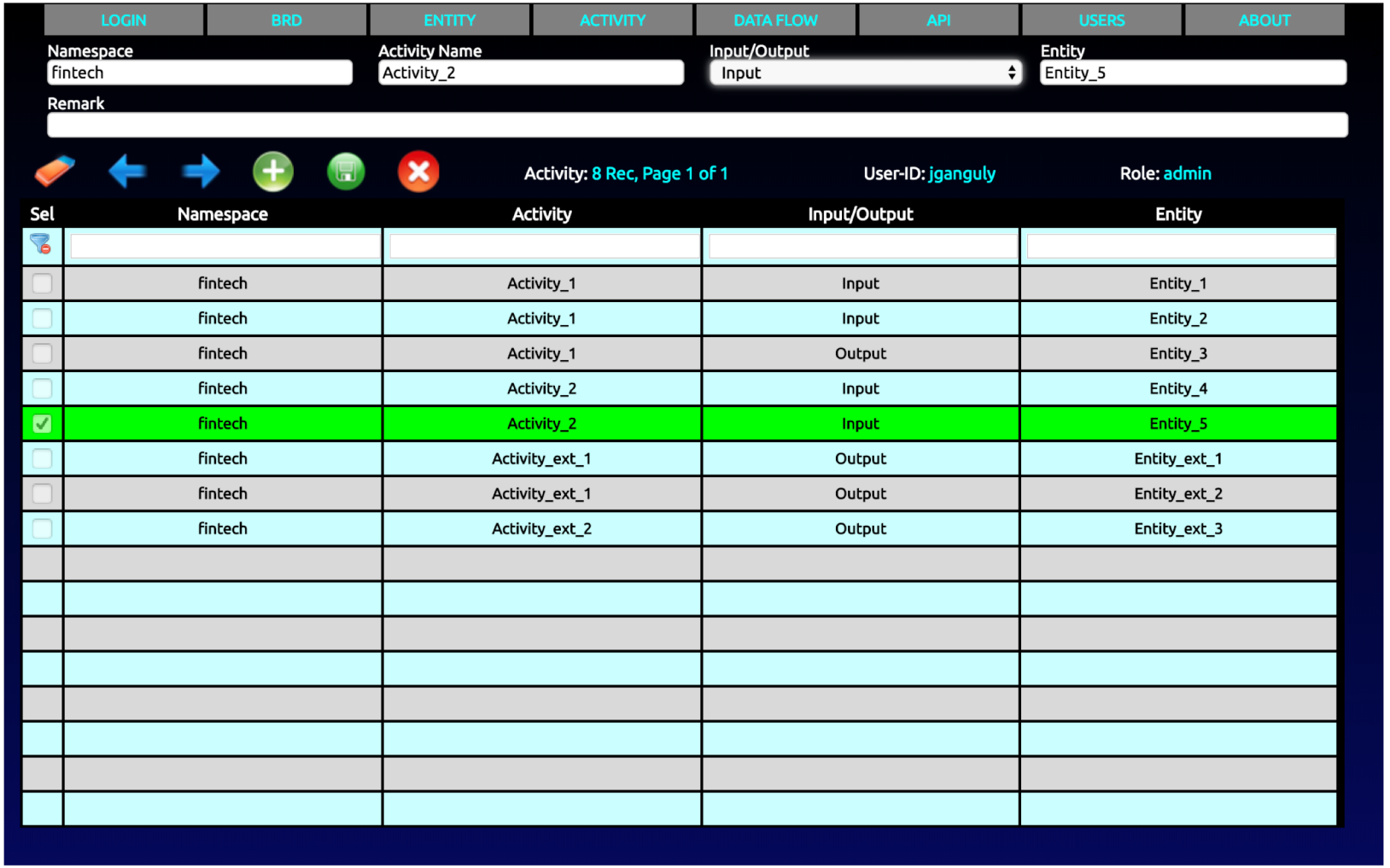
# FIG – BRDs


LOGIN
BRD
ENTITY
ACTIVITY
DATA FLOW
API
USERS
ABOUT
Author
viveksri
Date Required
2018-12-31
Tech Leader
viveksri
Status
Draft
Title
Invoice Processing Platform
Wiki Link
https://w.amazon.com/index.php/GFS/VPS/Corona
Click
Remark
Customers need to onboard by Q3, 2019
Click to sort
ge 1 of 1
User-ID: jganguly
Role: admin
Sel
Author
Title
Date Required
Direct
Status
viveksri
Invoice Processing Platform
2018-12-31
viveksri
Draft

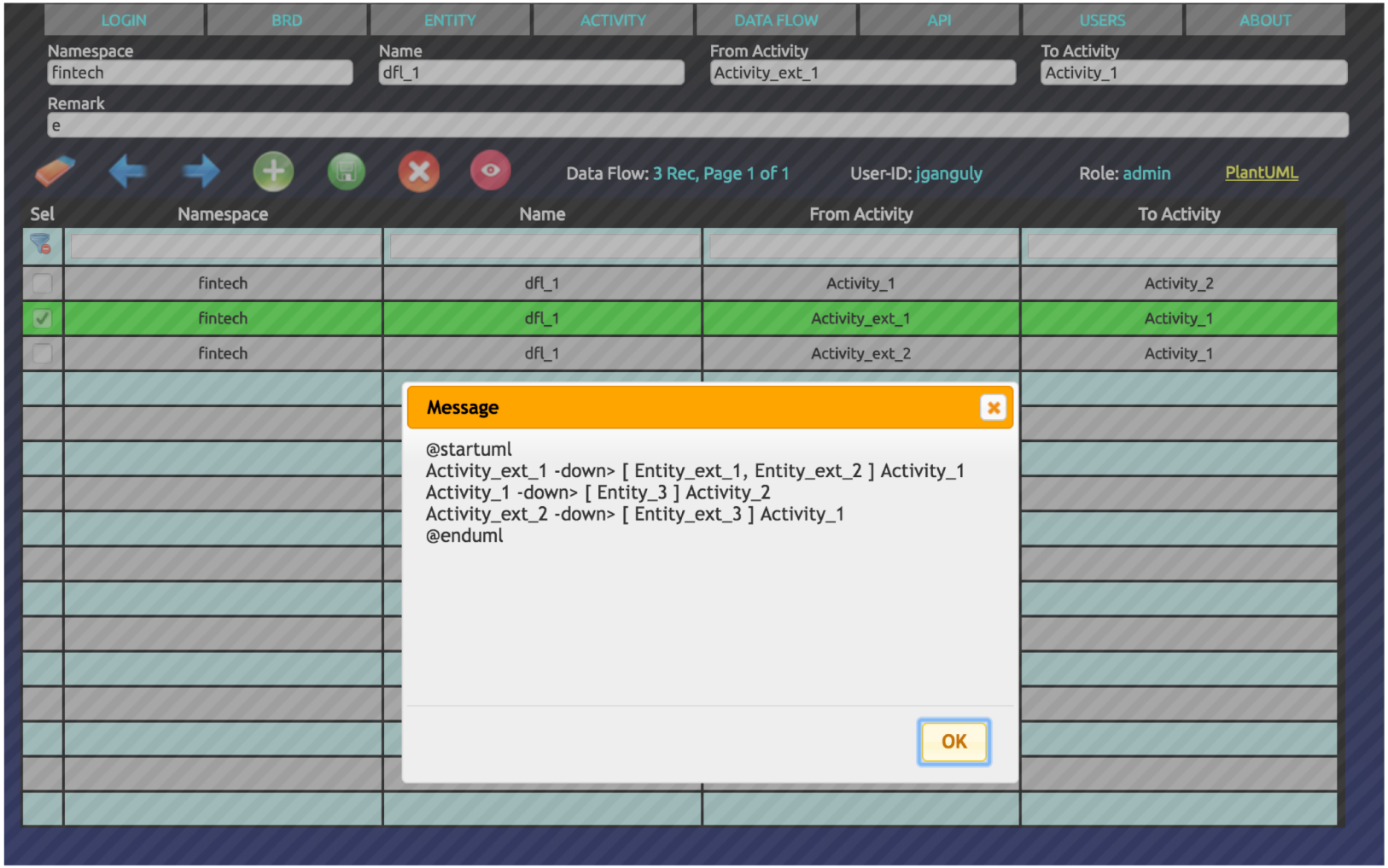
# FIG – Entities



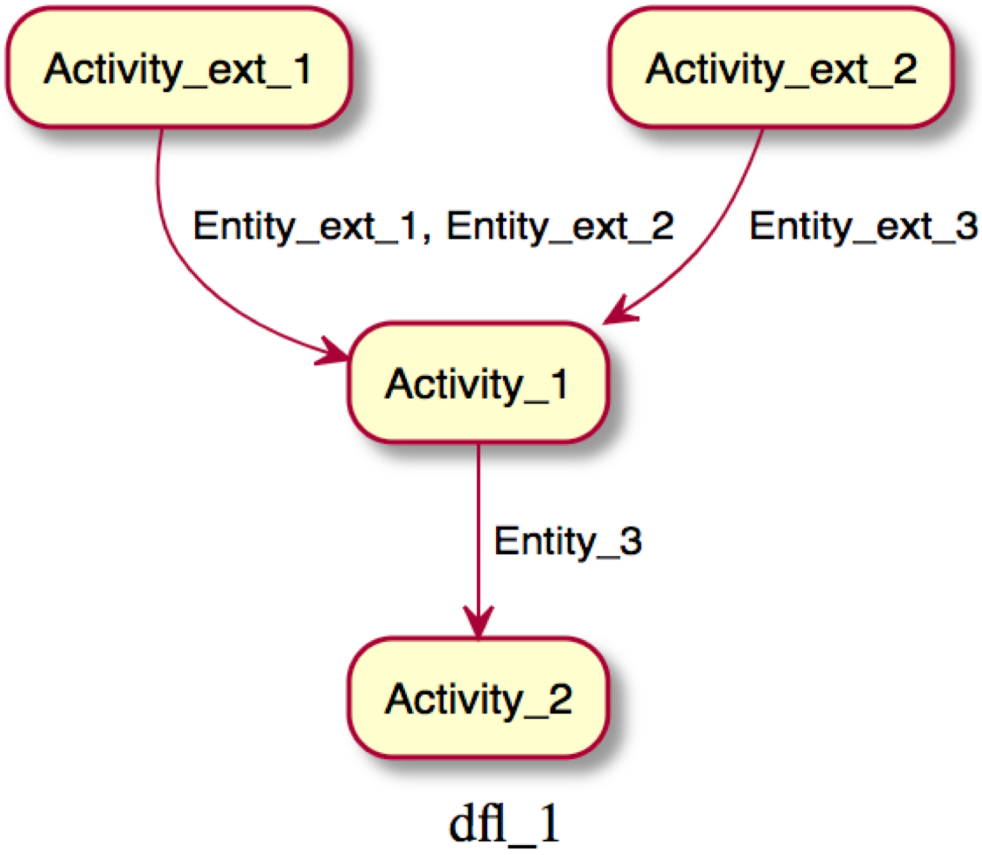
# FIG – Activities



# FIG – Data Flows

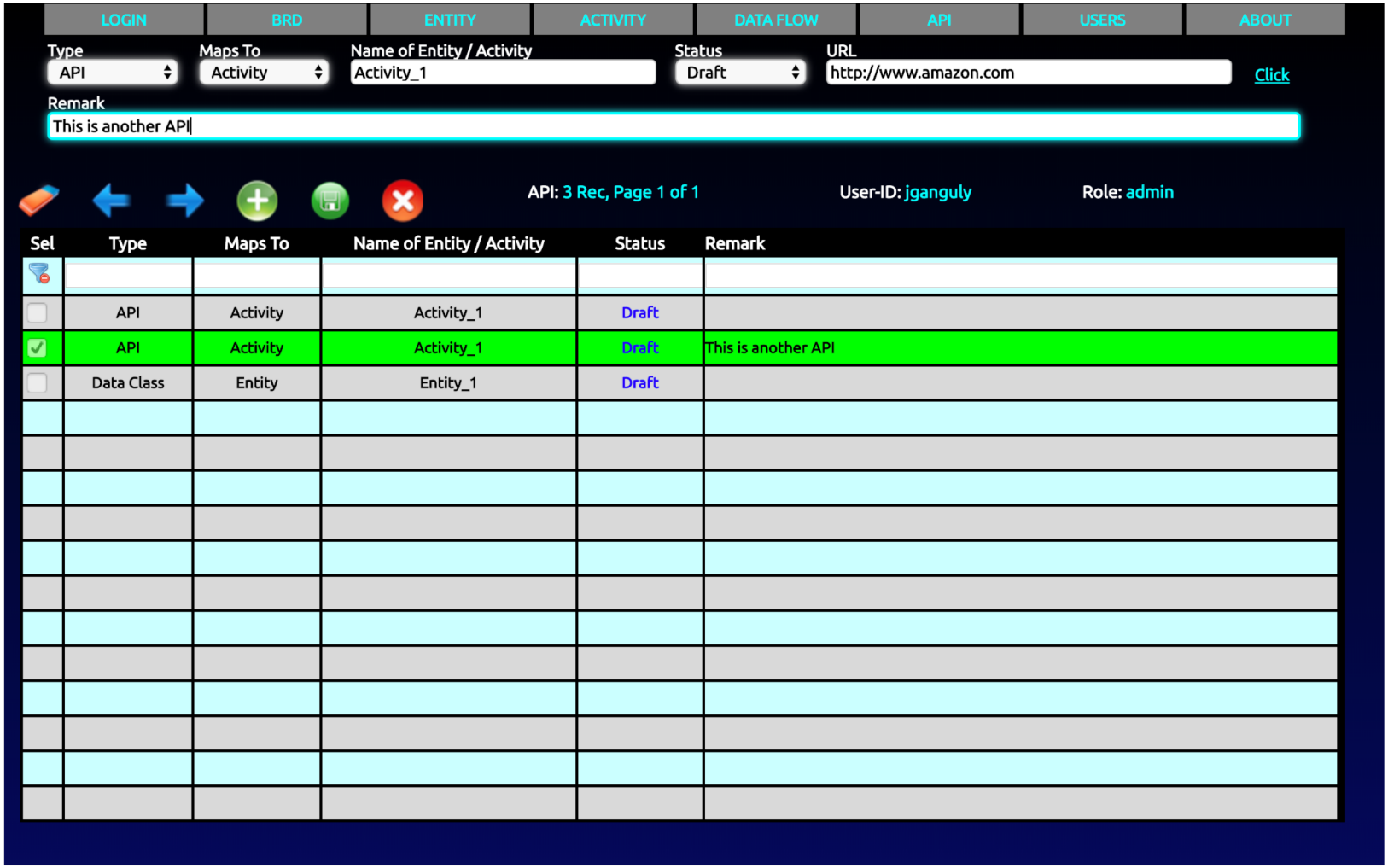


# FinTech Information Graph (FIG)




dfl_1

# FIG – Data Classes & APIs



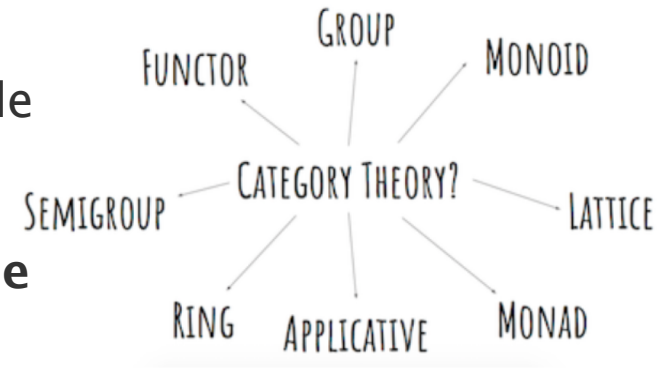
# Category Theory in Mathematics

The world of *Category Theory* in mathematics is full of obscure concepts as shown below. Luckily, we do not need to delve into all the gory details!

1. A **category** is an algebraic structure that comprises of **objects** that are linked by **arrows**. A simple example is the category of sets, whose objects are sets and whose arrows are functions. **In a statically typed language, we can translate the notion of *set* into the notion of *type*.**

2. A **functor** is simply a map between **categories**. An **endofunctor** is defined as a functor from one category back to the same category.

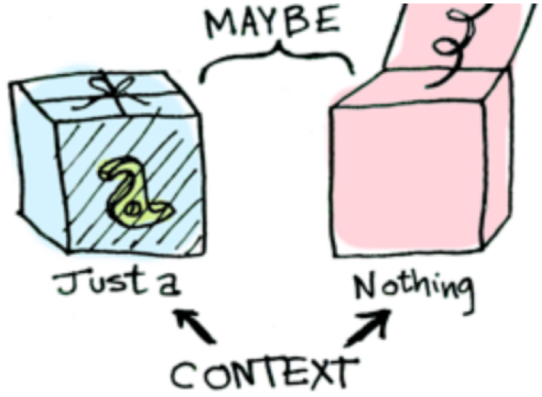
3. A **Monoid** is a set that obeys certain rules. A **functor** is said to be **Applicative** when it preserves the monoidal structure. And a Monad is just a monoid in the category of endofunctors.




It is both mathematically correct and totally useless to anybody learning functional programming.

# Functor, Applicative & Monad

When you apply a function to this value, you'll get different results **depending on the context.** This is the idea that Functors, Applicatives, Monads, Arrows, etc., are all based on.

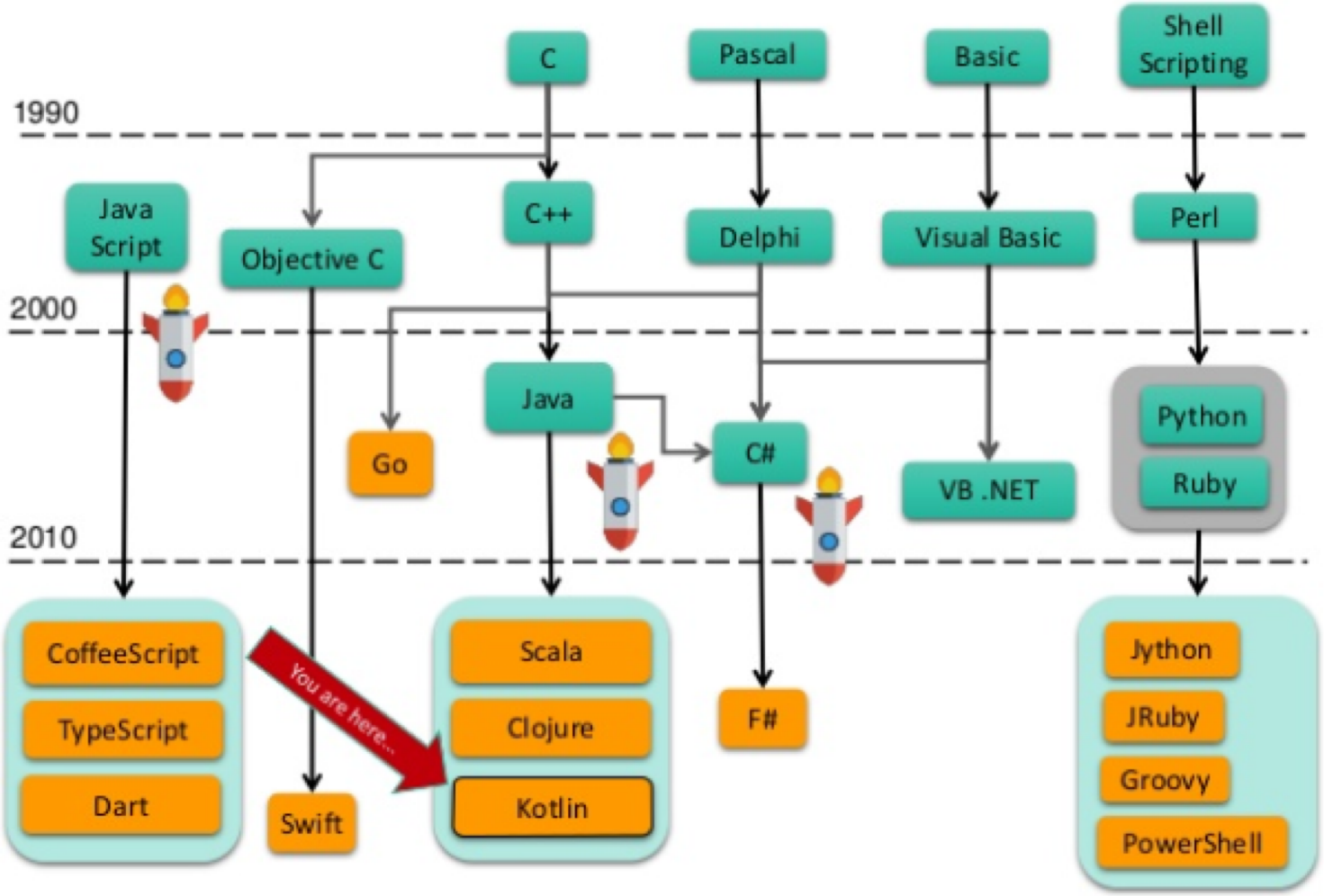



1. Functor is a typeclass. Typeclasses are interfaces that define a set of extension functions associated to one type. A Functor is any data type that defines how map (fmap in Haskell) applies to it.

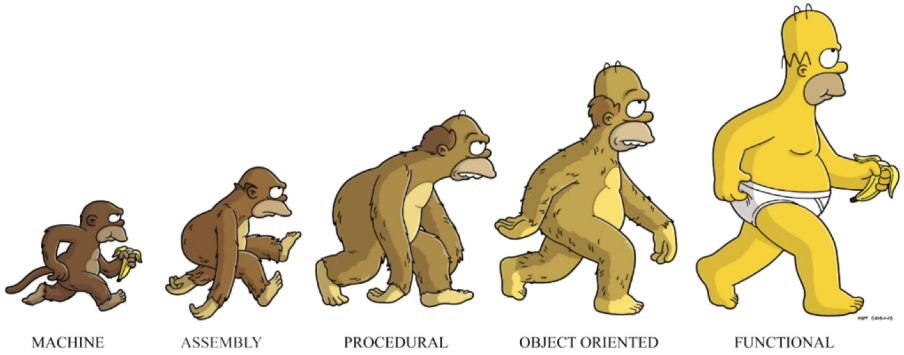
2. With an applicative, the input values are wrapped in a context.

3. Monads use a flatMap to apply a function that returns a wrapped value to an input wrapped value.

# Landscape of Common Languages



# Multi Paradigms Languages




" MOST PEOPLE TALK ABOUT JAVA THE LANGUAGE, AND THIS MAY SOUND ODD COMING FROM ME, BUT I COULD HARDLY CARE LESS. AT THE CORE OF THE JAVA ECOSYSTEM IS THE JVM. "

**James Gosling**

## Summary of Some Common Terms

1. Higher order function
2. Referential Transparency
3. Purity
4. Tail Recursion
5. Functional composition
6. Partial Application
7. Option
8. Functors
9. Applicative
10. Monoid
11. Monad
12. Closure
13. Foldable
14. Typeclass

# State of Pure functional languages in JVM

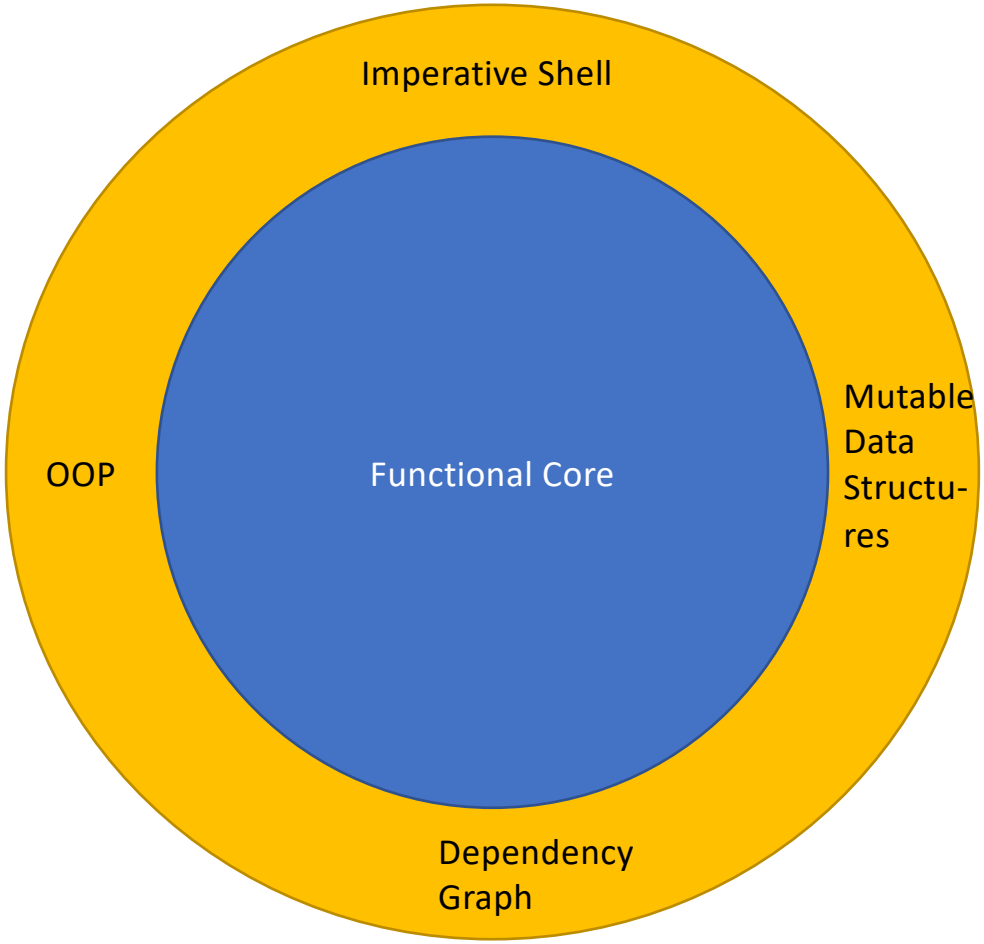
1. Dependencies are a part of life for a developer operating in a medium to large ecosystem.

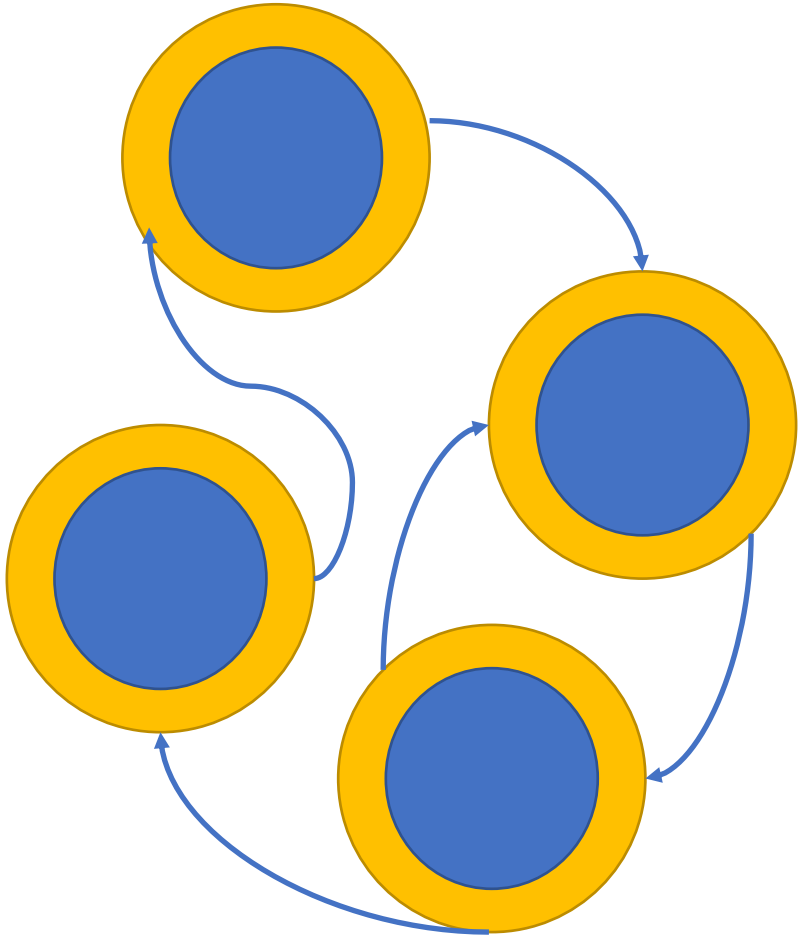
2. Mutability is necessary for optimization (else we can end up creating LOTs! of objects). Languages like Scala and Closure though provide persistent data structures, which is nothing but mutating an object behind the scene, but hidden from developers. Because they are language features, functional programming benefits are retained for developers. Kotlin does not have persistent data structures out of the box.

3. Side-effects like IO are unavoidable. Else, how will the program interact with the world?

4. The ecosystem is not pure. For example, dependencies are almost always impure, thus rendering the system a mix of impure and pure functions.

# JVM World


Imperative Shell
OOP
Functional Core
Mutable
Data
Structu-
res
Dependency
Graph



# A Sample Monad TypeClass

```
sealed class Monad<out A> {

    object None : Monad<Nothing>()
    data class Value<out A>(val value: A) : Monad<A>()

    //  Monad - Apply a function to a wrapped value and return a wrapped value using flatMap (liftM or >>= in Haskell)
    inline infix fun <B> flatMap(f: (A) -> Monad<B>) : Monad<B> = when (this) {
        is None  -> this
        is Value -> f(value)
    }
}
```

# Functions

```
data class DC (var name: String, var value: Double)

fun mysqrt(a: DC) = when {
    a.value >= 0 -> {
        var y = kotlin.math.sqrt(a.value)
        var ds = DC("JaideepG", y)
        Monad.Value(ds)
    }
    else   -> Monad.None
}

fun mylog(a: DC) = when {
    a.value > 0  -> {
        var y: Double = kotlin.math.ln(a.value)
        var ds = DC("JaideepG", y)
        Monad.Value(ds)
    }
    else   -> Monad.None
}
```

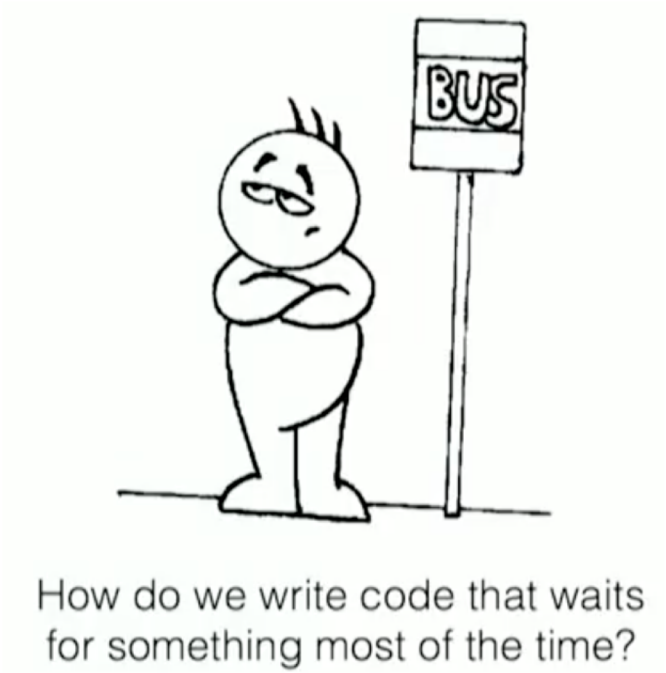
```
fun myinv(a: DC) = when {
    a.value >= 0 -> {
        var y: Double = 1/a.value
        var ds = DC("JaideepG", y)
        Monad.Value(ds)
    }
    else   -> Monad.None
}
```

# Functional Composition

```
fun testMonad() : Monad<out DC> {
    var inp: Monad<DC>

    var listOfFun: List<(DC) -> Monad<DC>> =
mutableListOf<(DC)->Monad<DC>>()
    listOfFun += ::mysqrt
    listOfFun += ::mylog
    listOfFun += ::myinv

    inp = Monad.Value(DC("Jaideep",100.0))
    var result = execute(inp, listOfFun)
    println("Folded = $result")
    println()
}

fun <T> execute(input: Monad<T>, fns: List<(T) ->
Monad<T>>): Monad<T> =
        fns.fold(input) { inp, fn -> inp.flatMap(fn) }

fun main(args: Array<String>) {
    testMonad()
}
```

```
// Composition with iteration and
has mutation
var iter = listOfFun.iterator()
while (iter.hasNext()) {
    inp = inp.flatMap(iter.next())
// Mutating
    println(inp)
}
```

# Concurrency




- Threads are blocking.
- Threads are expensive.
- How many threads can you have ?
- 100, 1,000, 10,000, 100,000?

# The Perils of Concurrency!

Managing concurrency is wickedly tricky for large programs. At each program point, you must reason about which locks are currently held. At each method call, you must reason about which locks it will try to hold, and convince yourself that it does not overlap the set of locks already held.

**A Diverging Problem** – This problem is magnified because the locks you reason about are not simply a finite list in the program since the program is free to create new locks at run time as it executes. Testing is not reliable with locks. Threads are non- deterministic, you can very well successfully test a program a thousand times and yet the program could go wrong the first time it runs on deployment!

**Will over Engineering work?** – Over engineering will not solve the problem. Just as you avoid locks, you cannot also put a lock around every operation. The problem is that while new lock operations remove possibilities for race conditions, it simultaneously introduces possibilities for deadlocks.

**Actors and Threads** – That is why Scala provides an alternative concurrency approach, one based on Erlang's message-passing actors. An actor is a kind of thread that has a mailbox for receiving messages. Actors are implemented on top of normal Java threads. But Java threads are not cheap. Java virtual machines can have millions of objects but only a few thousand threads. Switching threads can often take thousands of processor cycles. For a program to be efficient, it is important to be sparing with thread creation and thread switching.

# Async Computation

1. With Kotlin 1.3 we finally have a stable library for coroutines – a major release just around the corner. This means that there won't be any breaking changes to the API. Now is an especially good time to learn how to use them.

2. Coroutines are basically light-weight, much more efficient threads. As a bonus, they are extremely easy to work with once you know the basics.

3. Asynchronous coroutine code looks the same as a classic synchronous code. You don't need to learn any new programming paradigms.

```
runBlocking {
    var inp : Monad<DC>  = Monad.Value(DC("Jaideep", 100.0))
    val deferred1 = async { execute(inp, listOfFun) }
    val value = deferred1.await()
    println("${Thread.currentThread().name} : $value")
}
```

# Drawing the line

1. Functional programming (FP) has a higher learning curve for us folks coming from OOP background. You will feel frustrated at times. My suggestion is to give it some time. You will end up having learned something ☺

2. Functional programs can become cryptic. Focus on readability rather than using idioms of FP which your team does not understand.

3. Similarly, dive deep into the FP ecosystem before using them in production. Establish culture of learning, best practices, must-have-tools before going full prod-mode, the code of which only you or just a few understand.

# Questions?

Acknowledgment: With contributions from viveksri@amazon.com

Reference:
Functional Programming - Design through Functional Composition and Monads
in
https://jaideep.aka.corp.amazon.com/

jganguly@amazon.com