What's New In Java

Streams, lambdas, method references, LVTI, JPMS, …

Contact Info

Ken Kousen

Kousen IT, Inc.

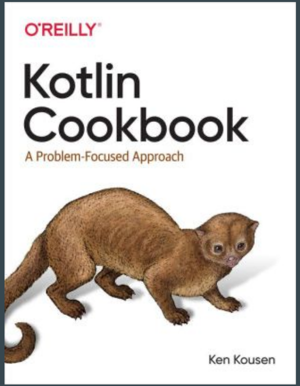
ken.kousen@kousenit.com

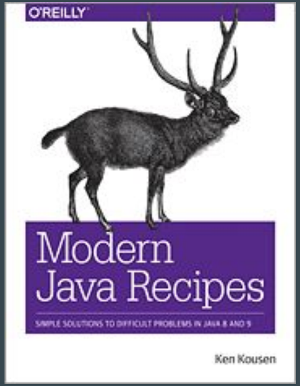
http://www.kousenit.com

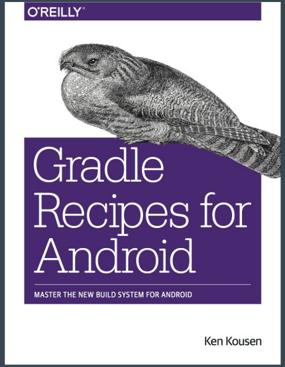
http://kousenit.org (blog)

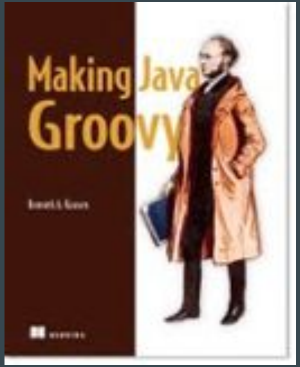
@kenkousen (twitter)

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






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



Videos (available on the O'Reilly Learning Platform)

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

GitHub Repository

Java Latest

https://github.com/kousen/java_latest

Documentation pages

https://docs.oracle.com/en/java/javase/11/

- Tools Reference
- JShell User Guide
- Javadoc Guide

Note: Actual API Javadocs are at:

https://docs.oracle.com/en/java/javase/11/docs/api/index.html

Java Licensing Is a Mess, But...

Java is Still Free 2.0.3 - Java Champions

Java 8

- End of life without commercial support (ended Jan 2019)
- Open JDK (and others) still provide updates

Java 11

- Oracle JDK requires license for production use
- Open JDK (and others) are free

Features You Need To Know

Java Functional Features

Streams, lambdas, 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

Lambdas can only be assigned to

functional interfaces

Functional Interface

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)
```

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, ...
```

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

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
```

Static And Default Methods in Interfaces

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`

Optional Type

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)`

The java.time Package

LocalDate, LocalTime, ZonedDateTime, and more

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

Date and Time API

`java.util.Date` is a disaster

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

Now we have `java.time`

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 ...")
```

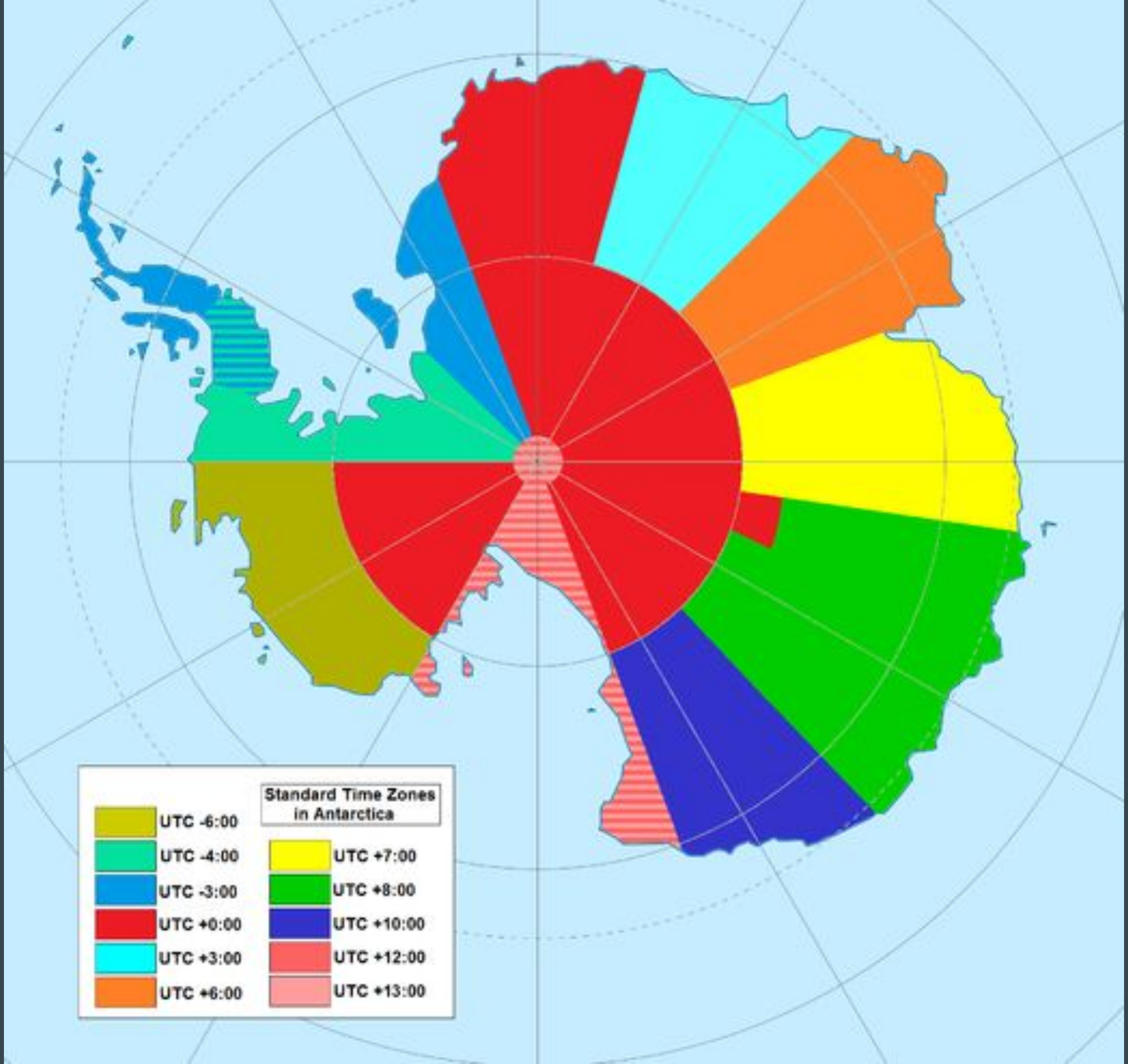
ZonedDateTime

`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

Collection Factory Methods

List.of, Set.of, Map.of, Map.ofEntries

Collection Factory Methods

```
List.of(a, b, b, c, ...)

Set.of(a, b, b, c, ...)

Map.of(k1, v1, k2, v2, k3, v3, ...)

Map.ofEntries(
    Map.entry(k1, v1),
    Map.entry(k2, v2),
    Map.entry(k3, v3), ...)
```

Local Variable Type Inference

The var reserved type name

var Data Type

Local variables only

- No fields
- No method parameters
- No method return types

var is a "reserved type name", not a keyword (can still have variable called "var")

Can also use on

- for loops
- try-with-resources blocks

var Data Type

Stuart Marks: Style Guidelines for Local Variable Type Inference in Java

- http://openjdk.java.net/projects/amber/LVTIstyle.html
- Local variables only

Features You Should Probably Know

HTTP Client

Built-in synch and asynch networking

HTTP 2 Client

- New HTTP Client API
  - Supports HTTP/2 and websockets
  - Replaces HTTPURLConnection
- Both synchronous and asynchronous modes

JShell

The Java REPL

JShell

- Java interpreter
  - https://docs.oracle.com/en/java/javase/11/jshell/introduction-jshell.html
- > `jshell` (or add -v for verbose)
- jshell>
  - /exit to leave
- No semicolons needed

Enhanced Switch Statement

Makes switch useable

Enhanced Switch

- Expressions → return a value
- Arrow rather than colon → no fall through
- Multiple case labels
- Statement blocks → yield
- Exhaustive

Text Blocks

Multiline Strings

Text Blocks

- Use "triple double" quotes (""") *and a newline*
- Indentation based on closing """
- `stripIndent`, `indent`, `translateEscapes`

Records

Preview feature of Java 14

Records

- Like a data class → intended to hold data
- Add attributes using constructor syntax
- generates getter methods
- final
- extends java.lang.Record
- generates toString, equals, and hashCode
- can add static fields

Pattern Matching

Preview feature of Java 14

Pattern matching

- Enhances the `instanceof` operator
- `if (shape instanceof Square s)` → use square methods on s
- Like a "smart cast"

Miscellaneous Features

Private Methods in Interfaces

Both `default` and `static` methods in interfaces

can call `private` methods

Try-With-Resources

Always had to declare variable inside the `try` block parentheses

Can now declare try-block variable outside

```
public void loadDataFromDB() throws SQLException {
    Connection dbCon = DriverManager.getConnection(url, user, password);
    try (dbCon; ResultSet rs = dbCon.createStatement().executeQuery("select * from emp")) {
```

dbCon variable will automatically be closed (no `finally` needed)

Deprecated Annotation

`@Deprecated` now has fields:

- `forRemoval`
- `since`

Tool `jdeprscan` to scan a jar file for deprecated uses

SafeVarargs

Until Java 8, `@SafeVarargs` could only be applied to:

- static methods
- final methods
- constructors

In Java 9, can add @SafeVarargs to private methods

Features You Can Probably Skip

The Module System

The Good and Bad of JPMS

JPMS

- Module descriptors
  - `module-info.java`
- exports, requires, opens, ...
  - Quick start guide:
    - http://openjdk.java.net/projects/jigsaw/quick-start
  - State of the Module System
    - http://openjdk.java.net/projects/jigsaw/spec/sotms/

JPMS

- module name ⟶ use "reverse dns" (like packages)
  - `requires` ⟶ add a module to the "module path"
    - `java.base` added automatically
    - `transitive` ⟶ any package using this module can read the arg
  - `exports` ⟶ list of packages exported by a module
    - can export to selected modules

JPMS

- Changes the nature of public and private
  - Reflection only works on opened packages
    - Use "`opens`" to expose a package to reflection
- requires `static` $\rightarrow$ make available at compile time but not runtime (optional)

Summary

- Need to know functional features
  - Streams with map / filter / reduce
  - Lambda expressions
  - Method references
  - Concurrent, parallel streams
- Need to use Optional
- Helpful to know preview features
  - Enhanced switch
  - Text blocks
  - Records
  - Pattern matching
- Can probably ignore modules (unless you're a library developer)