
AVISI

FP and OOP: Best Friends Forever?

Combining the best of two worlds

Agenda.

- What?
- So what?
- Now what?

What?

Setting the stage

Paradigms.

- Procedural (PP)
- Object-Oriented (OOP)
- Functional (FP)

Procedural Programming.

	PP
Imperative	✅
Mutable state	✅
Logic and data separated	✅

Object-Oriented Programming.

	PP	OOP
Imperative	✅	✅
Mutable state	✅	✅
Logic and data separated	✅	
Encapsulation / data hiding		✅
Dynamic polymorphism		✅

Functional Programming.

	PP	OOP	FP
Imperative	✅	✅	
Declarative			✅
Mutable state	✅	✅	
Referential transparency			✅
Logic and data separated	✅		✅
Encapsulation / data hiding		✅	
Dynamic polymorphism		✅	
Functions are first-class citizens			✅

Referential Transparency?

The result of a function depends **only** on its input parameters.

```
fun transparent(a: Int, b: Int) =
    a + b


fun opaque(a: Int, b: Int) =
    a + b + Random.nextInt()
```

First-class functions?

Functions are **values**, just like integers, strings, etc.

```
val list = listOf("hello", "world")

val toUppercase: (String) -> String =
   { s: String -> s.uppercase() }

println(
    list.map(toUppercase) // declarative!
)

// Output: [HELLO, WORLD]
```

So what?

We can use FP in OOP languages!

Many modern **OOP** languages have adopted **FP** concepts

Functional-Object Oriented Programming.

	PP	OOP	FP	F-OOP
Imperative	✅	✅		
Declarative			✅	🎉
Mutable state	✅	✅		🃏
Referential transparency			✅	🎉
Logic and data separated	✅		✅	
Encapsulation / data hiding		✅		🎉
Dynamic polymorphism		✅		🎉
Functions as first class citizens			✅	🎉

Demo!

Now what?

OOP and FP: BFFs?

Yes!

But a good relationship is about giving and receiving

FP

- → - Allow some impure code
- → + Encapsulation

OOP

- → - Constructors
- → + Referential transparency
- → + More patterns
- → + First-class functions

Encapsulation?

Not as easy as you think 😢

```
class TodoList {
   private final List<TodoItem> items =
           new ArrayList<>();

   public List<TodoItem> getItems() {
       return items;
   }
}

var list = new TodoList();
var items = list.getItems();
items.add(new TodoItem());
```

Encapsulation?

Just return an iterator 💡

```
class TodoList {
   private final List<TodoItem> items =
           new ArrayList<>();

   public Iterator<TodoItem> getItems() {
       return items.iterator();
   }
}
```

Encapsulation?

Hmm... 🤔

```
class TodoList {
   private final List<TodoItem> items =
           new ArrayList<>();

   public Iterator<TodoItem> getItems() {
       return items.iterator();
   }
}

var list = new TodoList();
list.getItems().remove();
```

Encapsulation?

OOP (Java) solution:
Return an **unmodifiable list**

```
class TodoList {
   private final List<TodoItem> items =
           new ArrayList<>();

   public List<TodoItem> getItems() {
       return Collections
               .unmodifiableList(items);
   }
}

var list = new TodoList();
var items = list.getItems();

// Throws UnsupportedOperationException
items.add(new TodoItem());
```

Encapsulation?

FP solution:
First-class **functions**!

```
class TodoList {
   private final List<TodoItem> items =
           new ArrayList<>();

   public <R> List<R> mapItems(
           Function<TodoItem, R> f
   ) {
       return items.stream()
               .map(f)
               .toList();
   }
}

var list = new TodoList();
list.mapItems(
   item -> item.toString().toUpperCase()
);
```

Validation?

OOP solution:
Exceptions

```
data class ProductName(val value: String) {
    init {
        if (value.isBlank())
            throw IllegalArgumentException(
                "Name must not be blank"
            )
    }
}
```

Validation?

OOP solution:
Exceptions

```
data class ProductName(val value: String) {
    init {
        if (value.isBlank())
            throw IllegalArgumentException(
                "Name must not be blank"
            )
    }
}

try  {
    val name = ProductName("Some product")
    // ...
} catch (e: IllegalArgumentException) {
    // ...
}
```

Validation?

FP solution:
Values!

```
sealed interface Result<T>

class Success<T>(val value: T) : Result<T>
class Failure<T>(val message: String) : Result<T>
```

Validation?

FP solution:
Values!

```
sealed interface Result<T>

class Success<T>(val value: T) : Result<T>
class Failure<T>(val message: String) : Result<T>

data class ProductName(val value: String) {
   companion object {
       fun of(value: String): Result<ProductName> =
           when {
               value.isBlank() ->
                   Failure("Name must not be blank")
               else ->
                   Success(ProductName(value))
           }
   }
}
```

Validation?

FP solution:
Values!

```
sealed interface Result<T>

class Success<T>(val value: T) : Result<T>
class Failure<T>(val message: String) : Result<T>

data class ProductName(val value: String) {
   companion object {
       fun of(value: String): Result<ProductName> =
           when {
               value.isBlank() ->
                   Failure("Name must not be blank")
               else ->
                   Success(ProductName(value))
           }
   }
}

when (val name = ProductName.of("Cool product")) {
   is Success ->
       println("Valid product name")
   is Failure ->
       println("Invalid product name: ${name.message}")
}
```

$ whoami_

Dirk Groot (47)

Software Architect at Avisi

27+ years of software engineering

Arnhem

Avisi?

OOP considerations.

Inheritance is hard

```
open class Super {
   var counter: Int = 0
   fun inc1() { counter++ }
   open fun inc2() { counter++ }
}

class Sub: Super() {
   override fun inc2() { inc1() }
}

val s = Sub()
s.inc2()
println(s.counter)
```

OOP considerations.

Inheritance is hard

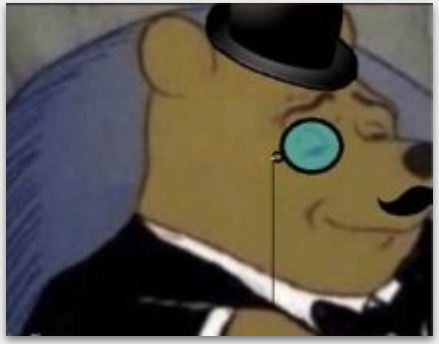
```
open class Super {
   var counter: Int = 0
   fun inc1() { inc2() } // <- Changed!
   open fun inc2() { counter++ }
}

class Sub: Super() {
   override fun inc2() { inc1() }
}

val s = Sub()
s.inc2()
println(s.counter)
```

FP considerations.

Side effects are hard



“A monad is just a monoid in the category of endofunctors. What’s the problem?”

FP considerations.

Lack of encapsulation

```
type BankAccount = double

let createAccount
   (initialBalance: double)
   : BankAccount =
   initialBalance

let withdraw
   (amount: double)
   (account: BankAccount)
   : BankAccount =
   if amount <= account then
       account - amount
   else
       account

let account = createAccount 100.0
let corruptedAccount = account - 1000.0
```