Java 8 Upgrade Class

Ken Kousen
ken.kousen@kousenit.com
http://www.kousenit.com
http://kousenit.org (blog)
@kenkousen

This document: http://bit.ly/2los3Lp

Course materials: http://www.kousenit.com/java8
Source code (labs): https://github.com/kousen/java8_upgrade
Source code (examples): https://github.com/kousen/java_8_recipes

For IntelliJ:

- clone repo or download zip and extract
- Use File -> Open or if not project is open, Import
- Navigate to the ***build.gradle*** file inside the project
- Click enter and accept all the defaults

For Eclipse (and Eclipse-based tools, like STS):

- Open a command prompt in the root of the unzipped project
- Type "gradlew cleanEclipse eclipse" (Note: if you're on a Unix-based system, including Macs, you need to use "./gradlew …")
- Wait for the dependencies to be downloaded
- Choose File -> Import… -> General -> Existing Projects into Workspace
- Navigate to the root of the project and select it

For any book at Manning, http://manning.com (Java 8 in Action, all others)
kousen37 → 37% discount

For any book at O'Reilly, http://shop.oreilly.com

AUTHD → 40% off books and 50% off videos

Default methods in interfaces:

- use the keyword `default`
- provide an implementation
- if you inherit a conflict, implement the method in a class

Benefit: you don't need to create utility classes (like Collections)

Whenever there is a conflict between a class method and a default method, class always wins

Static methods in interfaces:

- use the keyword `static`
- provide an implementation
- call the method *from the interface name*
- you do not need to implement the interface to use the static methods
- you can't override a static method

Abstract classes can have fields (attributes, data, …)
Interfaces can't.

In a Stream, any method that returns a stream is an intermediate operation
Any method that returns other than a stream is a terminal operation

```
widgets.stream()                          // Stream<Widget>
    .filter(w -> w.getColor() == RED)  // Stream<Widget>, but only RED
    .mapToInt(Widget::getWeight)       // IntStream
    .sum()
```

filter and mapToInt are **intermediate** operations
sum is a **terminal** operation

→ No elements are processed until you have a terminal operation
→ Each element passes through the pipeline one at a time

Day 1 exercises available:
StringExercises.java
FunctionInterfacesTest.java
both under src/test/java/...

In Java 9, interfaces can also have private methods

The method parallel() tells the stream to form a (common) ForkJoinPool for the processing
The method sequential() processes in order with a single thread
But:

- either of them just turns a parallel flag on or off
- nothing happens until you hit the terminal expression
- then flag is checked and processing begins

```
Stream.of(.....)
      .parallel()  // turns on parallel flag
      .map(...)
      .sequential()
      .sorted()  // turns it back off
      .collect(...)  // here, the flag is off, so → sequential
```

Find time to discuss findFirst() vs findAny() in parallel vs sequential
`The original collection has an encounter order, so findFirst() always returns the same value even in parallel`

Local variables (variables declared inside a method) must be *final* or *effectively final* if they are used inside a lambda expression.

Brian Goetz → author of "Java Concurrency in Practice"
"Concurrency is an optimization"

```
 long startDate = System.currentTimeMillis();
 System.out.println(new Date(startDate));
 long count = Stream.generate(() -> new Random().nextInt(10))
     .limit(Integer.MAX_VALUE)
     .parallel()
     .filter(x -> x%2 == 0)
     .count();

long endDate = System.currentTimeMillis();
System.out.println(new Date(endDate));
System.out.println("generated " + count);
System.out.println(endDate - startDate);

IntStream.rangeClosed(1, Integer.MAX_VALUE)
      .filter(x -> x % 2 == 0)
      .count();
```

Concurrency efficiency is based on N * Q

N → the number of elements being processed

Q → some measure of how long it takes to process each element

NQ > 10,000 is a minimum in order for parallelization to be beneficial

In order for a fork-join pool to be effective

- The data to be easily divided into chunks
- Works best with contiguous data

A Stream with a limit is not easy for the system to divide up
Best is if you have a well-defined collection (like a list or an array)

especially if it backed by an array in memory

Data suggests an IntStream can be as much as 100X faster than a Stream<Integer>

Keynote address by Rich Hickey (creator of the Clojure language)
"Simple Made Easy"
https://www.infoq.com/presentations/Simple-Made-Easy
simple → conceptually trivial
easy → can call it without trouble (like an API call)
may be hiding massive complexity under the hood

On our streams, the call to "parallel()" is easy, but not necessarily simple
Keep in mind that sequential processing of primitive streams is very fast

Optional in a DAO layer
Product class, PRODUCTS table in DB
name
description
price
All of those attributes naturally map to columns in the table
description may be nullable

1. Do not use Optional<..> on an attribute/field. Optional is deliberately made non-serializable
2. Do not need Optional on properties that are NOT NULL
3. For nullable values, use Optional on the getter, but not setter

```
class Product implements Serializable {
      private String name;
      private String description;
      private BigDecimal price;

      public String getName() { return name; }
      public void setName(String name) { this.name = name; }

      public Optional<String> getDescription() {
            return Optional.ofNullable(description);
      }
      public void setDescription(String desc) { description = desc; }
…
}
```

Most developers use Log4J or SLF4J or logback, …
The Java API includes Java Commons Logging
    java.util.logging.Logger

Any arguments to Java methods are evaluated before the body of the method is executed
    log.info(message) → message is evaluated whether we need it or not

In a supplier, we can control when the get() method is invoked
    log.info(() → message) → message is only evaluated if we are logged

Deferred execution → only execute when necessary
    done by putting our evaluation inside a Supplier

Optional means we are returning a value that may legitimately be null
    Never return a null if the return type is Optional!
    Returning an empty Optional is fine

The goal of Optional is NOT to remove all NPE from your code

In other words, don't replace every reference with an Optional

Instead of getMiddleName() returning String, you can return Optional<String>

How do you get a value out of an Optional?

- There is a get() method
- Never call get() unless you're sure the Optional contains a value
  - Otherwise get() throws an exception

Does that mean instead of checking for null, we have to call isPresent()?

That would work, but not necessary.
Can call orElse() instead