Java SE 8 Best Practices

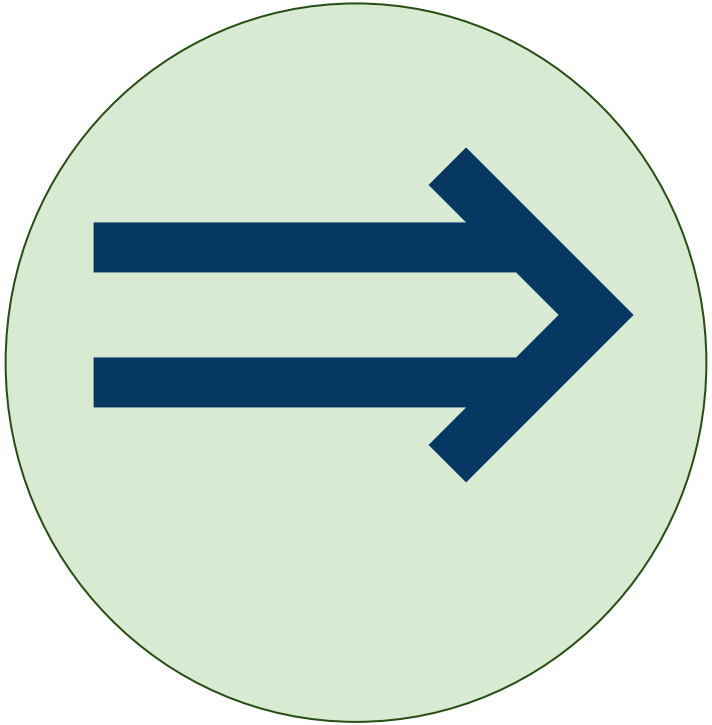
A personal viewpoint

Stephen Colebourne, October 2015

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 opinions are based on over a year using Java SE 8

Introduction

- Software Best Practice is mostly opinion
- Different conclusions perfectly possible
- My opinions are based on over a year using 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

Lambdas

Lambdas

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

Lambdas - Example

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

Lambdas - Example

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

Lambdas - Example

```
// Java 8
List<Person> people = loadPeople();
Collections.sort(people,

                        (Person p1, Person p2)
            p1.name.compareTo(p2.name)

 );
```

Lambdas - Example

```
// Java 8
List<Person> people = loadPeople();
Collections.sort(people,
  (Person p1, Person p2) -> p1.name.compareTo(p2.name));
```

Lambdas - Example

```
// Java 8
List<Person> people = loadPeople();
Collections.sort(people,
  (Person p1, Person p2) -> p1.name.compareTo(p2.name));
```

Lambdas - Example

```
// Java 8
List<Person> people = loadPeople();
Collections.sort(people,
                 (p1, p2) -> p1.name.compareTo(p2.name));
```

Lambdas - Example

```
// Java 8
List<Person> people = loadPeople();
people.sort((p1, p2) -> p1.name.compareTo(p2.name));
```

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 method1
  // lots of code
}
private int doBar() {
  // lots of code
  // logic specific to method2
  // lots of code
}
```

Lambas for Abstraction


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

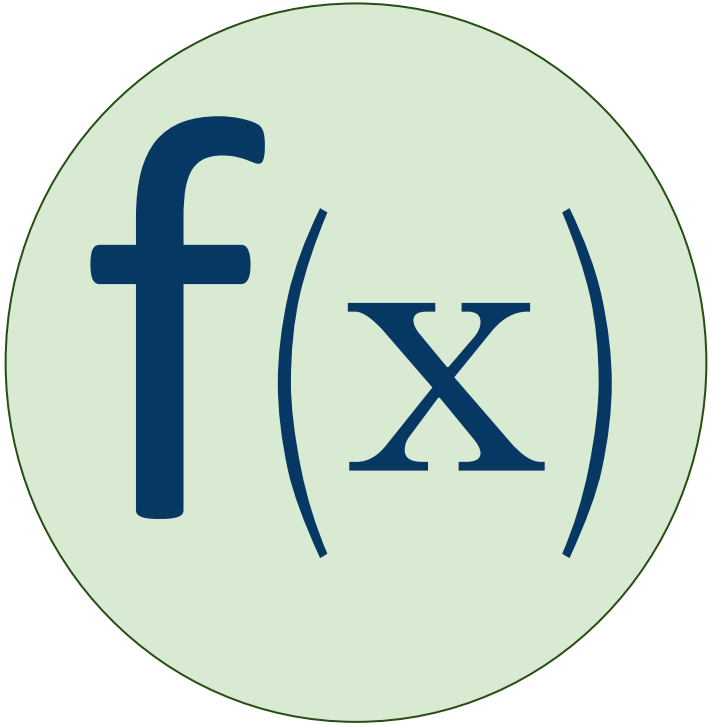
Example Abstraction

```
double[][] res = new double[rowCount][colCount];
for (int i = 0; i < rowCount; ++i) {
  for (int j = 0; j < colCount; ++j) {
    res[i][j] = pp.getCoefMatrix().get(i, j) * (nCoefs - j - 1);
  }
}
DoubleMatrix2D coef = new DoubleMatrix2D(res);
```

Example Abstraction

```
DoubleMatrix2D coef = DoubleMatrix2D.of(
    rowCount,
    colCount,
    (i, j) -> pp.getCoefMatrix().get(i, j) * (nCoefs - j - 1));

// new method
 public static DoubleMatrix2D of(
    int rows, int columns, IntIntToDoubleFunction valueFunction)
```

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
- 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
- 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
- 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
  - functional programming dudes think it is the saviour of the universe
- 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 findFoo(String key) { … }


// application code
Foo foo = findFoo(key);
if (foo == null) {
  foo = Foo.DEFAULT;  // or throw an exception
}
```

Optional and null

- Standard code using Optional

```
// library, returns null 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 and null

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

Optional

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

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


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

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 ↗

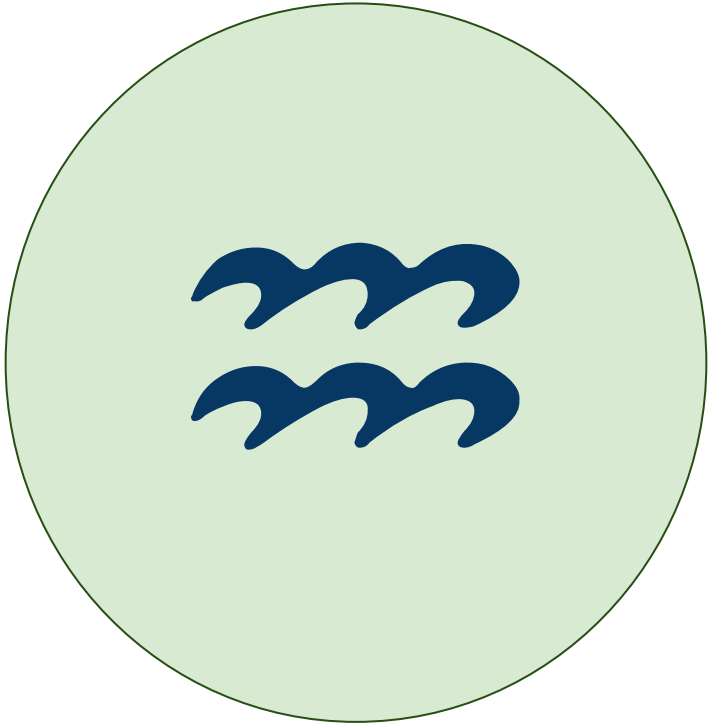
D. Use in a few selected places
E. Do not use

Optional

- Optional is a class
- Some memory/performance cost to using it
- Not 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

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);
  }
}
```

Loop to build output list from input

Only interested in some trades

Converts each trade to the money value

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 =          List
                  trades
      t.isActive()
                  presentValue(t)
```

Streams

```
List<Trade> trades = loadTrades();
List<Money> valued =                          // List
                                              // trades
                                              // t.isActive()
                                              // presentValue(t)
```

Streams

```
List<Trade> trades = loadTrades();
List<Money> valued =                          // List
   trades.stream()                            // trades
                                              // t.isActive()
                                              // presentValue(t)
```

Streams

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

Streams

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

Streams

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

Streams

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

Streams

- Streams are great, sometimes
- Important not to get carried away
- Design focus was on Collections, not Maps
- Key goal was simple parallelism

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());
```

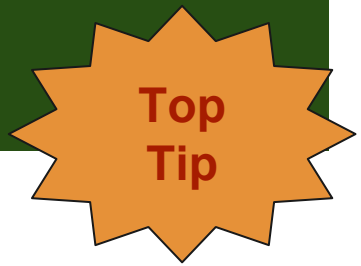
Streams

- Do not overdo it
- Stream not always more readable than loop
- Good for Collections, less so for Maps
- Don't obsess about method references
  - IntelliJ hint may not be the best idea

Streams

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

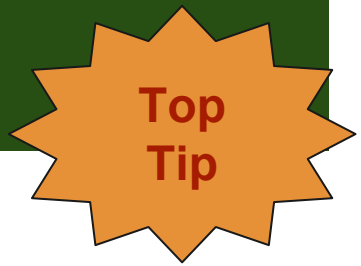
Streams



- 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



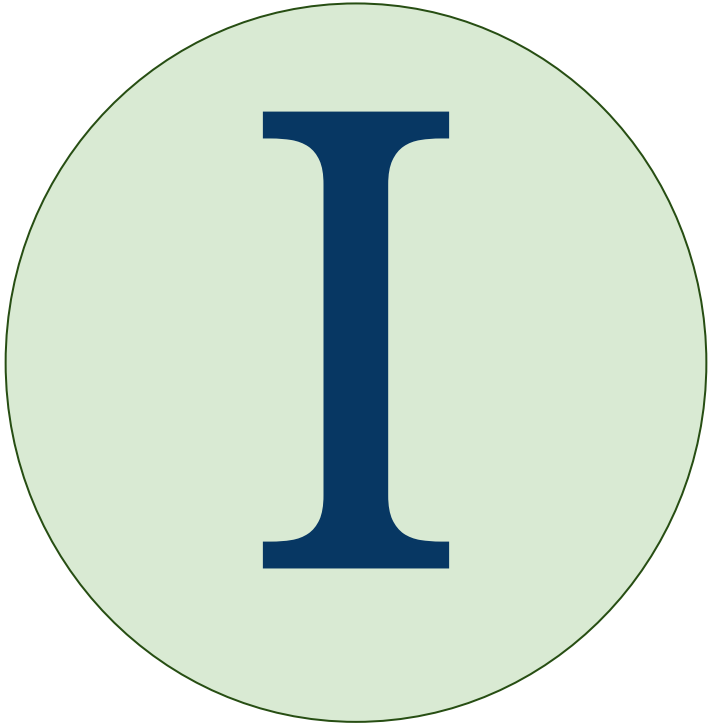
- Sometimes compiler needs a type hint

```
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

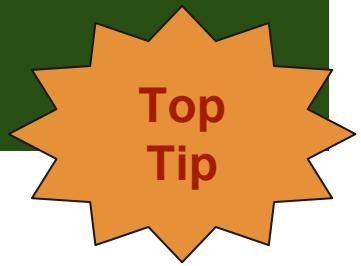
Interfaces



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

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 possible future with non-public methods

```
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

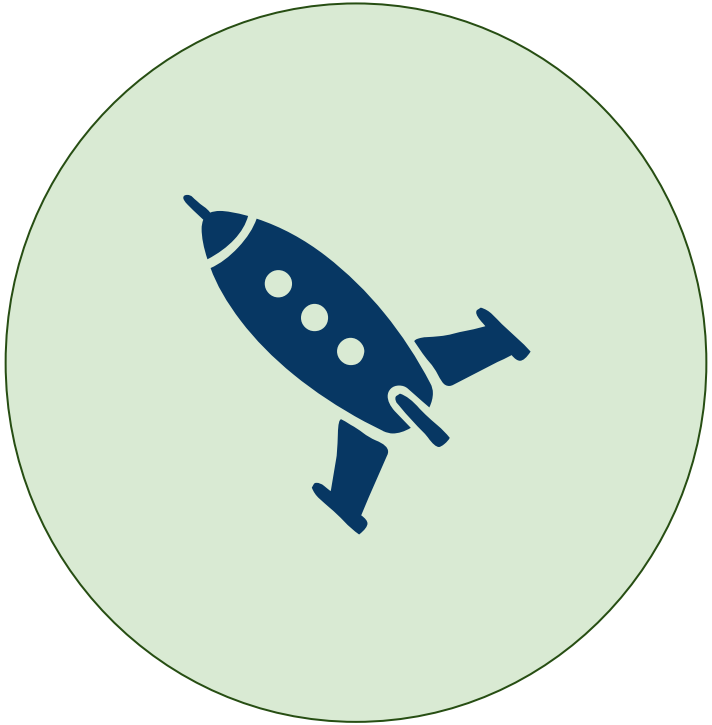
- Move away from Joda-Time
- Avoid java.util.Date and java.util.Calendar
- Use ThreeTen-Extra project if necessary
  - http://www.threeten.org/threeten-extra/
- Focus on four most useful types
  - LocalDate, LocalTime, ZonedDateTime, Instant
- Network formats like XML/JSON use offset types
  - OffsetTime,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

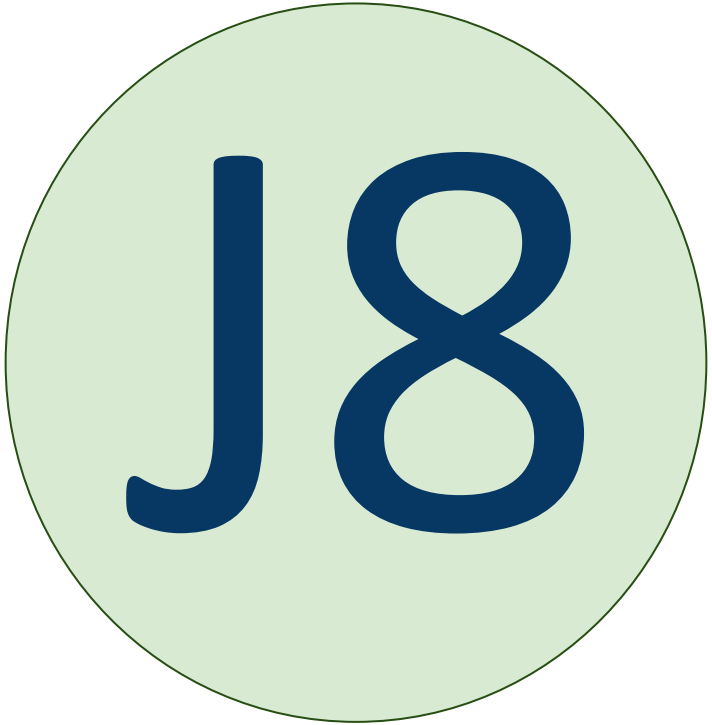
Try a Java 8 open source library

- JOOL
  - https://github.com/jOOQ/jOOL
- ThrowingLambdas
  - https://github.com/fge/throwing-lambdas
- Parts of OpenGamma Strata (strata-collect - Guavate)
  - https://github.com/OpenGamma/Strata
- But beware excessively functional ones
  - most push ideas that don't really work well in Java

Immutability

- Favour immutable classes
- Lambdas and streams prefer this
- Preparation for *value types* (Java 10?)
- 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
- Beware the functional programming/Scala dudes
  - a lot of their advice is simply not appropriate for Java