Functional Java

Streams, lambdas, method references and more…

Contact Info

Ken Kousen

Kousen IT, Inc.

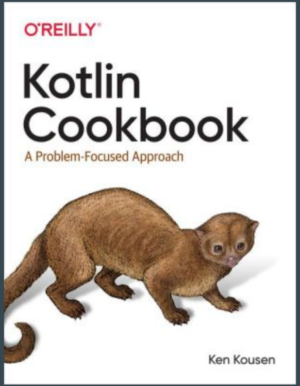
ken.kousen@kousenit.com

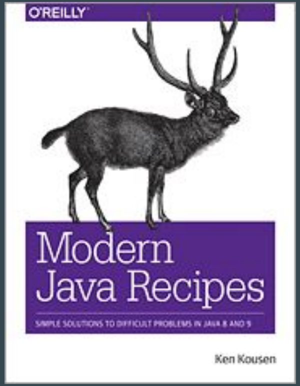
http://www.kousenit.com

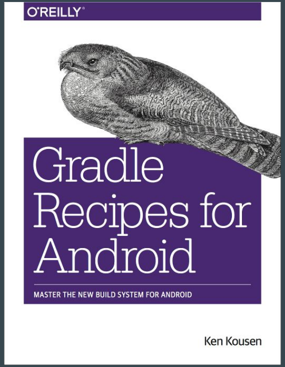
http://kousenit.org (blog)

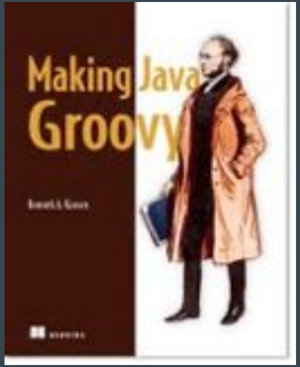
@kenkousen (twitter)

https://kenkousen.substack.com (newsletter)






O'REILLY
Gradle Recipes for Android
MASTER THE NEW BUILD SYSTEM FOR ANDROID
Ken Kousen



Videos (available on Safari)

O'Reilly video courses: See http://shop.oreilly.com for details

- Groovy Programming Fundamentals
- Practical Groovy Programming
- Mastering Groovy Programming
- Learning Android
- Practical Android
- Gradle Fundamentals
- Gradle for Android
- Spring Framework Essentials
- Advanced Java Development

Modern Java Recipes

Materials and examples are from the upcoming book

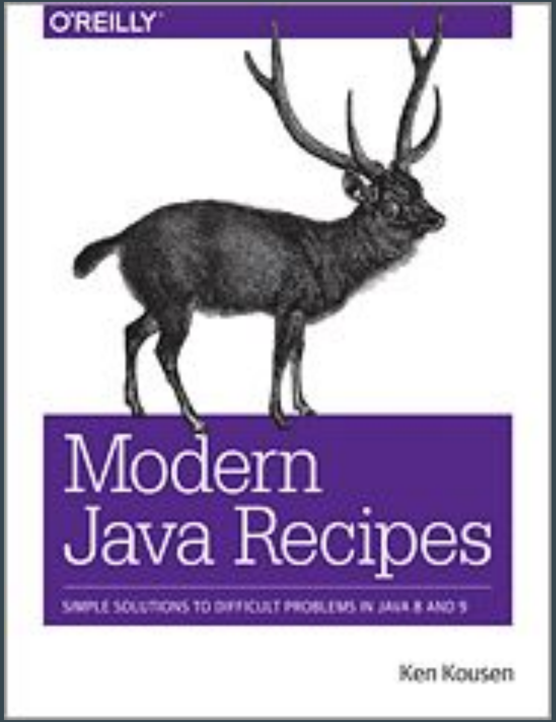
Source code:

https://github.com/kousen/java_upgrade

https://github.com/kousen/java_8_recipes

Materials:

http://www.kousenit.com/java8/



The Basics

- Streams
- Lambda Expressions
- Method References

Lambda Expressions

- Java lambda expressions
  - Assigned to Single Abstract Method interfaces
  - Parameter types inferred from context

Functional Interface

Interface with a Single Abstract Method

- `Runnable`

Lambdas can only be assigned to

- functional interfaces

Functional Interface

See `java.util.function` package

`@FunctionalInterface`

Not required, but useful

Functional Interfaces

Consumer → single arg, no result

```
void accept(T t)
```

Predicate → returns boolean

```
boolean test(T t)
```

Supplier → no arg, returns single result

```
T get()
```

Function → single arg, returns result

```
R apply(T t)
```

Functional Interfaces

Primitive variations

`Consumer`

```
IntConsumer, LongConsumer,

DoubleConsumer,

BiConsumer<T,U>
```

Functional Interfaces

BiFunction → binary function from T and U to R

```
    R apply(T, U)
```

UnaryOperator extends Function (T and R same type)

BinaryOperator extends BiFunction (T, U, and R same type)

Exceptions

Only checked exceptions declared

in the abstract method can be thrown

Either

Catch others in body of lambda

Define wrapper method that handles exceptions

Method References

Method references use :: notation

```
System.out::println
    x → System.out.println(x)
Math::max
    (x,y) → Math.max(x,y)
String::length
    x → x.length()
String::compareToIgnoreCase
    (x,y) → x.compareToIgnoreCase(y)
```

Constructor References

Can call constructors

```
ArrayList::new

Person[]::new
```

Default methods

- Default methods in interfaces
  - Use keyword `default`

Default methods

What if there is a conflict?

Class vs Interface → Class always wins

Interface vs Interface →

Child overrides parent

Otherwise compiler error

Static methods in interfaces

- Can add static methods to interfaces
  - See `Comparator.comparing`

Streams

- A sequence of elements
  - Does not store the elements
  - Does not change the source
  - Operations are lazy when possible
  - Closed when terminal expression reached

Streams

A stream carries values

from a source

through a pipeline

Pipelines

- Okay, so what's a pipeline?
  - A source
  - Zero or more **intermediate** operations
  - A **terminal** operation

Reduction Operations

- Reduction operations
  - Terminal operations that produce
    - one value from a stream

```
average, sum, max, min, count, ...
```

Streams

- Easy to parallelize
  - Replace `stream()` with
    - `parallelStream()`

Creating Streams

Creating streams

```
Collection.stream()

Stream.of(T... values)

Stream.generate(Supplier<T> s)

Stream.iterate(T seed, UnaryOperator<T> f)

Stream.empty()
```

Transforming Streams

Process data from one stream into another

```
filter(Predicate<T> p)

map(Function<T,R> mapper)
```

Transforming Streams

There's also flatMap:

```
Stream<R> flatMap(Function<T, Stream<R>> mapper)
```

Map from single element to multiple elements

Remove internal structure

Substreams

`limit(n)` returns a new stream

ends after n elements

```
DoubleStream.generate(Math::random)
      .limit(100)
```

Collectors

`Collector` interface

"Mutable reduction operation that accumulates elements into a mutable result container, optionally transforming the accumulated result after all input elements have been processed"

`Collectors` class

Convenient methods for converting into lists, sets, maps

Using Collectors

```
Stream.of( … )

    .collect( Collectors.toList() )
```
→ creates an ArrayList

```
    .collect( Collectors.toSet() )
```
→ creates a HashSet

```
    .collect( Collectors.toCollection( Supplier ))
```
→ creates the supplier (LinkedList::new, TreeSet::new, etc)

```
    .collect( Collectors.toMap( Function, Function ))
```
→ creates a map; first function is keys, second is values

Optional

Alternative to returning object or null

`Optional<T>` value

- `isPresent()` → boolean
- `get()` → return the value

Goal is to return a default if value is null

Optional

`ifPresent()` accepts a consumer

`optional.ifPresent(` ... do something ...)

`orElse()` provides an alternative

`optional.orElse(`... default ...)

```
optional.orElseGet(Supplier<? extends T> other)

optional.orElseThrow(Supplier<? extends X> exSupplier)
```

Deferred execution

Logging

```
log.info("x = " + x + ", y = " + y);
```

String formed even if not info level

```
log.info(() -> "x = " + x + ", y = " + y);
```

Only runs if at info level

Arg is a `Supplier<String>`

Date and Time API

`java.util.Date` is a disaster

`java.util.Calendar` isn't much better

Now we have `java.time`

LocalDate

A date without time zone info

contains year, month, day of month

```
LocalDate.of(2017, Month.FEBRUARY, 2)
```

months actually count from 1 now

LocalTime

`LocalTime` is just LocalDate for times

hh:mm:ss

`LocalDateTime` is both, but then you

might need time zones

ZonedDateTime

Database of timezones from IANA

https://www.iana.org/time-zones

```
Set<String> ZoneId.getAvailableZoneIds()

ZoneId.of("... tz name ...")
```

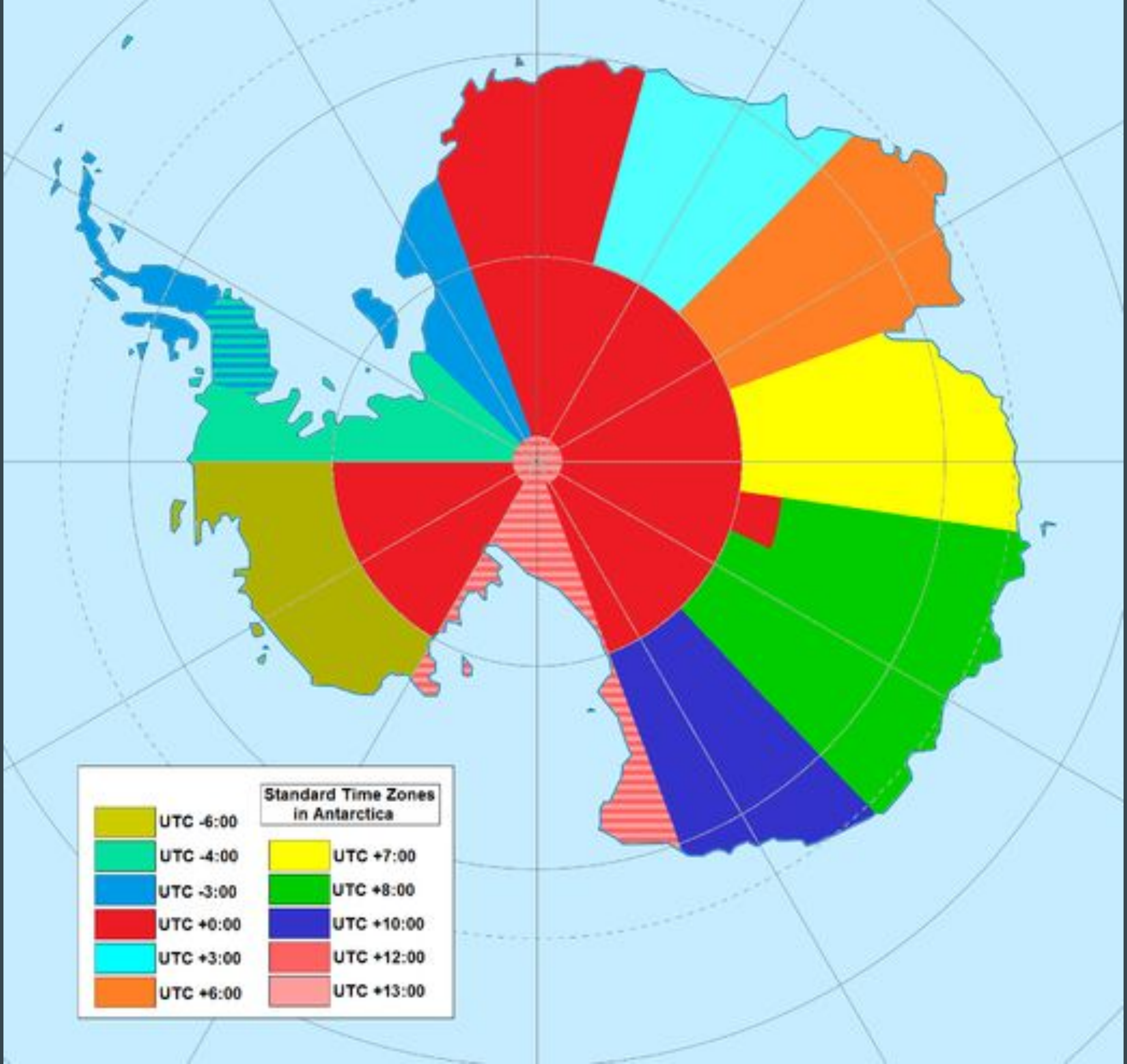
ZonedDateTime

`LocalDateTime` → `ZonedDateTime`

```
local.atZone(zoneId)
```

`Instant` → `ZonedDateTime`

```
instant.atZone(ZoneId.of("UTC"))
```


Standard Time Zones in Antarctica
UTC -6:00
UTC -4:00
UTC -3:00
UTC +0:00
UTC +3:00
UTC +6:00
UTC +7:00
UTC +8:00
UTC +10:00
UTC +12:00
UTC +13:00

Dates and Times

Java 8 Date-Time: java.time package

AntarcticaTimeZones.java

Summary

- Functional programming
  - Streams with map / filter / reduce
  - Lambda expressions
  - Method references
  - Concurrent, parallel streams
- Optional type
- Collectors and Comparators
  - Conversion from stream back to collections
  - Enable sorting, partitioning, and grouping
- Date/Time API
  - Good reason to upgrade