Java SE 8 Best Practices

A personal viewpoint

Stephen Colebourne, February 2017

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
- Twitter - **@jodastephen**
- 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

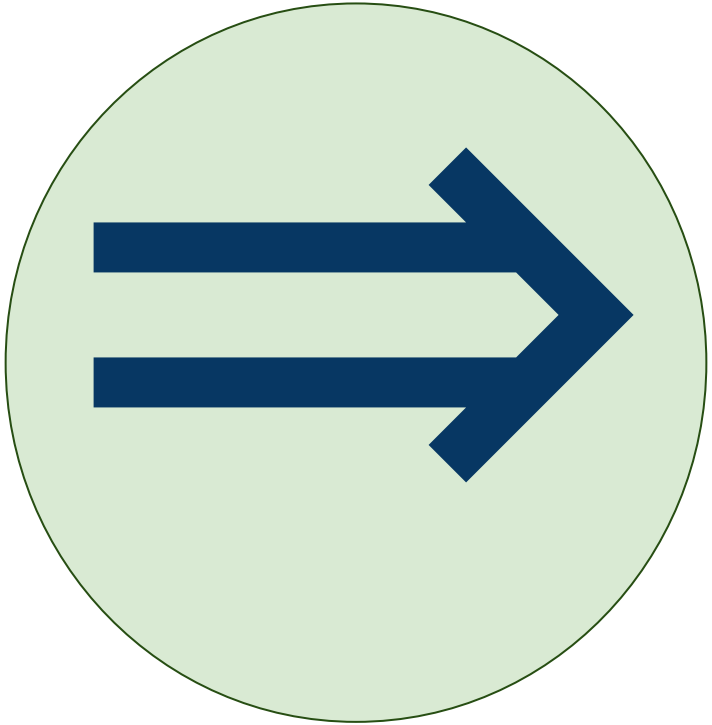
http://strata.opengamma.io/

OpenGamma

Agenda

- ⇒ Introduction
- λ Lambdas
- f(x) Functional interfaces
- ! Exceptions
- ? Optional
- ♒ Streams
- I Interfaces
-  Date and Time
-  Extras

Introduction



Introduction

- What is a Best Practice?

Introduction

- What is a Best Practice?

"commercial or professional procedures that are accepted or prescribed as being correct or most effective"

Introduction

- What is the Best Practice for Java SE 8?

Introduction

- What is the Best Practice for Java SE 8?

"whatever I say in the next 50 minutes"

Introduction

- Software Best Practice is mostly opinion
- Different conclusions perfectly possible
- My experience?
  - Used Java SE 8 in day job since mid 2014
  - Main author of java.time.* (part of Java SE 8)

Introduction

- Software Best Practice is mostly opinion
- Different conclusions perfectly possible
- My experience?
  - Used Java SE 8 in day job since mid 2014
  - Main author of java.time.* (part of Java SE 8)

But you must exercise your own judgement!

Java SE 8 version

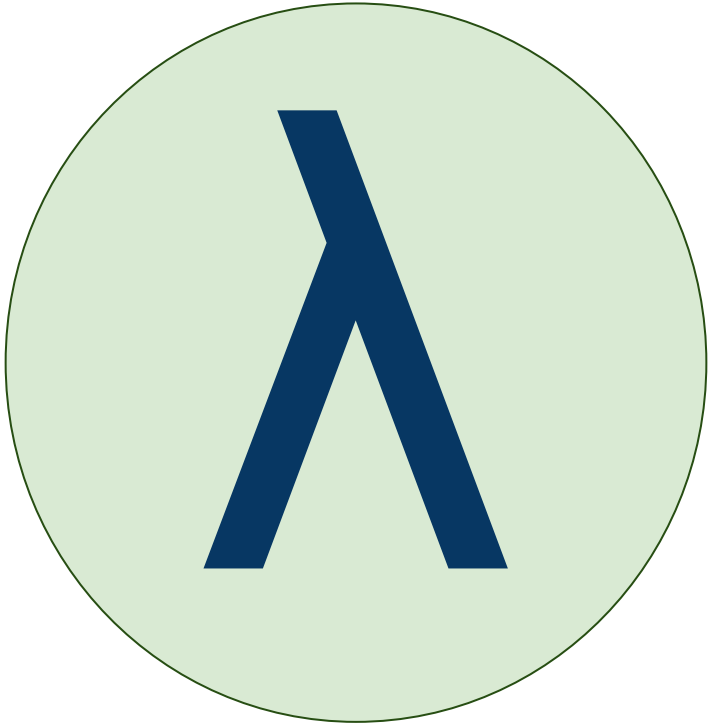
- Use Java SE 8 update 40 or later
  - preferably use the latest available
- Earlier versions have annoying lambda/javac issues

Java SE 8 version

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

This is painful on Travis CI, which still uses 8u31!

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));
```

Lambdas

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

public interface List<E> {
    void sort(Comparator<E> comparator);
}
```

Lambdas

- Make use of parameter type inference
- Only specify the types when compiler needs it

```
// prefer
(p1, p2) -> p1.name.compareTo(p2.name);

// avoid
(Person p1, Person p2) -> p1.name.compareTo(p2.name);
```

Lambdas

- Do not use parameter brackets when optional

```
// prefer
str -> str.toUpperCase(Locale.US);


// avoid
(str) -> str.toUpperCase(Locale.US);
```

Lambdas

- Do not declare local variables as 'final'
- Use new "effectively final" concept

```
public UnaryOperator<String> upperCaser(Locale locale) {
  return str -> str.toUpperCase(locale);
}
```

Do not declare as 'final'

Lambdas

- Prefer expression lambdas over block lambdas
- Use a separate method if necessary

```
// prefer
str -> str.toUpperCase(Locale.US);


// use with care
str -> {
  return str.toUpperCase(Locale.US);
}
```

Lambas for Abstraction

- Two large methods contain same code
- Except for one bit in the middle
- Can use a lambda to express the difference

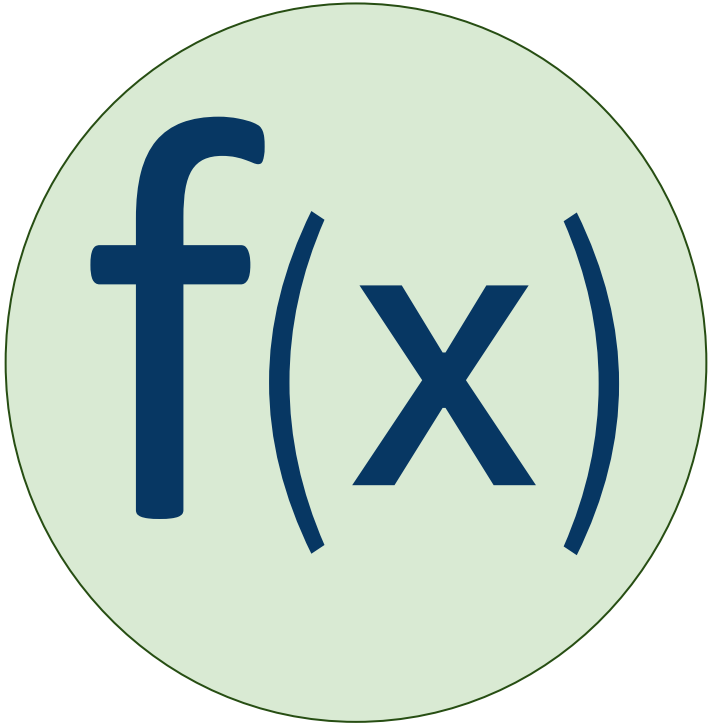
Lambas for Abstraction

```
private int doFoo() {
  // lots of code
  // logic specific to foo
  // lots of code
}
private int doBar() {
  // lots of code
  // logic specific to bar
  // lots of code
}
```

Lambas for Abstraction


```
private int doFoo() {
  return doFooBar( lambdaOfFooSpecificLogic );
}
private int doBar() {
  return doFooBar( lambdaOfBarSpecificLogic );
}
private int doFooBar(Function<A, B> fn) {
  // lots of code
  result = fn.apply(arg);
  // lots of code
}
```

Functional interfaces



Functional interfaces

- An interface with a single abstract method
  - Runnable
  - Comparable
  - Callable
- Java SE 8 adds many new functional interfaces
  - Function<T, R>
  - Predicate<T>
  - Supplier<T>
  - Consumer<T>
  - see java.util.function package

Functional interfaces

- Learn java.util.function package interface
- Only write your own if extra semantics are valuable
  - lots of params, mixture of primitive/object
- If writing one, use @FunctionalInterface

```
@FunctionalInterface
public interface FooBarQuery {
  public abstract Foo findAllFoos(Bar bar);
}
```

Higher order methods

- Methods accepting lambdas are nothing special
  - declared type is just a normal interface
- However there are some subtleties

```
private String nameGreet(Supplier<String> nameSupplier) {
  return "Hello " + nameSupplier.get();
}
// caller can use a lambda
String greeting = nameGreet(() -> "Bob");
```

Avoid method overloads

- Lambdas use target typing
- Clashes with method overloading

```
// avoid
public class Foo<T> {
  public Foo<R> apply(Function<T, R> fn);
  public Foo<T> apply(UnaryOperator<T> fn);
}
```

Avoid method overloads

- Lambdas use target typing
- Clashes with method overloading
- Use different method names to avoid clashes

```
// prefer
public class Foo<T> {
  public Foo<R> applyFunction(Function<T, R> fn);
  public Foo<T> applyOperator(UnaryOperator<T> fn);
}
```

Functional interface last

- Prefer to have functional interface last
  - when method takes mixture of FI and non-FI
- Mostly stylistic
  - slightly better IDE error recovery

```
// prefer
public Foo parse(Locale locale, Function<Locale,Foo> fn);


// avoid
public Foo parse(Function<Locale,Foo> fn, Locale locale);
```

Exceptions

Checked exceptions

- Most functional interfaces do not declare exceptions
- No simple way to put checked exceptions in lambdas

```
// does not compile!
public Function<String, Class> loader() {
  return className -> Class.forName(className);
}
```

Throws a checked exception

Checked exceptions

- Write or find a helper method
  - See 'Unchecked' from OpenGamma Strata
- Converts checked exception to unchecked

```
public Function<String, Class> loader() {
  return Unchecked.function(
                 className -> Class.forName(className));
}
```

Checked exceptions

- Helper methods can deal with any block of code
  - convert to runtime exceptions
- May be a good case for a block lambda

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

Testing for exceptions

- Complete unit tests often need to test for exceptions

```
public void testConstructorRejectsEmptyString() {
  try {
    new FooBar("");
    fail();
  } catch (IllegalArgumentException ex) {
    // expected
  }
}
```

Testing for exceptions

- Use a helper method
  - See 'TestHelper' from OpenGamma Strata
- Lots of variations on this theme are possible

```
public void testConstructorRejectsEmptyString() {
  TestHelper.assertThrows(
    IllegalArgumentException.class, () -> new FooBar(""));
}
```

Optional and null

https://www.flickr.com/photos/bigmacsc99/4003751542/

Optional and null

- New class 'Optional' added to Java 8
- Polarizes opinions
  - saviour of the universe vs utterly useless
- Used pragmatically, can be useful

Optional and null

- Simple concept - two states
  - present or empty
  - just like non-null reference vs null reference
- Must check which state it is in before querying

```
String a = "AB";
String b = null;
```

```
Optional<String> a = Optional.of("AB");
Optional<String> b = Optional.empty();
```

Optional and null

- Variable of type Optional must never be null
- Never ever
- Never, never, never, never!

```
String a = "AB";
String b = null;
```

```
Optional<String> a = Optional.of("AB");
Optional<String> b = Optional.empty();
Optional<String> c = null;  // NO NO NO
```

Optional and null

- Standard code using null

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


// application code must remember to check for null
Foo foo = findFoo(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> findFoo(String key) { … }

// application code
Foo foo = findFoo(key).orElse(Foo.DEFAULT);
// or
Foo foo = findFoo(key).orElseThrow(RuntimeException::new);
```

Optional

- Prefer "functional" methods like 'orElse()'
- using 'isPresent()' a lot is misusing the feature

```
// prefer
Foo foo = findFoo(key).orElse(Foo.DEFAULT);


// avoid ifPresent()
Optional<Foo> optFoo = findFoo(key);
if (optFoo.ifPresent()) { … }
```

Optional

- Have a discussion and choose an approach

A. Use everywhere
B. Use instead of null on public APIs, input and output
C. Use instead of null on public return types
D. Use in a few selected places
E. Do not use

Optional

- Have a discussion and choose an approach

A. Use everywhere
B. Use instead of null on public APIs, input and output
C. Use instead of null on public return types

↖ my preferred choice ↗

D. Use in a few selected places
E. Do not use

Optional

- Optional is a class
  - some memory/performance cost to using it
- Not ideal to be an instance variable
  - not serializable
  - convert null instance variable to Optional in getter
- JDK authors added it for return types
- Use in parameters typically 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

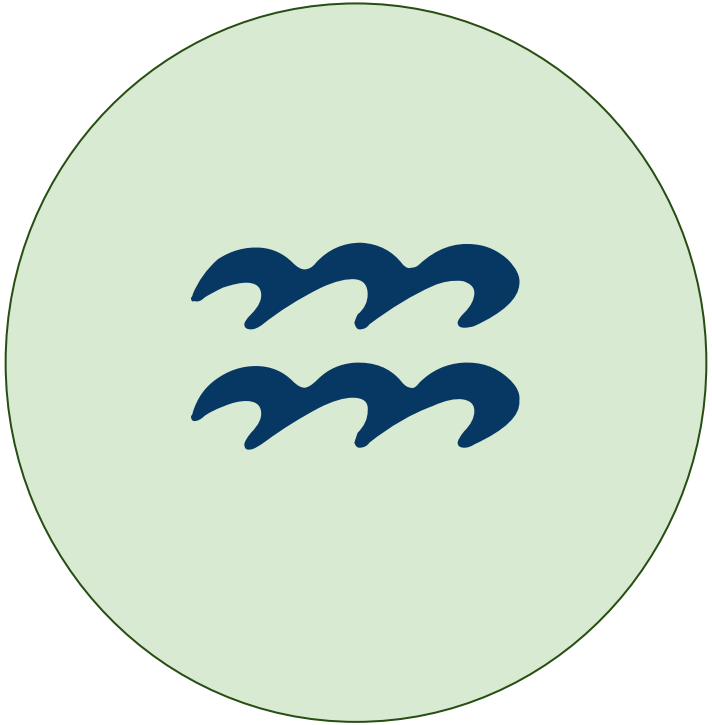
Strata

- Strata has no exposed nulls
- All methods return Optional, not null
- Within a class, null is allowed
- Instance variables either
  - non-null checked by constructor
  - nullable, with getter converting to Optional
- Nulls have ceased to be a problem

Monads

- Beware the functional programming sales people
  - Optional MUST lead to use of Either/Maybe/Try…
  - You HAVE to use Javaslang/FunctionalJava...
- Java is NOT a functional programming language
- And it never will be
- Vital to absorb the ideas, but code must still be Java

Streams



Streams

- Most loops are the same
- Repetitive *design patterns*
- Stream library provides an abstraction
- Lambdas used to pass the interesting bits

Streams

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

Streams

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

Streams

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

Streams

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

Streams

- Do not overdo it
- Stream not always more readable than loop
- Good for Collections, less so for Maps
- Streams over 'Map' best with a dedicated wrapper
  - See 'MapStream' from OpenGamma Strata

Streams

- Benchmark use in performance critical sections
- Parallel streams must be used with great care
- Shared execution pool can be deceiving

Streams

- Be cautious about overuse of method references
- IntelliJ has an unhelpful hint

```
public List<Money> value(List<Trade> trades) {
 return trades.stream()
         .filter(t -> t.isActive())
         .map(valueFn)
         .collect(Collectors.toList());
}
```

Streams

- Be cautious about overuse of method references
- IntelliJ has an unhelpful hint

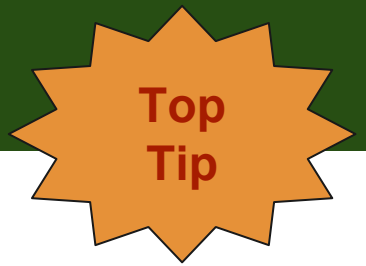
```
public List<Money> value(List<Trade> trades) {
 return trades.stream()
         .filter(Trade::isActive)
         .map(valueFn)
         .collect(Collectors.toList());
}
```

Streams

- Annoyingly common to need to convert back to lambda

```
public List<Money> value(List<Trade> trades, Data data) {
 return trades.stream()
         .filter(t -> t.isActive(data))
         .map(valueFn)
         .collect(Collectors.toList());
}
```

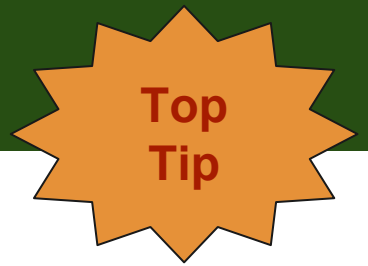
Streams



- Extract lines if struggling to get to compile

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

Streams



- Sometimes compiler needs a type hint

```
List<Trade> trades = loadTrades();



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

Streams

- Learn to love 'Collector' interface
- Complex, but useful
- Sometime necessary to write them
- Need collectors for Guava 'ImmutableList' and friends
  - see 'Guavate' class in OpenGamma Strata
  - now available in Guava v21

Streams

- Debugging streams can be painful
- Code path is non-obvious
- Large JDK call stack
- Methods that return a 'Stream' make this much worse

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

Streams

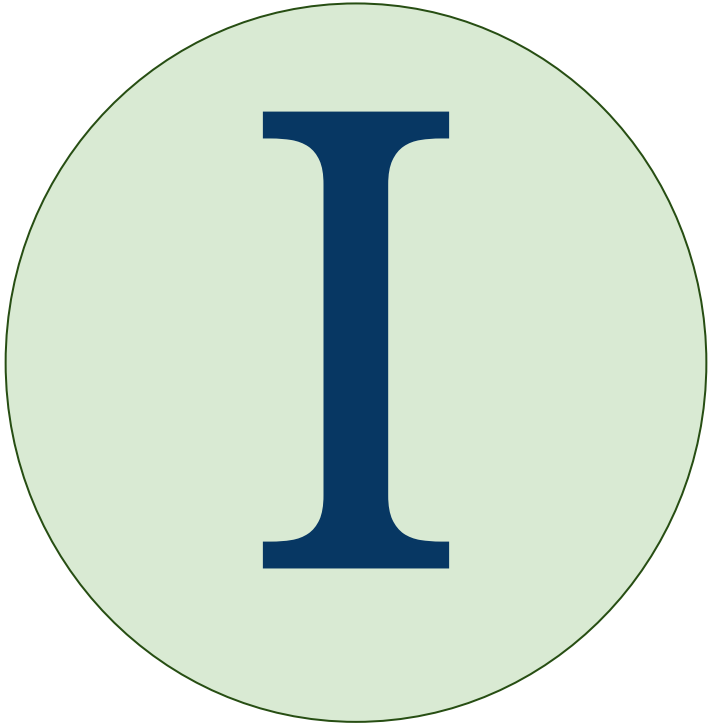
```
java.lang.IllegalArgumentException: Oops
    at com.opengamma.strata.calc.DefaultCalculationRunner.calculate(DefaultCalculationRunner.java:102)
    at com.opengamma.strata.calc.DefaultCalculationRunner.calculate(DefaultCalculationRunner.java:87)
    at com.opengamma.strata.calc.DefaultCalculationRunnerTest.calculate(DefaultCalculationRunnerTest.java:49)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

Stack trace of for-each loop

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
  - don't return streams from methods (at least not initially)
- Strata uses for-each loops at top level
  - solely for shorter stack traces

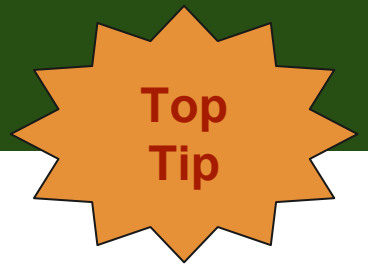
Interfaces



Interfaces

- Now have super-powers
- Default methods
  - normal method, but on an interface
- Static methods
  - normal static method, but on an interface
- Extend interfaces without breaking compatibility
- Cannot default equals/hashCode/toString

Interfaces



- New macro-design options
- Instead of factory class, use static method on interface
- Instead of abstract class, use interface with defaults
- Result tends to be fewer classes and better API
  - See 'RollConvention', and many others, from OpenGamma Strata

Interfaces

- If factory method is static on interface
- And all API methods are on interface
- Can implementation class be package scoped?

Coding Style

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

```
public interface Foo {
  public static of(String key) { … }
  public abstract getKey();
  public default isActive() { … }
}
```

Date and Time

Date and Time

- New Date and Time API - JSR 310
- Covers dates, times, instants, periods, durations
- Brings 80%+ of Joda-Time to the JDK
- Fixes the mistakes in Joda-Time

Date and Time

Class	Date	Time	ZoneOffset	ZoneId	Example
LocalDate	✔	×	×	×	2015-12-03
LocalTime	×	✔	×	×	11:30
LocalDateTime	✔	✔	×	×	2015-12-03T11:30
OffsetDateTime	✔	✔	✔	×	2015-12-03T11:30+01:00
ZonedDateTime	✔	✔	✔	✔	2015-12-03T11:30+01:00 [Europe/London]
Instant	×	×	×	×	123456789 nanos from 1970-01-01T00:00Z

Date and Time

- Move away from Joda-Time
- Avoid java.util.Date and java.util.Calendar
- Use ThreeTen-Extra project if necessary
  - http://www.threeten.org/threeten-extra/
- Focus on four most useful types
  - LocalDate, LocalTime, ZonedDateTime, Instant
- Network formats like XML/JSON use offset types
  - OffsetTime, OffsetDateTime

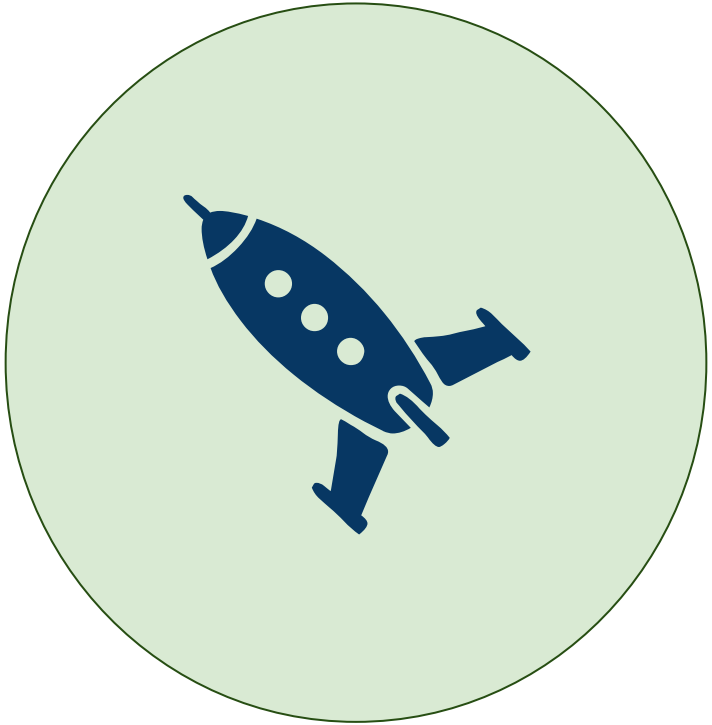
Date and Time

- Temporal interfaces are low-level
- Use concrete types

```
// prefer
LocalDate date = LocalDate.of(2015, 10, 15);


// avoid
Temporal date = LocalDate.of(2015, 10, 15);
```

Rocket powered



Other features

- Base64
- Arithmetic without numeric overflow
- Unsigned arithmetic
- StampedLock
- CompletableFuture
- LongAdder/LongAccumulator
- Enhanced control of OS processes

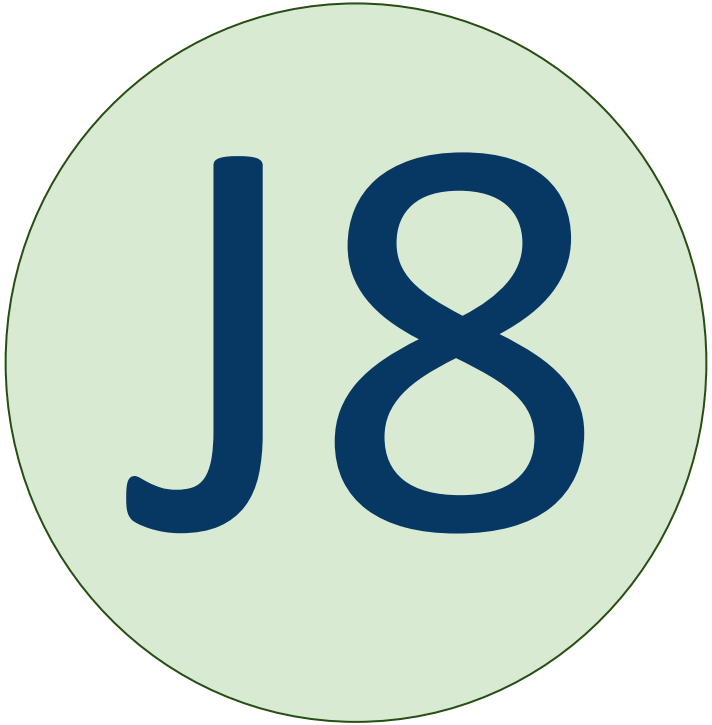
Other Features

- Enhanced annotations
- Reflection on method parameters
- No PermGen in Hotspot JVM
- Nashorn JavaScript
- JavaFX is finally ready to replace Swing

Immutability

- Favour immutable classes
- Lambdas and streams prefer this
- Preparation for *value types*
- Use Joda-Beans to generate immutable "beans"
  - http://www.joda.org/joda-beans/

Summary



Summary

- Java 8 is great
- Can be quite different to Java 7 and earlier
- Vital to rethink coding style and standards
  - methods on interfaces make a big difference
- Be cautious about functional programming
  - Java is not an FP language
  - we need to take the knowledge and apply it to Java
  - FP libraries in Java typically not what you want

Summary

- OpenGamma Strata is a good exemplar project
  - 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, v1.2 coming soon

http://strata.opengamma.io/

Thanks!

- Questions and Feedback
  - Java 8 best practices
  - JSR-310 / java.time.*
  - Strata
- Twitter - @jodastephen