Java 8 Recipes

Ken Kousen
NFJS

Contact Info

Ken Kousen

Kousen IT, Inc.

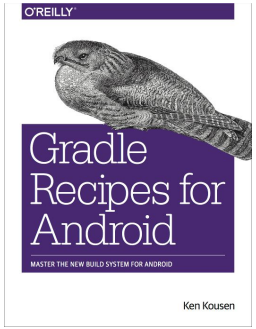
ken.kousen@kousenit.com

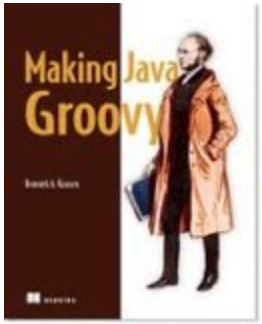
http://www.kousenit.com

http://kousenit.wordpress.com (blog)

@kenkousen

https://github.com/kousen (repo)





Publications

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

Lambda Expressions

Basic Syntax

parameters → expression

Parameters may be in ()

Expression may be in { }

Lambda Expressions

- Parameter types inferred from use

Lambda Expressions

Parameter types inferred from use

If expression is a single statement,
can omit braces

Lambda Expressions

Parameter types inferred from use

If expression is a single statement,
can omit braces

If single parameter, can omit parens

Lambda Expressions

```
(String s1, String s2) -> {
   return Integer.compare(
      s1.length(), s2.length())
}

(s1, s2) -> Integer.compare(
      s1.length(), s2.length())
```

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)
```

Functional Interfaces

Consumer → single arg, no result

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

Predicate → returns boolean

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

Functional Interfaces

Consumer → single arg, no result

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

Predicate → returns boolean

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

Supplier → returns single result

```
T get()
```

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

Often 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

BinaryOperator extends BiFunction

Functional Interfaces

Existing interfaces from standard library

FileFilter

```
boolean accept(File pathname)
```

Can assign lambda to interface reference

Exceptions

Only checked exceptions declared
    in the abstract method can be thrown

Either
    Catch others in body of lambda
    Define your own interface with exceptions

Method References

Method references use :: notation

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

Method References

Three general forms:

object::instanceMethod

Class::staticMethod

Class::instanceMethod

Constructor References

Can call constructors

```
MyClass::new
MyClass[]::new
```

Closure Variables

Variables in scope can be used in lambdas

Vars must be final

or "effectively final" (can not change)

Scope vars + lambda == closure

Closure Variables

If external var is a reference

can modify object assigned to it

Legal, but not safe

Default methods

Default methods in interfaces

- Use keyword `default`

Add method to existing interface

- All existing implementations break
  - until they add the method

Default methods

Add new method as default

All existing implementations gain method

See `forEach` in `Iterable`

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

Streams

Use the `stream()` method on collections

(It's a default method in Collection)

(Yes, let's pretend we never heard of

I/O streams…)

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.emptyStream()
```

Transforming Streams

Process data from one stream into another

```
filter(Predicate<T> p)
map(Function<T,R> mapper)
```

(Yeah, let's pretend we never heard of Map either…)

Transforming Streams

There's also flatMap:

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

Wait, what?

Used when composing functions

(wait until we get to optionals…)

Substreams

`limit(n)` returns a new stream

ends after n elements

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

Sorting

`Collections.sort(...)`

destructive; sorts in place

`Stream.sorted(...)`

returns a new sorted stream

Streams

Streams are lazy

computation on source only when terminal operation initiated

source elements consumed as needed

Reductions

Reduce a stream to a value

count, max, min, …

Return an Optional

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 function

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

`orElse()` provides an alternative

```
optional.orElse(... default …)
optional.orElseGet(... function …)
```

Reductions, redux

More generally, reduce

```
reduce(BinaryOperator<T> accumulator)
```

Also overloaded to take an initial value

Collecting values

From stream to array

```
stream.toArray(T[]::new)
```

From stream to collection

```
stream.collect(Collectors.toList())
```

Collecting values

Collectors utility class

```
toList(), toSet(), toCollection()
joining() // or with delimiter
summarizing(Int|Long|Double)
toMap(key,value)
```

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`

Date and Time API

java.time

- objects are immutable
- `Instant` → point on timeline
- `Duration` → time between two Instants
- `LocalDateTime` → no time zones
- `ZonedDateTime` → has time zones

Date and Time API

Requirements:

- Every day have exactly 86,400 sec
- Must match official time at noon
- Closely match elsewhere

Instant

`Instant` → a point on the time line

- `Instant.EPOCH` → Jan 1, 1970 UTC
- `Instant.MAX, Instant.MIN`
  - go forward and back
  - about a billion years
- `Instant.now()` → now

Duration

`Duration.between(instant1,instant2)`

returns a Duration

methods: `toNanos, toMillis,`
`toSeconds, toMinutes,`
`toHours, toDays`

LocalDate

A date without time zone info

contains year, month, day of month

```
LocalDate.of(2015, 2, 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

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

ZonedDateTime

LocalDateTime → ZonedDateTime

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

Instant → ZonedDateTime

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

Concurrency

Atomic classes from Java 5

- `AtomicInteger, AtomicLong`
- new methods like
  - `updateAndGet`
  - `accumulateAndGet`

Miscellaneous

Nashorn

JavaFX

Conclusions

They remade the whole language

- lambdas
- streams
- method references
- functional interfaces
- date/time API
- concurrency, and more...