The Conference for Java
& Software Innovation
Oct 10 – 12, 2016 | London

Stephen Colebourne | OpenGamma

Java SE 8 Library Design

Stephen Colebourne

- Java Champion, regular conference speaker
- Best known for date & time - Joda-Time and JSR-310
- More Joda projects - http://www.joda.org
- Major contributions in Apache Commons
- Blog - http://blog.joda.org
- Worked at OpenGamma for 6 years

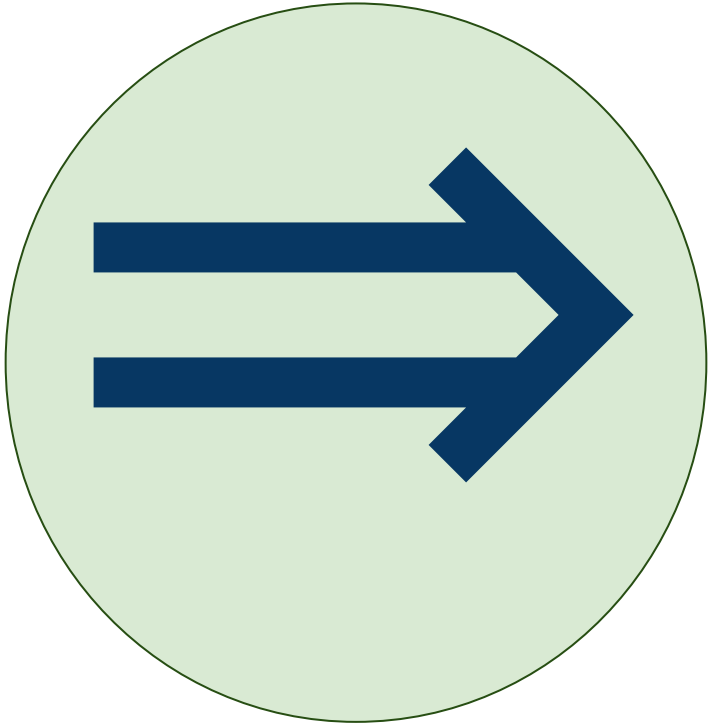
Strata, from OpenGamma

- Open Source market risk library
- Valuation and risk calcs for finance
  - interest rate swap, FRA, CDS
- Great example of Java SE 8 coding style

http://strata.opengamma.io/

OpenGamma

Introduction



Introduction

- Java SE 8 is a major update to Java
- Major new features
  - Lambdas
  - Streams
  - Methods on interfaces
  - Date and Time

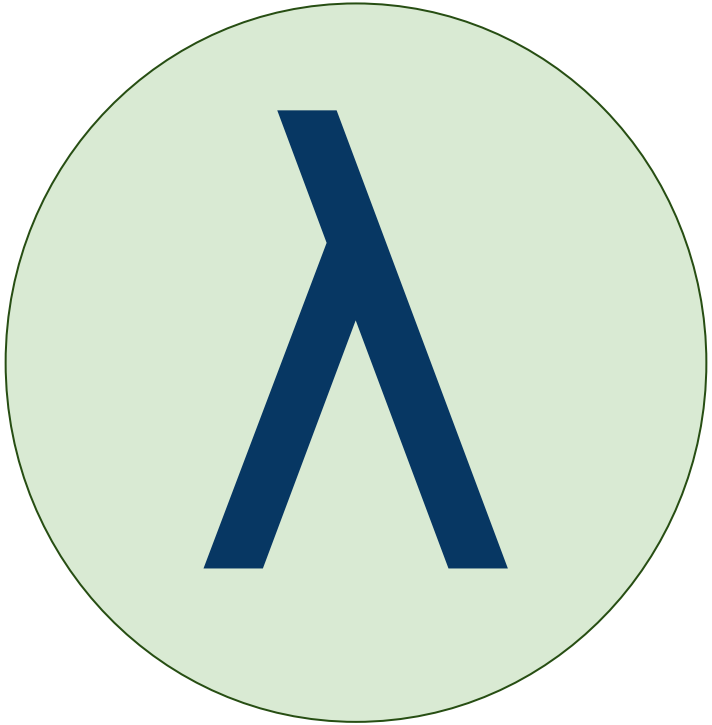
Introduction

- Essential to rethink how you code
- Reconsider coding conventions
- Appreciate new design options

Agenda

- Lambdas
- Streams
- Design with Lambdas
- Abstraction
- Immutability
- Interfaces
- Optional
- Odds and Ends

Lambdas



Lambdas

- Block of code
  - like an anonymous inner class
- Always assigned to a *Functional Interface*
  - an interface with one abstract method
  - Runnable, Callable, Comparator
- Uses *target typing*
  - context determines type of the lambda

Lambdas

```
public interface Comparator<T> {
    int compare(T obj1, T obj2);
}
// Java 7
Collections.sort(people, new Comparator<Person>() {
  @Override
  public int compare(Person p1, Person p2) {
    return p1.name.compareTo(p2.name);
  }
});
```

Lambdas

```
public interface Comparator<T> {
    int compare(T obj1, T obj2);
}
// Java 7
Collections.sort(people, new Comparator<Person>() {
  @Override
  public int compare(Person p1, Person p2) {
    return p1.name.compareTo(p2.name);
  }
});
```

Lambdas

```
public interface Comparator<T> {
    int compare(T obj1, T obj2);
}
// Java 8
people.sort((p1, p2) -> p1.name.compareTo(p2.name));
```

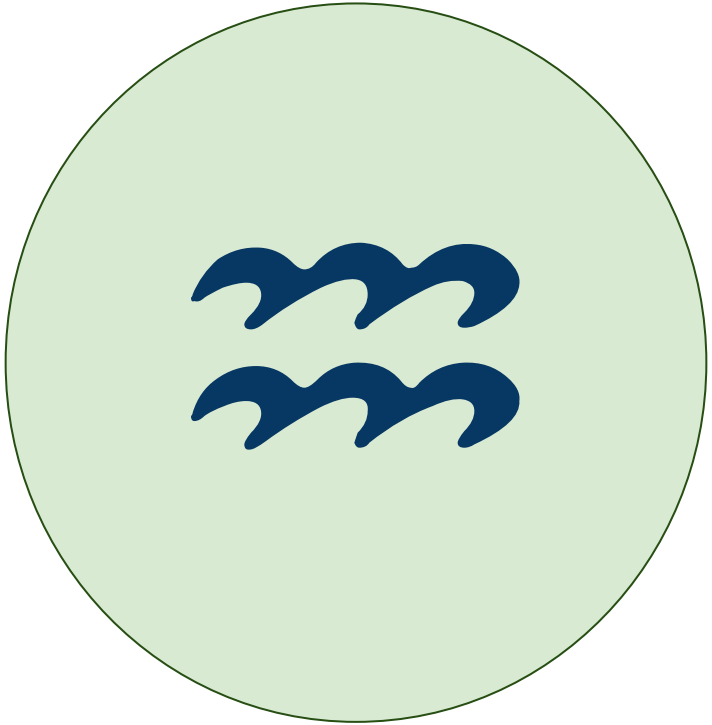
Top tips for Lambdas

- Use lambdas wherever appropriate
- Mostly, they just work
- Sometimes, the compiler needs a hint
  - use local variable
  - add the type to the lambda parameter

Lambdas

Lambdas affect code
but do they affect design?

Streams



Streams

- Many loops have a similar "shape"
- Repetitive *design patterns*
- Stream library provides a way to abstract this
- Lambdas used to pass the interesting bits

Streams

```
List<Trade> trades = loadTrades();
List<Money> valued = new ArrayList<>();
for (Trade t : trades) {
  if (t.isActive()) {
    Money pv = t.presentValue();
    valued.add(pv);
  }
}
```

Streams

```
List<Trade> trades = loadTrades();
List<Money> valued = new ArrayList<>();
for (Trade t : trades) {
  if (t.isActive()) {
    Money pv = t.presentValue();
    valued.add(pv);
  }
}
```

Streams

```
List<Trade> trades = loadTrades();
List<Money> valued =
  trades.stream()
        .filter(t -> t.isActive())
        .map(t -> t.presentValue())
        .collect(Collectors.toList());
```

Streams

- New `stream()` method on `Collection`
- Sequence of operations on underlying data
- Logic passed in using a lambda
  - filter() to retain/remove
  - map() to change
  - reduce() to summarise
  - sorted() to sort using a comparator

Streams

```
trades.stream()
      .filter(t -> t.isActive())
      .map(t -> t.presentValue())
      .collect(Collectors.toList());
```

Streams

```
trades.stream()
        .filter(new Predicate<Trade>() {
            public boolean test(Trade t) {
              return t.isActive();
            }
          })
        .map(new Function<Trade, Money>() {
            public Money apply(Trade t) {
              return t.presentValue();
            }
          })
        .collect(Collectors.toList());
```

Streams

Stream API not practical without lambdas

Exceptions in Streams

- For-each loop is a language feature
- Streams are implemented using regular methods
- Big difference in stack traces

Exceptions in Streams

```
java.lang.IllegalArgumentException: Oops
    at com.opengamma.strata.calc.DefaultCalculationRunner.lambda$2(DefaultCalculationRunner.java:98)
    at java.util.stream.ReferencePipeline$11$1.accept(ReferencePipeline.java:372)
    at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline.java:193)
    at java.util.Iterator.forEachRemaining(Iterator.java:116)
    at java.util.Spliterators$IteratorSpliterator.forEachRemaining(Spliterators.java:1801)
    at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:481)
    at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:471)
    at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:708)
    at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
    at java.util.stream.ReferencePipeline.collect(ReferencePipeline.java:499)
    at com.opengamma.strata.calc.DefaultCalculationRunner.calculate(DefaultCalculationRunner.java:100)
    at com.opengamma.strata.calc.DefaultCalculationRunner.lambda$0(DefaultCalculationRunner.java:86)
    at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline.java:193)
    at java.util.Iterator.forEachRemaining(Iterator.java:116)
    at java.util.Spliterators$IteratorSpliterator.forEachRemaining(Spliterators.java:1801)
    at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:481)
    at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:471)
    at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:708)
    at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
    at java.util.stream.ReferencePipeline.collect(ReferencePipeline.java:499)
    at com.opengamma.strata.calc.DefaultCalculationRunner.calculate(DefaultCalculationRunner.java:87)
    at com.opengamma.strata.calc.DefaultCalculationRunnerTest.calculate(DefaultCalculationRunnerTest.java:49)
```

Stack trace of inner stream

Stack trace of outer stream

Exceptions in Streams

```
java.lang.IllegalArgumentException: Oops
    at com.opengamma.strata.calc.DefaultCalculationRunner.calculate(DefaultCalculationRunner.java:102)
    at com.opengamma.strata.calc.DefaultCalculationRunner.calculate(DefaultCalculationRunner.java:87)
    at com.opengamma.strata.calc.DefaultCalculationRunnerTest.calculate(DefaultCalculationRunnerTest.java:49)
```

Stack trace of for-each loop

Top tips for streams

- Stream not always more readable than loop
- Stream exceptions can be much worse
- My advice:
  - use streams for small, localized, pieces of logic
  - be cautious using streams for large scale logic
- Strata uses for-each loops at top level
  - solely for shorter stack traces

Design with Lambdas

Design with Lambdas

- Lambda is converted to a *functional interface*
- Normal interface with one abstract method
- Java SE 8 adds many new functional interfaces
  - Function<T, R>
  - Predicate<T>
  - Supplier<T>
  - Consumer<T>
  - see java.util.function package
- Primitive versions only for **`long`**, **`int`**, **`double`**

Functional interfaces

- Learn the standard functional interfaces
- Only create new ones if adding additional value
  - lots of parameters
  - mix of primitive and object parameters
  - feature really needs a good name or Javadoc

Functional interface example

```
// API functional interface
@FunctionalInterface
public interface Perturbation {
  public abstract double perturb(int index, double value);
}
// API method that can be used by a lambda
public Curve perturbed(Perturbation perturbation) { … }
// caller code
curve = curve.perturbed((i, v) -> v + 1e-4);
```

Functional interface example

```
// API functional interface
@FunctionalInterface
public interface Perturbation {
  public abstract double perturb(int index, double value);
}
// API method that can be used by a lambda
public Curve perturbed(Perturbation perturbation) { … }
// caller code
curve = curve.perturbed((i, v) -> v + 1e-4);
```

Name has meaning
Method signature is complex

Time-series example

- A time-series stores changes to a value over time
- Date-based one like `Map<LocalDate, Double>`
- What if you want to change the values?

Time-series example

```
// API method that can be used by a lambda
public LocalDateDoubleTimeSeries
                    mapValues(DoubleUnaryOperator mapper){ … }


// caller code
ts = ts.mapValues(v -> v * 2);
```

Time-series example

```
// API method that can be used by a lambda
public LocalDateDoubleTimeSeries
                    mapValues(DoubleUnaryOperator mapper){ … }

// caller code
ts = ts.mapValues(v -> v * 2);
```

Multiplication - no need for `multipliedBy(double)` on the API

Time-series example

```
// API method that can be used by a lambda
public LocalDateDoubleTimeSeries
                    mapValues(DoubleUnaryOperator mapper){ … }

// caller code
ts = ts.mapValues(v -> v * 2);
ts = ts.mapValues(v -> v / 4);
```

Multiplication - no need for **`multipliedBy(double)`** on the API

Division - no need for **`dividedBy(double)`** on the API

Design with Lambdas

Method taking a lambda can be more flexible design

Abstraction with Lambdas

Abstraction

- Two or more classes with the same methods
- Abstract using an interface?
- Lambdas provide an alternative
- Consider an example with static methods
  - no way to abstract that with interfaces...

Abstraction

```
// standard API producing results
public static Money pv(FraTrade trade, Market md) { … }
public static Sens pv01(FraTrade trade, Market md) { … }
public static Double par(FraTrade trade, Market md) { … }
```

Abstraction

```
// standard API producing results
public static Money pv(FraTrade trade, Market md) { … }
public static Sens pv01(FraTrade trade, Market md) { … }
public static Double par(FraTrade trade, Market md) { … }
// functional interface matching all three methods
interface Calc<T> {
  public abstract T invoke(FraTrade trade, Market market);
}
```

Abstraction

```
// create abstraction to access method by "measure" key
Map<Measure, Calc> CALCS = ImmutableMap.builder()
    .put(Measures.PRESENT_VALUE, FraCalcs::pv)
    .put(Measures.PV01, FraCalcs::pv01)
    .put(Measures.PAR_RATE, FraCalcs:: par)
    .build();
// can now invoke using measure
return CALCS.get(measure).invoke(trade, market);
```

Abstraction

- Class being abstracted was not changed
- Provides way to abstract over code you do not own
- Less need for reflection

Design with Lambdas

Lambdas can abstract
in dynamic/flexible ways

Immutability

Multi-threaded

- JDK provides many tools for concurrency
- Parallel streams makes it even easier
- But parallel code is not simple to get right

Thread problems

- What if trade modified by some other piece of code?
- Check-then-act bug

```
List<Trade> trades = loadTrades();
List<Money> valued =
  trades.stream()
        .filter(t -> t.isActive())
        .map(t -> presentValue(t))
        .collect(Collectors.toList());
```

Check

then Act

Immutable

- Threading bugs due to shared mutable state
- One solution is to use **immutable beans**
- No possibility of check-then-act type bug

Immutable beans

- Class should be final
  - no subclasses
- Fields must be final
  - needed for Java Memory Model
- Field types should be immutable
  - eg. don't use java.util.Date
- Factory methods and Builders instead of constructors

Immutable beans

- IDEs help you write mutable beans
- Need better tooling for immutable beans
  - AutoValue
  - Immutables.org
  - Joda-Beans
- Strata uses Joda-Beans

http://www.joda.org/joda-beans

Joda-Beans

```
@BeanDefinition
public final TradeInfo implements ImmutableBean {
  /** The trade identifier. */
  @PropertyDefinition(validate = "notNull")
  private final StandardId tradeId;
  /** The trade date. */
  @PropertyDefinition(validate = "notNull")
  private final LocalDate tradeDate;
}
```

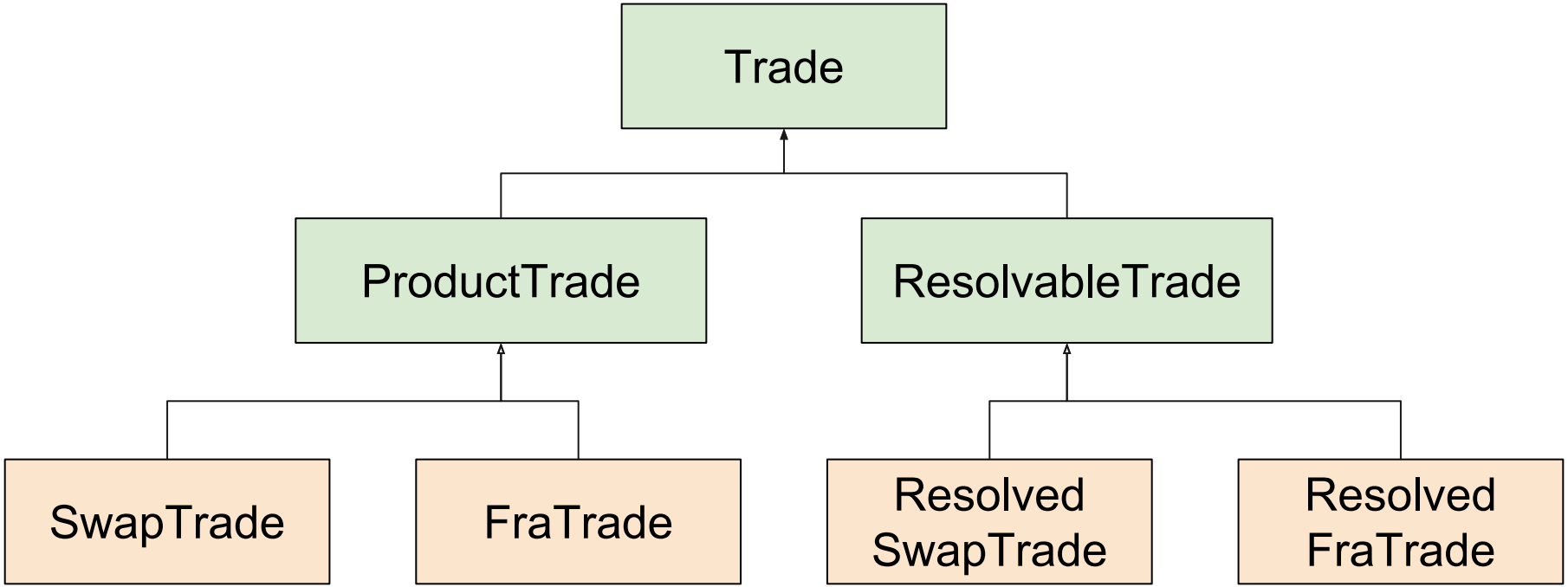
Joda-Beans

- Source code generated for
  - getters
  - builder
  - equals/hashCode/toString
  - properties - like C#
- Can add your own code to the class and still regenerate
- Built in XML, JSON and Binary serialization

Immutable beans

- Most systems better using immutability everywhere
- Java SE 8 `parallelStream()` pushes at this
- Threading issues mostly eliminated
- No class hierarchies, use interfaces

Use interfaces



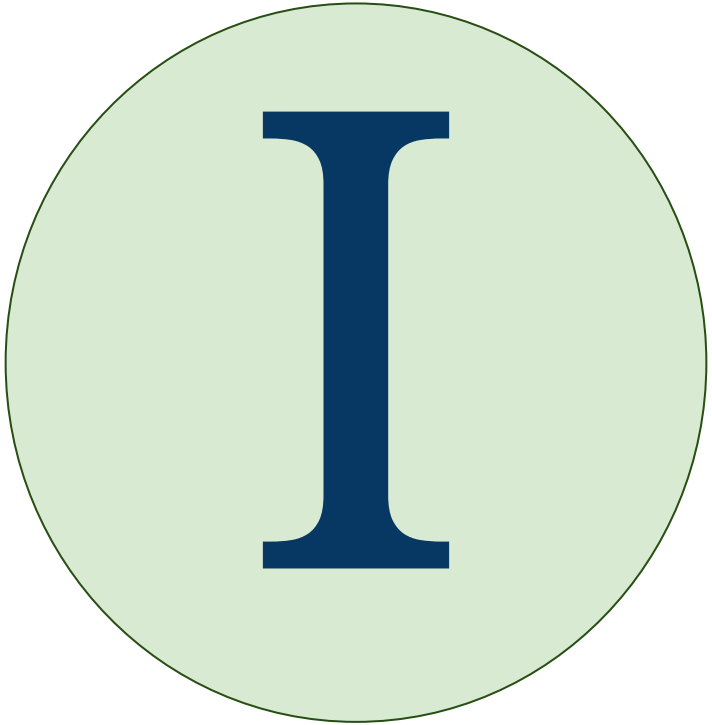
Use interfaces

- Concrete classes with no hierarchies
- Interfaces provide the hierarchy
- Methods on interfaces make this practical
- All implementations of interface should be immutable
  - "Implementations must be immutable and thread-safe beans."
- Strata uses immutable beans everywhere

Immutability

It is time to move on
from mutable data objects

Interfaces



Interfaces

- Two changes to interfaces
- Default methods
  - normal method, but on an interface
  - cannot default equals/hashCode/toString
- Static methods
  - normal static method, but on an interface

Coding Style

- Use modifiers in interfaces
- Much clearer now there are different types of method
- Prepares for private methods in Java SE 9

```
public interface Foo {
  public static of(String id) { … }
  public abstract isEmpty();
  public default isNotEmpty() { … }
}
```

Interfaces

- Methods on interfaces changes design
- Interfaces are part of macro-design
  - lambdas and streams affect micro-design
- Strata uses default and static methods liberally

Holiday Calendar

- Strata interface to specify which days are holidays

```
public interface HolidayCalendar {
  // a normal abstract interface method
  public abstract isHoliday(LocalDate date);
  …
}
```

Holiday Calendar

- Default methods make the interface more useful

```
public interface HolidayCalendar {
  public abstract isHoliday(LocalDate date);
  // check if date is a business day
  public default isBusinessDay(LocalDate date) {
    return !isHoliday(date);
  }
}
```

Holiday Calendar

- Default methods make the interface more useful

```
public interface HolidayCalendar {
  public abstract isHoliday(LocalDate date);
  // find the next business day
  public default next(LocalDate date) {
    LocalDate nextDay = date.plusDays(1);
    return isHoliday(nextDay) ? next(nextDay) : nextDay;
  }
}
```

Holiday Calendar

- Static methods avoid **`HolidayCalendarFactory`**

```
public interface HolidayCalendar {
  // find holiday calendar by identifier such as DKCO
  public static of(String id) {
    // lookup calendar
  }
}
```

Holiday Calendar

- Interface used just as would be expected

```
// find holiday calendar by identifier such as DKCO
HolidayCalendar cal = HolidayCalendar.of("DKCO");

// use calendar to select the trade start date
LocalDate startDate = cal.next(tradeDate);
```

Interfaces

- Interface now acts as abstract class
  - Only need abstract class if need abstracted state
  - but abstracted state is generally a bad idea
- Interface can now acts as a factory
  - Not suitable for all factory use cases*

* In Strata, holiday calendars are not really fixed at startup, but it made a good example for this talk!

Package-scoped implementation

- Can the interface be the only public API?
- Can the implementation class be package-scoped?

- Strata uses this pattern a lot

Interfaces

Consider package-scoped factory and implementation

Optional and null

Optional and null

- New class `Optional` added to Java 8
- Opinions are polarized
  - some think it is the saviour of the universe
  - others think it is useless
- Used pragmatically, can be very useful

Optional and null

- Simple concept - two states
  - present, with a value - Optional.of(foo)
  - empty - Optional.empty()

Optional and null

- Standard code using null

```
// library, returns null if not found
public Foo getValue(String key) { … }


// application code must remember to check for null
Foo foo = getValue(key);
if (foo == null) {
  foo = Foo.DEFAULT;  // or throw an exception
}
```

Optional and null

- Standard code using Optional

```
// library, returns Optional if not found
public Optional<Foo> findValue(String key) { … }

// application code
Foo foo = findValue(key).orElse(Foo.DEFAULT);
// or
Foo foo = findValue(key).orElseThrow( … );
```

Optional and null

- Important that a variable of type Optional is never null
- Prefer methods like `map()` and `orElse()`
- Minimise use of `isPresent()`

Optional

- Strata often uses set of 3-methods

```
public abstract Optional<T> findValue(DataId<T> id);
public default boolean containsValue(DataId<T> id) {
  return findValue(id).isPresent();
}
public default T getValue(DataId<T> id) {
  return findValue(id).orElseThrow(
    () -> new MarketDataException());
}
```

Optional

- Optional is a class
- Some memory/performance cost to using it
- Not serializable
- Not ideal to be an instance variable
- JDK authors added it for return types
- Use in parameters often annoying for callers
- Use as return type gets best value from concept

http://blog.joda.org/2015/08/java-se-8-optional-pragmatic-approach.html

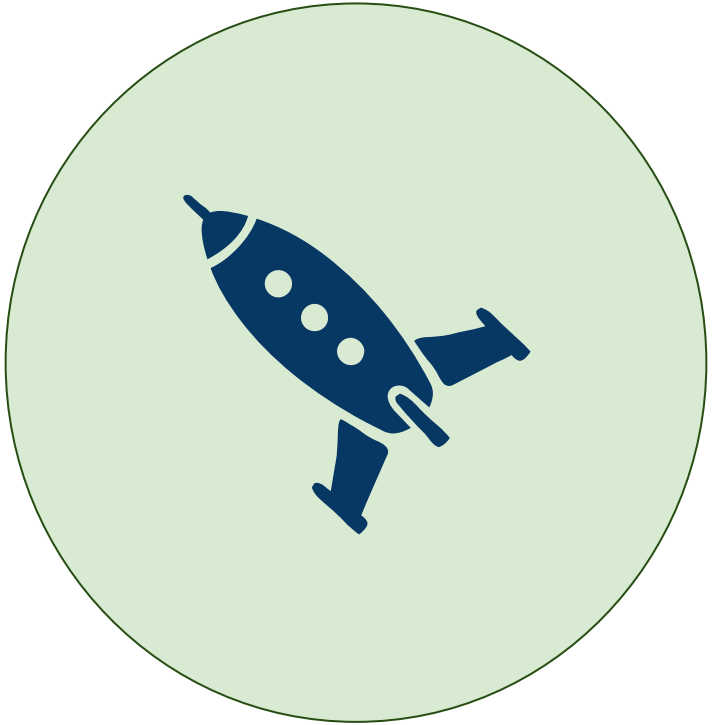
Optional in Strata

- Strata has no exposed nulls
- No part of the API will return null
- `Optional` used when something is optional
- Pragmatically, null is used within classes

Optional

Use Optional in a pragmatic way

Odds and Ends



Java SE 8 version

- Use Java SE 8 update 40 or later
  - preferably use the latest available
- Earlier versions have annoying lambda/javac issues

Internal JDK packages

- Java SE 9 will remove access to some JDK packages
  - sun.*
  - com.sun.*
  - com.oracle.*
- Now is the time to prepare for this
  - Avoid **`sun.misc.Unsafe`**
  - Stick to the standard JDK API

Parameters

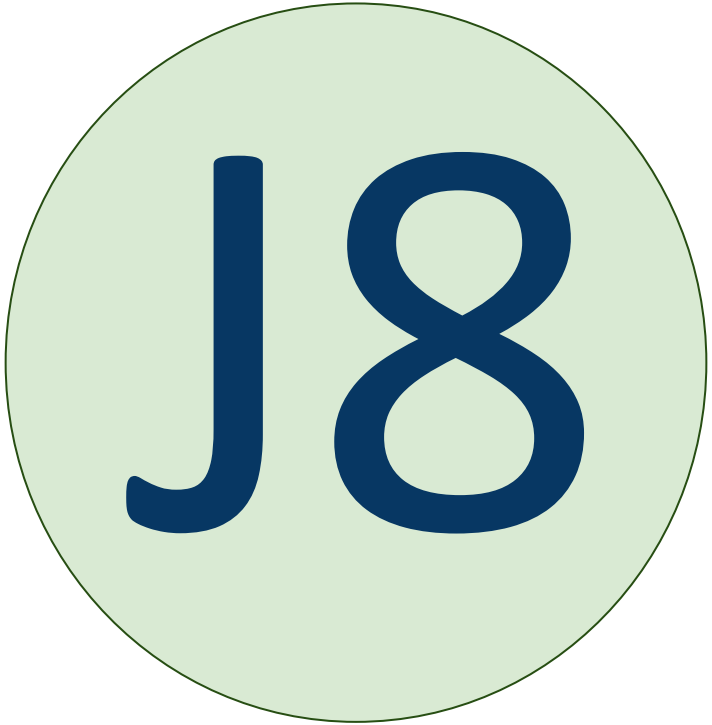
- Java SE 8 can reflect on parameter names
- Avoids need for additional libraries like paranamer
- Not enabled by default, must choose to include data

Checked exceptions

- Checked exceptions can be made to disappear
- Helper methods can convert to runtime exceptions

```
Unchecked.wrap(() -> {
   // any code that might throw a checked exception
   // converted to a runtime exception
});
```

Summary



Summary

- Lots of good stuff in Java SE 8
- Design and coding standards change
- Lots more potential to abstract, but don't over-use
- Methods on interfaces add a lot of power

Key Strata design features

- Immutable data objects, using Joda-Beans
- Static methods on interfaces, package-scope impls
- Make use of new abstractions
- Beware stack traces with streams
- Pragmatic use of Optional, null never returned from API

Work in Finance?

- Take a look at OpenGamma Strata
  - developed from the ground up in Java 8
  - lots of good Java 8 techniques and utilities
- High quality library for market risk
  - day counts, schedules, holidays, indices
  - models and pricing for swaps, FRAs, swaptions, FX, futures…
  - open source and stable release v1.1

http://strata.opengamma.io/

OpenGamma