Functional programming in Java

•••

Dominik Moštěk

Claim #1

Java is not Object Oriented

Claim #2

OOP language is nonsense
(and does not exist)

* see my OOP Myths presentation - https://dominikmostek.cz/2021-11-12-oop.html

Claim #3

SOLID is functional

Claim #4

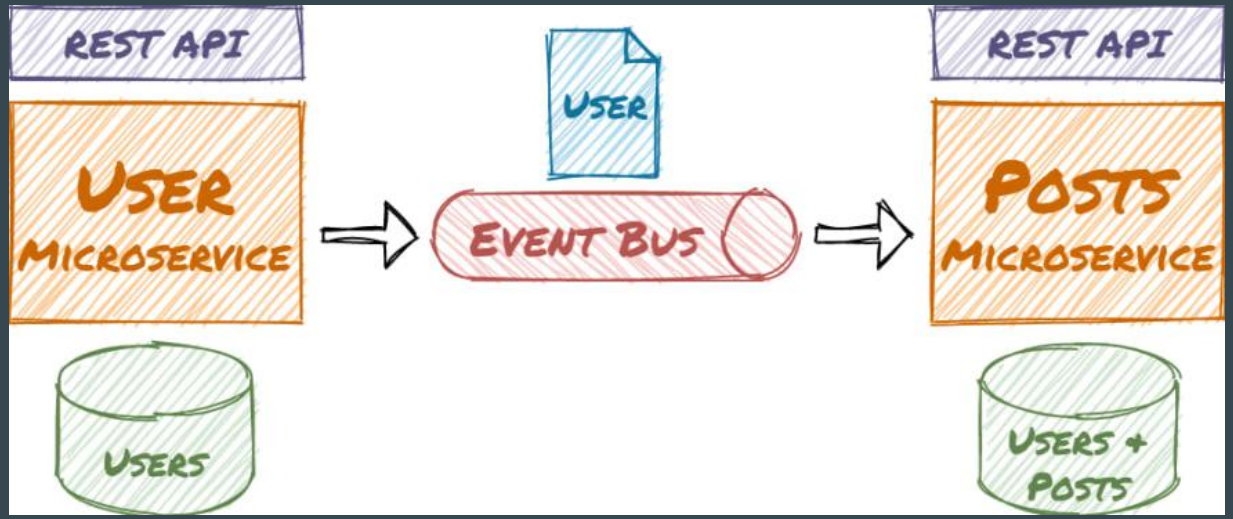
Java is all you need

Java is not Object Oriented

OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things ~ Alan Kay

Java = Classes + Main method

```java
public Id createUser(String username) {
    User u = new User();
    u.setUsername(username);
    return userRepository.save(u);
}
```



OOP language is nonsense

Messaging

Local retention and protection and hiding of state-process

Extreme late-binding of all things

SmallTalk

```
(someVariable > 0)
    ifTrue:
        [(someVariable < 10)
            ifTrue:
                [Transcript showCR:'between 1 and 9']
            ifFalse:
                [Transcript showCR:'positive']]
    ifFalse:
        [Transcript showCR:'zero or negative'].
```

SOLID is functional

S ingle-responsibility principle

O pen–closed principle

L iskov substitution principle

I nterface segregation principle

D ependency inversion principle

Single responsibility

Single responsibility = very small, usually one method = one function

One method class ~ function

```
new LengthOfInput(
  new TeeInput(
    new BytesAsInput(
      new TextAsBytes(
        new StringAsText(
          "Hello, world!"
        )
      )
    ),
    new FileAsOutput(
      new File("/tmp/hello.txt")
    )
  )
)
```

Open–closed principle

How to be open to extension?

Functional composition

Composing through constructors

```
new LengthOfInput(
  new TeeInput(
    new BytesAsInput(
      new TextAsBytes(
        new StringAsText(
          "Hello, world!"
        )
      )
    ),
    new FileAsOutput(
      new File("/tmp/hello.txt")
    )
  )
)
```

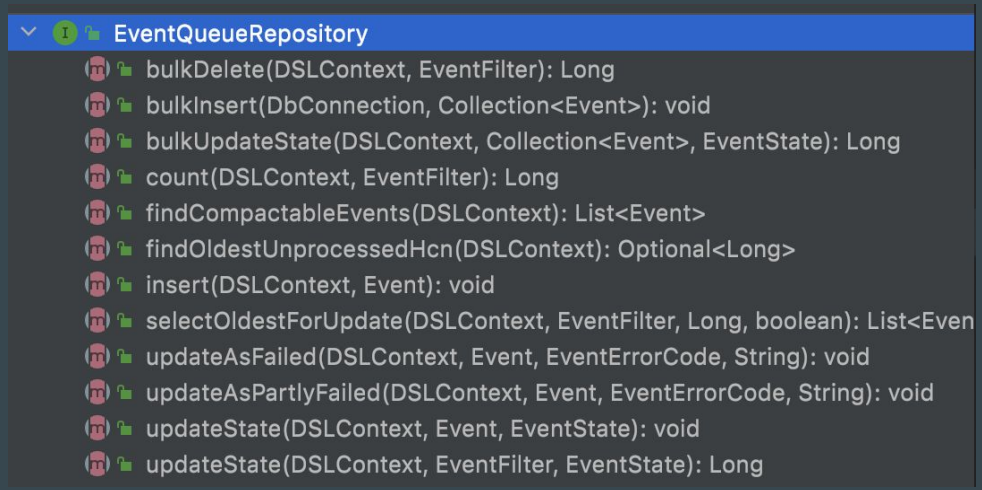
Interface segregation principle

“Clients should not be forced to depend upon interfaces that they do not use”

Small interfaces = functions

Export only public functions = do not depend on internal (no getters)

Interface segregation principle




```java
@Override
public void store(IMdTransaction transaction, Event event) {
    Assert.notNull(transaction.getDbConnection(), message: "transaction.dbConnection must not be null");

    eventCounter.increment();
    repository.insert(transaction.getDbConnection().getDsl(), event);
}
```

Interface segregation principle

```
public static class OnlyChangedFilter implements PolicyCacheOperation {
    private final Iterable<AssignedPolicyAlgorithmValues> values;

    public OnlyChangedFilter(Iterable<AssignedPolicyAlgorithmValues> values) {
        this.values = values;
    }
```

```
PolicyCacheOperation operation =
        new OnlyChangedFilter(() -> repository.values().iterator(),
```

Dependency inversion principle

High order functions

Again: composition through constructors

```
public Diff<PolicyAlgorithmValues> merge(Diff<PolicyAlgorithmValues> diff) {
    if (this.isEmpty()) {
        return diff;
    }
    if (diff.isEmpty()) {
        return this;
    }
    return new MergedDiff( left: this, diff);
}
```

Java is all you need

Functional =

Pure Functions

No state

No side effects

Functions as first-class entities

High order functions

Laziness

Immutability

Referential transparency

Pure Functions

Class instance is a set of partially applied functions

```
public static final class SomeObject {
    private final int iterations;

    // constructor = partial application
    public SomeObject(int iterations) {
        this.iterations = iterations;
    }
```

Pure Functions

```
public static class Foo {
    private final UserRepository userRepository;

    public Foo(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public void doSomething(String email) {
        User u = this.userRepository.findByEmail(email);
        // ...
    }
}
```

```java
public static class Foo {
    private final Iterable<User> users;

    public Foo(Iterable<User> users) {
        this.users = users;
    }

    public void doSomething(String email) {
        Optional<User> first = StreamSupport.stream(this.users.spliterator(), parallel: false)
                .filter(u -> u.hasEmail(email))
                .findFirst();
        // ...
    }
}
public void test(String email) {
    Foo foo = new Foo(Collections.singleton(userRepository.findByEmail(email)));
    foo.doSomething(email);
}
```

No state

Class instance is a set of partially applied functions

~~No state~~ No shared mutated state

Return results instead of state mutation

No side effects

```
PolicyCacheOperation operation = new ClearCacheOperation(
        repository,
        new UpdateRepositoryOperation(repository,
                new MergeResultsOperation(
                        new CascadedPolicyCacheOperation(
                                new ApplyDiffToAllDataOperation()))));
```

```
PolicyCacheOperation operation =
        new OnlyChangedFilter(repository,
                new CascadedPolicyCacheOperation(
                        new ApplyDiffToAllDataOperation()));
```

```
PolicyCacheOperation operation =
        new ClearAndInsertOperation(
                repository,
                new StartWithParentDiffOperation(
                        repository,
                        new CascadedPolicyCacheOperation(
                                new ApplyDiffToAllDataOperation())));
```

Functions as first-class citizens

Functions same as data

Again: Class instance is a set of partially applied functions

High order functions

Composition through constructors

Return values (no side effects)

No getters though :)

Laziness

```
public void insert(NodeReference nodeReference) {
    PolicyCacheOperation.DiffSource diffSource = node -> PolicyAlgorithmValuesDiff.add(
            new CachedPolicyAlgorithmValues(
                    new NodeRefWithSettingsValues(nodePolicySetting, node.asTraversingReference().asNodeReference())));
    PolicyCacheOperation operationForAffectedNodes = new CascadedPolicyCacheOperation(new ApplyDiffToAllDataOperation());
    PolicyCacheOperation operation =
            new ClearAndInsertOperation(
                    repository,
                    new StartWithParentDiffOperation(
                            repository,
                            new CascadedPolicyCacheOperation(
                                    new ApplyDiffToAllDataOperation())));
    PolicyCacheOperation affectedSettingsOperation =
            new UpdateRepositoryOperation(
                    repository,
                    new UpdateAffectedBySettings(
                            nodePolicySetting, nodeNameToNode,
                            PolicyAlgorithmValuesDiff::add,
                            operationForAffectedNodes));
    PolicyCacheOperation finalOperation =
            new CombinedPolicyCacheOperation(affectedSettingsOperation, operation);
    finalOperation.execute(nodeReference, diffSource);
}
```

Immutability

Immutability is a choice

Immutable by default

Referential transparency

Identity by value

```
class PortfolioServiceImpl implements PortfolioService {

    @Autowired
    private BitstampPriceApi bitstampPriceApi;
    @Autowired
    private WalletRepository walletRepository;

    public BigDecimal value() {
        return walletRepository.findById("id")
                .getBalance()
                .multiply(bitstampPriceApi.value());

    }

}
```

```
new WalletRepository().equals(new WalletRepository()); // ???
```

Referential transparency

JDK does not help much :(

But still possible

```
public Scalar<BigDecimal> portfolioValue() {
    return new Product(
            new BitcoinWallet(args),
            new Cached<>(
                    new BitstampPriceApi( key: "key")
            )
    );
}
```

```
class Product implements Scalar<BigDecimal> {
    private final Scalar<BigDecimal> x;
    private final Scalar<BigDecimal> y;

    Product(Scalar<BigDecimal> x, Scalar<BigDecimal> y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public BigDecimal value() { return x.value().multiply(y.value()); }
}

interface Scalar<T> {
    T value();
}
```

Real life example

```
public void clearAndInitPart(NodeReference nodeReference) {
    PolicyCacheOperation.DiffSource diffSource = node -> PolicyAlgorithmValuesDiff.add(
            new CachedPolicyAlgorithmValues(
                    new NodeRefWithSettingsValues(nodePolicySetting, node.asTraversingReference().asNodeReference())));
    PolicyCacheOperation operation =
            new ClearAndInsertOperation(
                    repository,
                    new StartWithParentDiffOperation(
                            repository,
                            new CascadedPolicyCacheOperation(
                                    new ApplyDiffToAllDataOperation())));
    operation.execute(nodeReference, diffSource);
}
```

Performance

Just-In-Time (JIT)

- Escape analysis
- Scalar replacement
- Tail call


Java is such a bad language
Have to switch to Kotlin
OOP is bad
Never learned writing good code in the first place