ThoughtWorks®

DevNexus '14

# Refactoring towards functional Java

John Napier

*1970 -*

*1970 - "...Lambdas are relegated to relative obscurity*

*1970 - "...Lambdas are relegated to relative obscurity until Java makes them popular by not having them."*

*1970 - "...Lambdas are relegated to relative obscurity until Java makes them popular by not having them."*

http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html

# What we’re going to be talking about today

# What we're going to be talking about today

* Key benefits

# What we're going to be talking about today

* Key benefits
* How to leverage OOP + FP

# What we're going to be talking about today

- Key benefits
- How to leverage OOP + FP
- Functional patterns in Java

# What we're going to be talking about today

- Key benefits
- How to leverage OOP + FP
- Functional patterns in Java
- Refactoring techniques

# What we're going to be talking about today

- Key benefits
- How to leverage OOP + FP
- Functional patterns in Java
- Refactoring techniques
- Performance implications

# Key Benefits

# Key Benefits

**Minimize side-effects**

# Key Benefits

**Minimize side-effects**

(Favor immutability)

# Key Benefits

**Minimize side-effects**

(Favor immutability)

* Principle of Least Surprise - an object's state won't change out from under you

# Key Benefits

**Minimize side-effects**

(Favor immutability)

- Principle of Least Surprise - an object's state won't change out from under you
- No more errors resulting from "bean-style" decoupling of object construction from object initialization

# Key Benefits

**Minimize side-effects**

(Favor immutability)

- Principle of Least Surprise - an object's state won't change out from under you
- No more errors resulting from "bean-style" decoupling of object construction from object initialization
- Thread Safety

# Key Benefits

## Declarative Syntax

# Key Benefits

**Declarative Syntax**

(Focus on the language of your problem domain)

# Key Benefits

**Declarative Syntax**

(Focus on the language of your problem domain)

- Functional semantics can help to de-emphasize non-obviously incidental implementation details

# Key Benefits

**Declarative Syntax**

(Focus on the language of your problem domain)

- Functional semantics can help to de-emphasize non-obviously incidental implementation details
- Refocus on the language of your problem domain

# Key Benefits

**Declarative Syntax**

(Focus on the language of your problem domain)

- Functional semantics can help to de-emphasize non-obviously incidental implementation details
- Refocus on the language of your problem domain
- Help identify conceptual duplication in your codebase

# Ways to Leverage OOP + FP

# Ways to Leverage OOP + FP

[ OOP in the large ]

# Ways to Leverage OOP + FP

**OOP in the large**

(FP in the small)

# Ways to Leverage OOP + FP

**OOP in the large**

(FP in the small)

Objects are still a very powerful way to represent your domain model and business interactions

# Ways to Leverage OOP + FP

**OOP in the large**

(FP in the small)

- Objects are still a very powerful way to represent your domain model and business interactions
- Functional style tends to shine brightest when used for discrete implementations (method bodies, etc.)

# Functional Patterns in Java

# Functional Patterns in Java

# Functional Patterns in Java

**Immutability**

(techniques for reducing side-effects)

# Functional Patterns in Java

**Immutability**

(techniques for reducing side-effects)

- Make your objects' fields final

# Functional Patterns in Java

## Immutability

(techniques for reducing side-effects)

- Make your objects' fields final
- Provide single, all-args constructors and companion Builders, Factories, or static factory methods for usability

# Functional Patterns in Java

## Immutability

(techniques for reducing side-effects)

- Make your objects' fields final
- Provide single, all-args constructors and companion Builders, Factories, or static factory methods for usability
- Avoid exposing access to internal, inherently mutable data structures like Lists and Maps, or use immutable alternate implementations

# Functional Patterns in Java

## Immutability

(techniques for reducing side-effects)

- Make your objects' fields final
- Provide single, all-args constructors and companion Builders, Factories, or static factory methods for usability
- Avoid exposing access to internal, inherently mutable data structures like Lists and Maps, or use immutable alternate implementations
- Utilize defensive copies where necessary

# Functional Patterns in Java

## Immutability

(techniques for reducing side-effects)

- Make your objects' fields final
- Provide single, all-args constructors and companion Builders, Factories, or static factory methods for usability
- Avoid exposing access to internal, inherently mutable data structures like Lists and Maps, or use immutable alternate implementations
- Utilize defensive copies where necessary
- Produce new instances of an object for update operations, leaving original instance unchanged

# Functional Patterns in Java

## Higher-level iteration patterns

# Functional Patterns in Java

## Higher-level iteration patterns

(Understanding Folds)

# Functional Patterns in Java

## Higher-level iteration patterns

(Understanding Folds)

Left folding / right folding

# Functional Patterns in Java

## Higher-level iteration patterns

(Understanding Folds)

- Left folding / right folding
- Map / Filter

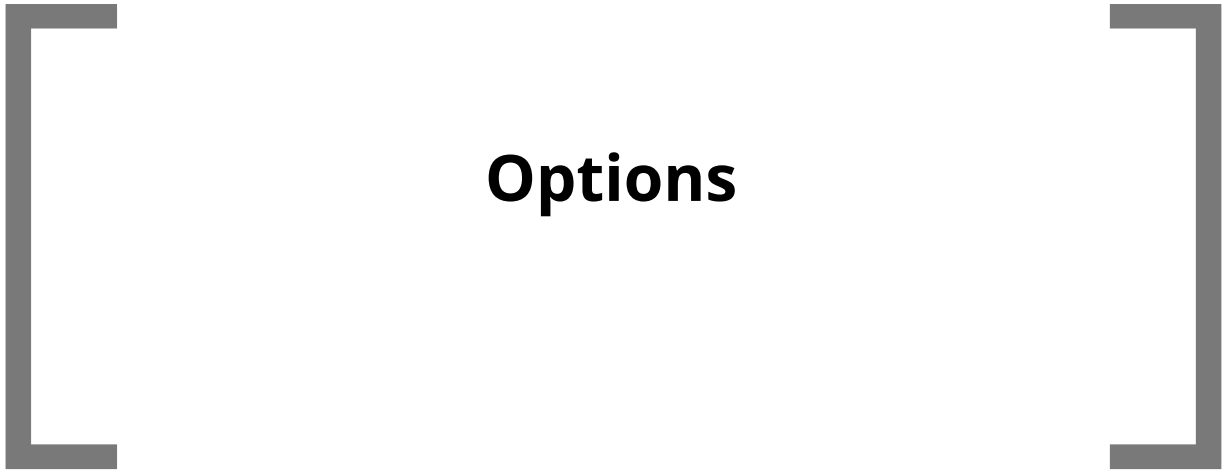
# Functional Patterns in Java

## Higher-level iteration patterns

(Understanding Folds)

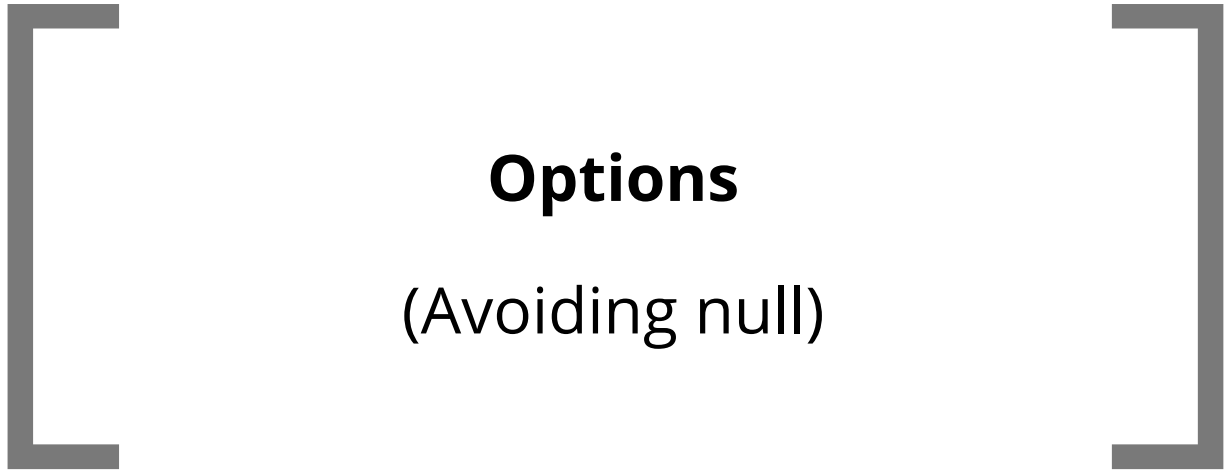
- Left folding / right folding
- Map / Filter
- Any / All / Length / Head / Last / Join / etc.

# Functional Patterns in Java



## Options

# Functional Patterns in Java



**Options**

(Avoiding null)

# Functional Patterns in Java

## Options

(Avoiding null)

- Use a meaningful representation of the presence or absence of a value

# Functional Patterns in Java

## Options

(Avoiding null)

- Use a meaningful representation of the presence or absence of a value
- Use Tell, Don't Ask for value transformations and failure-case recovery

# Functional Patterns in Java

## Options

(Avoiding null)

- Use a meaningful representation of the presence or absence of a value
- Use Tell, Don't Ask for value transformations and failure-case recovery
- Powerful companion to Null Objects and Special Case Objects

# Refactoring Techniques

# Refactoring Techniques

# Refactoring Techniques

**DAOs**

(Dealing with persistence)

# Refactoring Techniques

## DAOs

(Dealing with persistence)

- Return None<T> instead of null

# Refactoring Techniques

## DAOs

(Dealing with persistence)

- Return None<T> instead of null
- Update operations should return new instances

# Refactoring Techniques

## DAOs

(Dealing with persistence)

- Return None<T> instead of null
- Update operations should return new instances
- Leverage monadic types like Persisted<T>

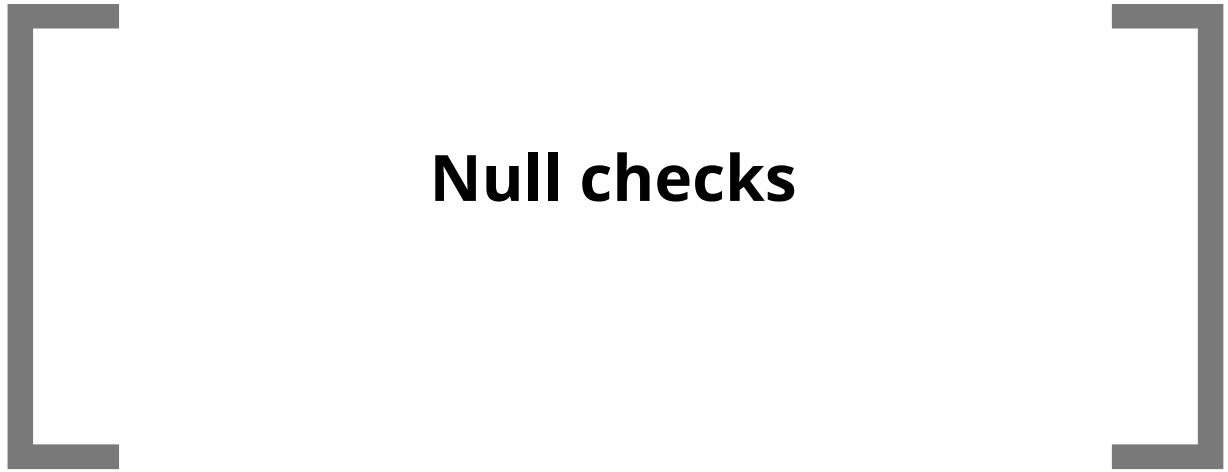
# Refactoring Techniques

## DAOs

(Dealing with persistence)

- Return None<T> instead of null
- Update operations should return new instances
- Leverage monadic types like Persisted<T>
- Use exceptions when the case is actually exceptional

# Refactoring Techniques



**Null checks**

# Refactoring Techniques

**Null checks**

(Replacing with Option)

# Refactoring Techniques

## Null checks

(Replacing with Option)

- Let Option resolve the possibly nullable values

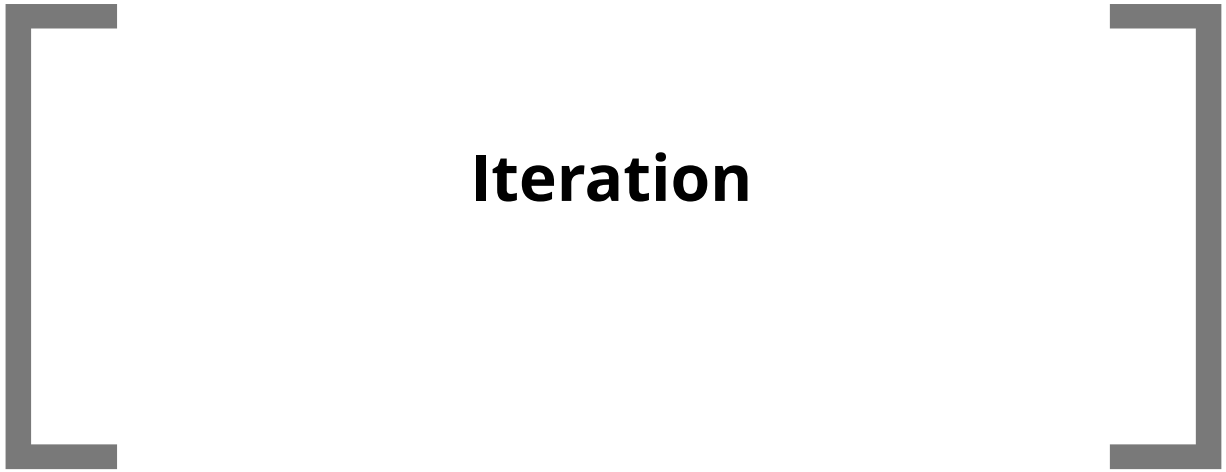
# Refactoring Techniques

## Null checks

(Replacing with Option)

- Let Option resolve the possibly nullable values
- Resolve an underlying value as early as possible

# Refactoring Techniques



# Refactoring Techniques

**Iteration**

(Using folds)

# Refactoring Techniques

## Iteration

(Using folds)

- Extract the body of your loop into a pure function

# Refactoring Techniques

## Iteration

(Using folds)

- Extract the body of your loop into a pure function
- Use the newly-created function's signature to inform you about what type of fold is best-suited for your case

# Refactoring Techniques

## Iteration

(Using folds)

- Extract the body of your loop into a pure function
- Use the newly-created function's signature to inform you about what type of fold is best-suited for your case
- Replace the loop with the fold

# Performance implications

# Performance implications

**There aren’t any! Yay!**

# Performance implications

**There aren’t any! Yay!**

(Just kidding.)

# Performance implications

**There are some.**

(No such thing as a free lunch.)

# Performance implications

**There are some.**

(No such thing as a free lunch.)

GC that implicitly supports mutation can’t optimize for immutability

# Performance implications

**There are some.**

(No such thing as a free lunch.)

- GC that implicitly supports mutation can’t optimize for immutability
- Stack overflows due to recursion

# Performance implications

**There are some.**

(No such thing as a free lunch.)

- GC that implicitly supports mutation can’t optimize for immutability
- Stack overflows due to recursion
- Stack overflows due to function composition

# Performance implications

**There are some.**

(No such thing as a free lunch.)

- GC that implicitly supports mutation can’t optimize for immutability
- Stack overflows due to recursion
- Stack overflows due to function composition
- YMMV. Take it on a case-by-case basis

# Questions?

ThoughtWorks®

# Thank You.

John Napier

jnapier@thoughtworks.com

@jnape