Functional Modern 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)

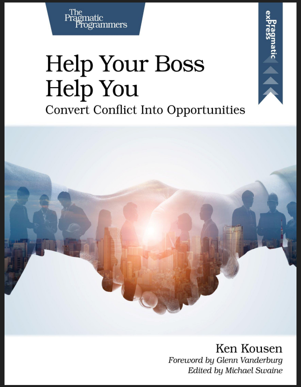
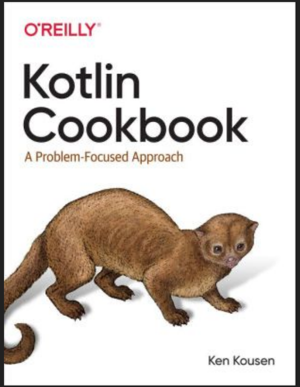
@kenkousen (twitter)

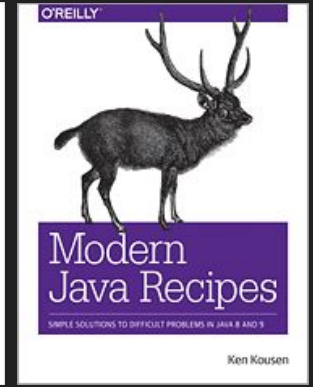
@kenkousen@mastodon.social (mastodon)

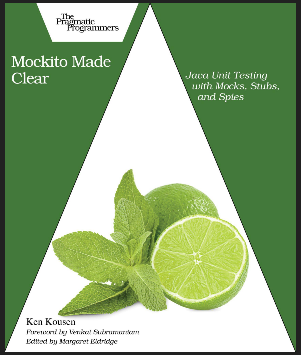
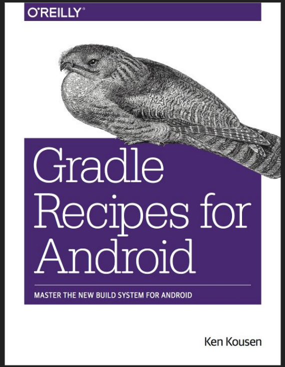
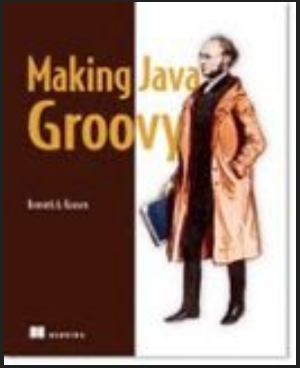
Tales from the jar side (free newsletter)

https://kenkousen.substack.com

https://youtube.com/@talesfromthejarside


The Pragmatic Programmers
Pragmatic exPress
Help Your Boss Help You
Convert Conflict Into Opportunities
Ken Kousen
Foreword by Glenn Vanderburg
Edited by Michael Swaine





The Pragmatic Programmers
Mockito Made Clear
Java Unit Testing with Mocks, Stubs, and Spies
Ken Kousen
Foreword by Venkat Subramaniam
Edited by Margaret Eldridge

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


Modern Java Recipes

Materials and examples are from the upcoming book

Source code:

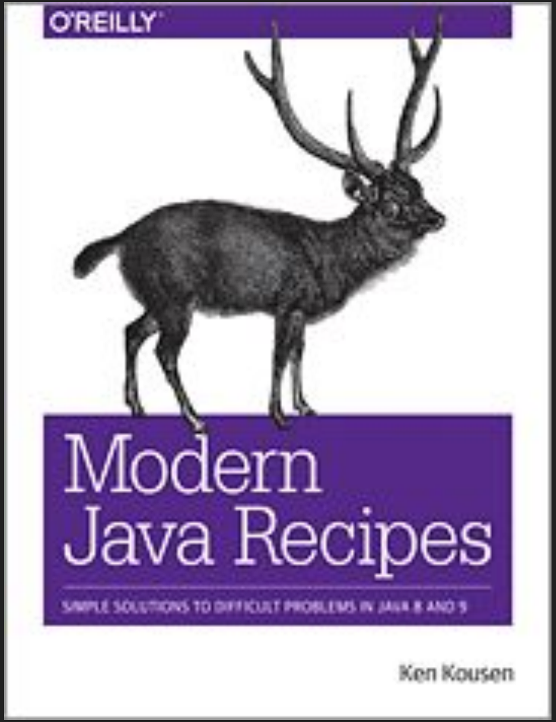
https://github.com/kousen/java_upgrade

https://github.com/kousen/java_8_recipes

https://github.com/kousen/java_latest

Materials:

http://www.kousenit.com/java/



Java Functional Features (JDK 8)

Streams, lambdas, method references

Lambda Expressions

Java lambda expressions

- Assigned to functional interfaces
- Parameter types inferred from context

```
Predicate<String> evenFilter = s → s.length() % 2 == 0
```

Predicate: functional interface with generic type

Lambda: RHS expression

Functional Interface

Interface with a Single Abstract Method

Lambdas (and method references) can ONLY be assigned
to functional interfaces

Functional Interfaces in the JDK

- See `java.util.function` package
- `@FunctionalInterface`
    - Not required, but used in library

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)

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
```

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

Streams

- Intermediate operations
    - Methods on Stream that return a Stream
    - `map, filter, flatMap, sorted, distinct, limit, peek`
- Terminal operations
    - Methods on Stream that return anything else
    - `count, allMatch, anyMatch, findFirst, forEach, min, max, reduce`

Reduction Operations

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

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

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

`Stream<T> filter(Predicate<T> predicate)`

Return only elements satisfying the predicate

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

Convert a `Stream<T>` into a `Stream<R>`

Transforming Streams

There's also flatMap:

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

Maps from single element of type T
to *wrapped* element of type `Stream<R>`

Removes internal wrapping

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

Substreams

`limit(n)` returns a new stream

ends after n elements

What does this code do (Note: Trick question)?

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

Static And Default Methods in Interfaces (JDK 8)

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
- Do not need to implement the interface to use it
- Access static methods from the interface name
  - See `Comparator.comparing`

Optional Type (JDK 8)

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(2023, 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 …")
```

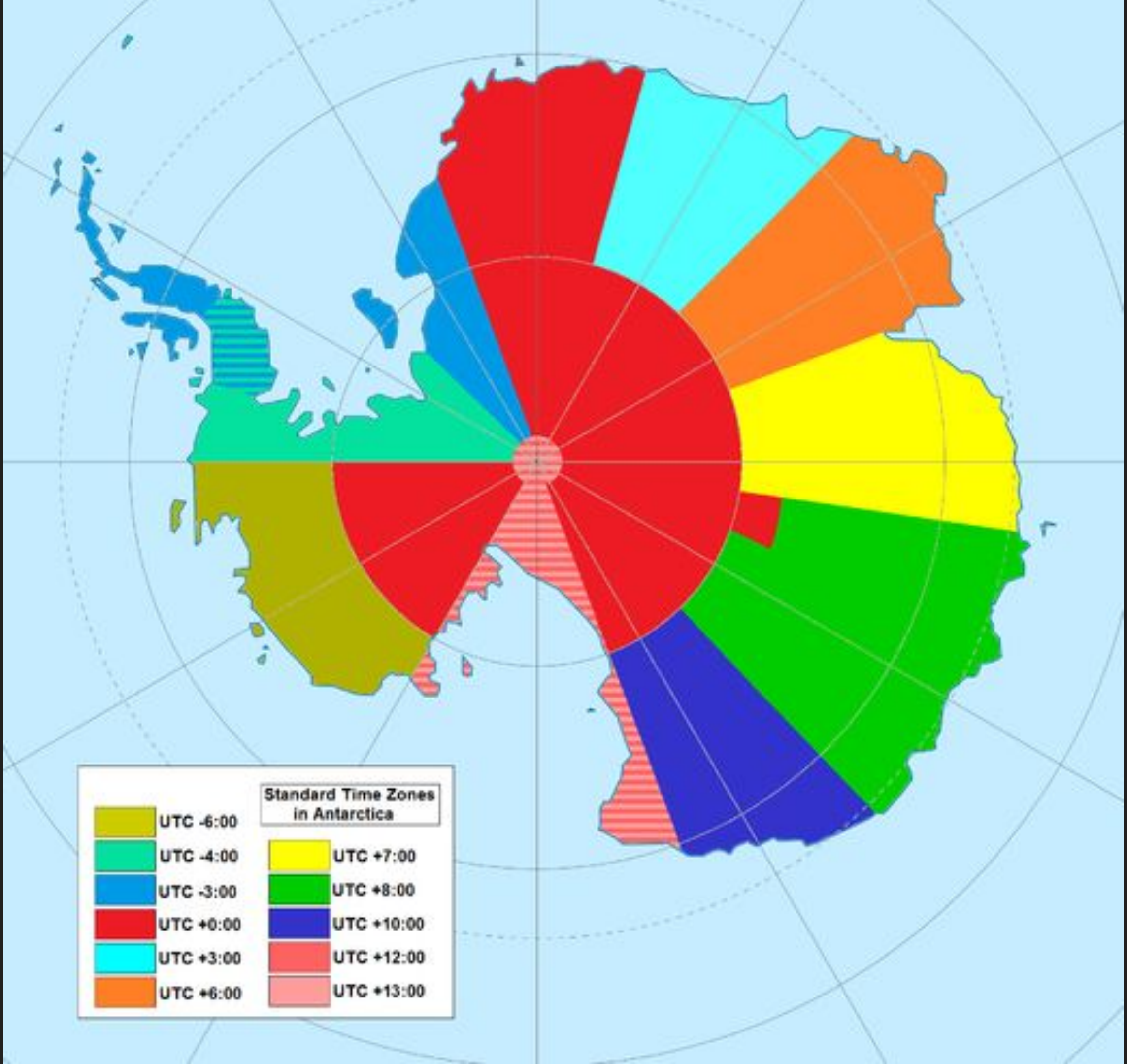
ZonedDateTime

`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