
goto;
chicago
Achieving FP in Java
John Napier
follow us @gotochgo

goto; chicago

Click ‘Rate Session’
to rate session
and ask questions.

follow us @gotochgo

DRW

Achieving FP in Java

John Napier

DRW

Background

- Software developer at DRW, working with Trading Infrastructure teams
- Maintainer of lambda

DRW

Trading Infrastructure teams

- Polyglot developers
- Ruby, C#, Clojure, Java
- ~30 Java applications built on lambda

DRW

Guiding Principles

- Constraints should be precisely stated via types
- Generic operations should have generic interfaces
- Lazy evaluation is a useful default
- Partial operations should be encoded as total operations
- Pure and impure operations should be separate

DRW

Guiding Principles

- Constraints should be precisely stated via types
- Generic operations should have generic interfaces
- Lazy evaluation is a useful default
- Partial operations should be encoded as total operations
- Pure and impure operations should be separate

```java
boolean exists(String id);
```

```
boolean exists(UUID id);
```

```
List<Integer> numericParts(Float f);
```

```
Tuple2<Integer, Integer> numericParts(Float f);
```

DRW

```java
public void process(List<Serializable> items) {
    for (Serializable item : items) {
        Integer value;
        if (item instanceof String) {
            value = ((String) item).length();
        } else if (item instanceof Integer) {
            value = (Integer) item;
        } else {
            throw new IllegalArgumentException("Only allows strings and ints");
        }
        // ...
    }
}
```

DRW

```java
public void process(List<CoProduct2<String, Integer, ?>> items) {
    for (CoProduct2<String, Integer, ?> item : items) {
        Integer value = item.match(String::length, integer -> integer);
        // ...
    }
}
```

```
Tuple2<Maybe<Error>, Maybe<Payload>> parseInput(String input);
```

```
Either<Error, Payload> parseInput(String input);
```

DRW

Guiding Principles

- Constraints should be precisely stated via types
- Generic operations should have generic interfaces
- Lazy evaluation is a useful default
- Partial operations should be encoded as total operations
- Pure and impure operations should be separate

DRW

Guiding Principles

- Constraints should be precisely stated via types
- Generic operations should have generic interfaces
- Lazy evaluation is a useful default
- Partial operations should be encoded as total operations
- Pure and impure operations should be separate

```java
Function<String, Integer>  function = String::length;
Optional<Integer>          optional = Optional.of(1);
Stream<Integer>            stream   = Stream.of(1, 2, 3);
CompletableFuture<Integer> future   = CompletableFuture.completedFuture(1);
```

DRW

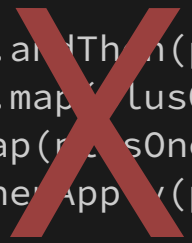
```java
Function<String, Integer>  function = String::length;
Optional<Integer>          optional = Optional.of(1);
Stream<Integer>            stream   = Stream.of(1, 2, 3);
CompletableFuture<Integer> future   = CompletableFuture.completedFuture(1);

Function<Integer, Float> plusOneToFloat = x -> x + 1F;
```

DRW

```java
Function<String, Integer>  function = String::length;
Optional<Integer>          optional = Optional.of(1);
Stream<Integer>            stream   = Stream.of(1, 2, 3);
CompletableFuture<Integer> future   = CompletableFuture.completedFuture(1);

Function<Integer, Float> plusOneToFloat = x -> x + 1F;

Function<String, Float>  mappedFunction = function.andThen(plusOneToFloat);
Optional<Float>          mappedOptional = optional.map(plusOneToFloat);
Stream<Float>            mappedStream   = stream.map(plusOneToFloat);
CompletableFuture<Float> mappedFuture   = future.thenApply(plusOneToFloat);
```

DRW

```
Function<String, Integer>  function = String::length;
Optional<Integer>          optional = Optional.of(1);
Stream<Integer>            stream   = Stream.of(1, 2, 3);
CompletableFuture<Integer> future   = CompletableFuture.completedFuture(1);

Function<Integer, Float> plusOneToFloat = x -> x + 1F;

Function<String, Float>  mappedFunction = function.andThen(plusOneToFloat);
Optional<Float>          mappedOptional = optional.map(plusOneToFloat);
Stream<Float>            mappedStream   = stream.map(plusOneToFloat);
CompletableFuture<Float> mappedFuture   = future.thenApply(plusOneToFloat);
```



DRW

```java
public interface Functor<A, F extends Functor<?, F>> {
    <B> Functor<B, F> fmap(Function<? super A, ? extends B> fn);
}

class ResponseEnvelope<Body> implements Functor<Body, ResponseEnvelope<?>> {

    @Override
    public <B> ResponseEnvelope<B> fmap(Function<? super Body, ? extends B> fn) {
        // ...
    }
}

class ImportJob<Result> implements Functor<Result, ImportJob<?>> {

    @Override
    public <B> ImportJob<B> fmap(Function<? super Result, ? extends B> fn) {
        // ...
    }
}
```

```java
ResponseEnvelope<String>  envelope       = /*...*/;
ResponseEnvelope<Integer> mappedEnvelope = envelope.fmap(String::length);

ImportJob<Integer> job       = /*...*/;
ImportJob<Float>   mappedJob = job.fmap(Integer::floatValue);
```

DRW

```java
Function<String, Integer>  function = String::length;
Optional<Integer>          optional = Optional.of(1);
Stream<Integer>            stream   = Stream.of(1, 2, 3);
CompletableFuture<Integer> future   = CompletableFuture.completedFuture(1);

// Not on Function, but it's easily doable!

Optional<Float> flatMappedOptional = optional
        .flatMap(x -> x % 2 == 0 ? Optional.of(x / 2F) : Optional.empty());

Stream<Float> flatMappedStream = stream
        .flatMap(x -> x % 2 == 0 ? Stream.of(x / 2F) : Stream.empty());

CompletableFuture<Float> flatMappedFuture = future
        .thenCompose(x -> x % 2 == 0
              ? completedFuture(x / 2F)
              : new CompletableFuture<Float>() {{
                  completeExceptionally(new IllegalStateException("oops"));
              }});
```

```java
public interface Monad<A, M extends Monad<?, M> extends Applicative<A, M> {
    // ...

    <B> Monad<B, M> flatMap(Function<? super A, ? extends Monad<B, M>> f);
}

Maybe<Integer> maybe       = just(1);
Maybe<Float>   mappedMaybe = just(1)
        .flatMap(x -> x % 2 == 0 ? just(x / 2F) : nothing());


Iterable<Maybe<Integer>> maybes  = asList(just(1), just(2), just(3));
// Just [1, 2, 3]
Maybe<Iterable<Integer>> flipped = sequence(maybes, Maybe::just);

Iterable<Either<String, Integer>> eithers = asList(right(1),
                                                   right(2),
                                                   right(3));
// Right [1, 2, 3]
Either<String, Iterable<Integer>> alsoFlipped = sequence(eithers, Either::right);
```

DRW

Guiding Principles

- Constraints should be precisely stated via types
- Generic operations should have generic interfaces
- Lazy evaluation is a useful default
- Partial operations should be encoded as total operations
- Pure and impure operations should be separate

DRW

Guiding Principles

- Constraints should be precisely stated via types
- Generic operations should have generic interfaces
- Lazy evaluation is a useful default
- Partial operations should be encoded as total operations
- Pure and impure operations should be separate

```java
public static List<LocalDate> daysBetween(LocalDate start, LocalDate end) {
    List<LocalDate> dates   = new ArrayList<>();
    LocalDate       current = start;
    while (!end.isBefore(current)) {
        dates.add(current);
        current = current.plusDays(1);
    }
    return dates;
}

LocalDate       today         = LocalDate.now();
LocalDate       distantFuture = LocalDate.of(3000, 12, 25);
List<LocalDate> lotsOfDays    = daysBetween(today, distantFuture);

// later...

List<LocalDate> whatWeWant = lotsOfDays.subList(0, 10);
```

DRW

```java
public static Iterable<LocalDate> daysBetween(LocalDate start, LocalDate end) {
    return takeWhile(lt(end), iterate(current -> current.plusDays(1), start));
}

LocalDate           today          = LocalDate.now();
LocalDate           distantFuture = LocalDate.of(3000, 12, 25);
Iterable<LocalDate> lotsOfDays     = daysBetween(today, distantFuture);

// later...

Iterable<LocalDate> whatWeWant = take(10, lotsOfDays); // only ever computed 10


List<LocalDate>     onHeap     = toCollection(ArrayList::new, whatWeWant);
```

DRW

Guiding Principles

- Constraints should be precisely stated via types
- Generic operations should have generic interfaces
- Lazy evaluation is a useful default
- Partial operations should be encoded as total operations
- Pure and impure operations should be separate

DRW

Guiding Principles

- Constraints should be precisely stated via types
- Generic operations should have generic interfaces
- Lazy evaluation is a useful default
- Partial operations should be encoded as total operations
- Pure and impure operations should be separate

```
public static Integer parseInt(String input) {
    // ...
}
```

DRW

```java
public static Maybe<Integer> parseInt(String input) {
    // ...
}
```

```
class FixMessage {
    // ...
    public static FixMessage parse(String input) {
        // ...
    }
}

String     input   = "8=FIX.4.4|9=126|35=A|49=theBroker.12…";
FixMessage message = FixMessage.parse(input);
FixMessage kaboom  = FixMessage.parse("malformed");
```

DRW

```
class FixMessage {
    // ...
    public static Either<Errors, FixMessage> parse(String input) {
        // ...
    }
}

String                     input   = "8=FIX.4.4|9=126|35=A|49=theBroker.12…";
Either<Errors, FixMessage> message = FixMessage.parse(input);
Either<Errors, FixMessage> whew    = FixMessage.parse("malformed");
```

DRW

Guiding Principles

- Constraints should be precisely stated via types
- Generic operations should have generic interfaces
- Lazy evaluation is a useful default
- Partial operations should be encoded as total operations
- Pure and impure operations should be separate

DRW

Guiding Principles

- Constraints should be precisely stated via types
- Generic operations should have generic interfaces
- Lazy evaluation is a useful default
- Partial operations should be encoded as total operations
- Pure and impure operations should be separate

```java
public static Iterable<String> readTenLongestWords(Path path) throws IOException {
    return Map.<String, Iterable<String>>map(line -> asList(line.split("\\W+")))
            .fmap(flatten())
            .fmap(map(String::toLowerCase))
            .fmap(distinct())
            .fmap(sortWith(comparing(String::length).reversed()))
            .fmap(take(10))
            .apply(Files.readAllLines(path));
}

// side-effect mixed with pure operation
Iterable<String> words = readTenLongestWords(Paths.get("/tmp/two_cities.txt"));
```

```java
public static Iterable<String> tenLongestWords(Iterable<String> lines) {
    return Map.<String, Iterable<String>>map(line -> asList(line.split("\\W+")))
            .fmap(flatten())
            .fmap(map(String::toLowerCase))
            .fmap(distinct())
            .fmap(sortWith(comparing(String::length).reversed()))
            .fmap(take(10))
            .apply(lines);
}

public static IO<List<String>> readLines(Path path) {
    return io(checked(() -> Files.readAllLines(path)));
}

IO<Iterable<String>> wordsIO = readLines(Paths.get("/tmp/two_cities.txt"))
        .fmap(Sandbox::tenLongestWords);

// at the end of the world...
Iterable<String> words = wordsIO.unsafePerformIO();
```

DRW

```
Iterable<Path> paths = asList(Paths.get("/tmp/two_cities.txt"),
                              Paths.get("/tmp/sun_also_rises.txt"));
Iterable<IO<List<String>>> readAllFiles = map(Sandbox::readLines, paths);

IO<Iterable<String>> parallelizableIO = sequence(readAllFiles, IO::io)
        .fmap(flatten())
        .fmap(Sandbox::tenLongestWords);

// at the end of the world, parallelized...
CompletableFuture<Iterable<String>> words = parallelizableIO.unsafePerformAsyncIO();
```

DRW

Guiding Principles

- Constraints should be precisely stated via types
- Generic operations should have generic interfaces
- Lazy evaluation is a useful default
- Partial operations should be encoded as total operations
- Pure and impure operations should be separate

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

```
public interface Functor<A, F extends Functor<?, F>> { /* … */ }


public interface Applicative<A, App extends Applicative<?, App>>
    extends Functor<A, App> { /* … */ }


public interface Traversable<A, T extends Traversable<?, T>>
    extends Functor<A, App> { /* … */ }


public interface Monad<A, M extends Monad<?, M>>
    extends Applicative<A, M> { /* … */ }


public interface Contravariant<A, C extends Contravariant<?, T>> { /* … */ }


public interface Profunctor<A, B, PF extends Profunctor<?, ?, PF>>
    extends Contravariant<A, Profunctor<?, B, PF>> { /* … */ }
```

DRW

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

```
Maybe<Integer>           maybe  = just(1);

Either<String, Boolean>  right  = right(true);

Try<Exception, Float>    try_   = trying(() -> 1F);

Choice2<String, Integer> choice = a("string");

These<String, Integer>   these  = both("string", 1);

Unit                     unit   = UNIT;
```

DRW

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

```
Fn2<Integer, Integer, Integer> add = Integer::sum;

Fn1<Integer, Fn1<Integer, Integer>> alsoAdd = add;

Fn1<Integer, Integer> add1 = add.apply(1);

Integer sum = add.apply(1, 2);

Fn0<Integer> deferredSum = add1.thunk(2);
```

DRW

```java
Semigroup<Integer> sumSg = Integer::sum;

Integer foldedSum = sumSg.foldLeft(0, asList(1, 2, 3));

Monoid<Integer> sumM = monoid(sumSg, 0);

Integer reducedSum = sumM.reduceLeft(asList(1, 2, 3));

Integer sumLengths = sumM.foldMap(String::length, asList("foo", "bar", "baz"));
```

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

```
Iterable<Integer>           nats            = iterate(nat -> nat + 1, 0);

Iterable<Integer>           evens           = filter(x -> x % 2 == 0, nats);

Iterable<Integer>           evenSquares     = map(x -> x * x, evens);

Iterable<Integer>           firstHundred    = take(100, evenSquares);

Iterable<Integer>           greaterThan2500 = dropWhile(lte(2500), firstHundred);

Iterable<Iterable<Integer>> groupedInTens   = inGroupsOf(10, greaterThan2500);

Iterable<Integer>           sums            = map(foldLeft(Integer::sum, 0),
                                                  groupedInTens);
```

DRW

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

```
HList.HNil                nil           = HList.nil();
SingletonHList<String> singleton     = singletonHList("singleton");
SingletonHList<String> alsoSingleton = nil.cons("also singleton");

Tuple3<Float, String, Integer>     tuple3  = nil.cons(1).cons("two").cons(3f);
Tuple4<Byte, Short, Integer, Long> tuple4  = tuple((byte) 1, (short) 2, 3, 4L);
Integer                            integer = tuple4._3();
Tuple3<Short, Integer, Long>       tail    = tuple4.tail();
```

DRW

```java
HMap emptyHMap = HMap.emptyHMap();

TypeSafeKey.Simple<String>  fooKey = typeSafeKey();
TypeSafeKey.Simple<Integer> barKey = typeSafeKey();

HMap updated = emptyHMap
        .put(fooKey, "string")
        .put(barKey, 1);

boolean containsBar = updated.containsKey(barKey);

Maybe<Integer> maybeBar   = updated.get(barKey);
String         fooOrError = updated.demand(fooKey);
```

DRW

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

```
Lens.Simple<List<Integer>, Maybe<Integer>> firstElement = elementAt(0);

Fn1<List<Integer>, Maybe<Integer>> viewFn = view(firstElement);
Maybe<Integer>                     empty  = view(firstElement, emptyList());
Maybe<Integer>                     just1  = view(firstElement, asList(1, 2, 3));

List<Integer> updated = over(firstElement,
                             maybeX -> maybeX.fmap(x -> x + 1),
                             asList(1, 2, 3));

List<Integer> updatedDifferently = set(firstElement, just(1), emptyList());
```

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

```
public static IO<Unit> threadLog(Object message) {
    return io(() -> System.out.println(currentThread().getName()
                                       + ": " + message));
}


IO<Unit> helloWorld = threadLog("hello, “)
                        .flatMap(constantly(threadLog(“world!")));

Unit unit = helloWorld.unsafePerformIO();
```

DRW

```
public static IO<Unit> threadLog(Object message) {
    return io(() -> System.out.println(currentThread().getName()
                                       + ": " + message));
}


IO<Unit> helloWorld = threadLog("hello, “)
                        .flatMap(constantly(threadLog(“world!")));

CompletableFuture<Unit> runningFuture = helloWorld.unsafePerformAsyncIO();
```

DRW

DRW

Lambda Offerings

- A model for functors, applicative functors, monads, and more
- Common algebraic data types like `Maybe` and `Either`
- Curried functions with specializations like `Semigroup` and `Monoid`
- A rich library of functional iteration patterns like `map` and `filter`
- Type-safe heterogeneous data structures like `HList` and `HMap`
- Profunctor optics like `Lens` and `Iso`
- An `IO` monad

DRW

Demo

λ

https://github.com/palatable/lambda
https://gitter.im/palatable/lambda

goto; chicago

Please

Remember to rate this session

Thank you!

follow us @gotochgo