Functional Programming

With Streams and Lambdas

Contact Info

Ken Kousen

Kousen IT, Inc.

ken.kousen@kousenit.com

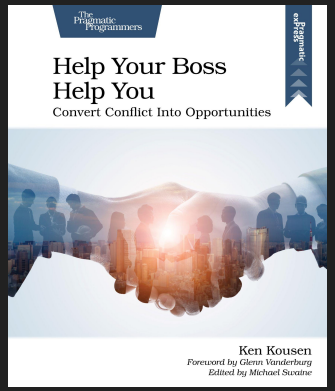
http://www.kousenit.com

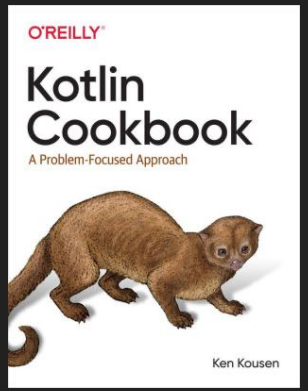
http://kousenit.org (blog)

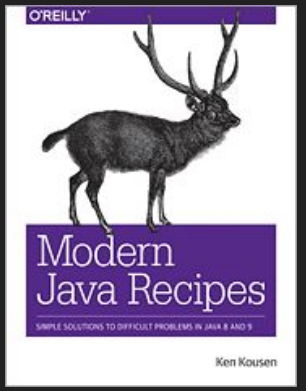
@kenkousen (twitter)

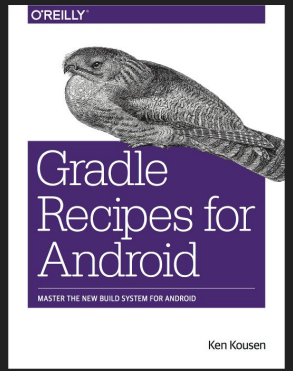
Tales from the jar side (free newsletter)

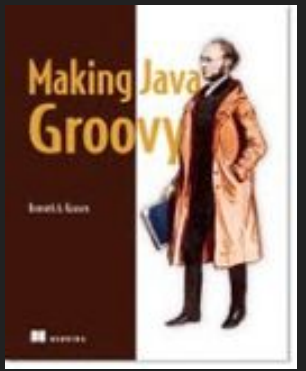
https://kenkousen.substack.com











GitHub repository

All demo code is at:

https://github.com/kousen/functional_java

Java

Recently had its 26th anniversary

Managed by Oracle, but with many implementations

OpenJDK, Azul, Amazon Coretto, SAP, …

Functional features added in version 1.8

Current LTS version is 17 (14 September 2021)

Documentation pages

https://docs.oracle.com/en/java/javase/11/ (replace 11 with any other version)

- 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 3.0.0 - 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

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 Interfaces

Interface with a Single Abstract Method

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

`@FunctionalInterface`

Not required, but used in library

Functional Interfaces in the JDK

- See `java.util.function` package
  - 43 interfaces (Java 17), but only 4 categories

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 for each generic variable

```
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

Use :: notation

```
System.out::println
    x → System.out.println(x)
    context variable is method argument
Math::max
    (x,y) → Math.max(x,y)
String::length
    x → x.length()
    context variable is method target
```

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

No data is processed unless the stream has a terminal operation

Reduction Operations

Reduction operations

Terminal operations that produce one value from a stream

- reduce (several overloads)
- collect (several overloads)

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

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

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`
- Don't need to implement the interface
- Just call the method from the interface name

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)
```

Functional Programming

Lambda, Method References, and Streams

- LambdaDemo.java
- MapFilterReduce.java
- PrimeChecker.java
- StreamsDemo.java

Lazy Streams

- Java Streams are lazy
  - Only process as much data as needed
  - LazyStreams.java

map, filter, reduce

```
int total = myNums.stream()
    .filter(n → n % 3 == 0)
    .map(n → n * 2)
    .reduce(0, (acc, val) → acc + val);

total = myNums.stream()
    .filter(n → n % 3 == 0)
    .mapToInt(n → n * 2)   // map to IntStream
    .sum();
```

Function Composition

Combining lambdas

Java library only combines:

Consumers with Consumers
Predicates with Predicates
Functions with Functions

CombineLambdas.java

Summary

- Java is an OO language with functional features
- Lambdas, method references treat methods like objects
- Streams are lazy and don't modify their source
- Functional programming favors immutability
  - Works well with records
- Java functional features are limited
  - Some function composition, but not much
  - No built-in currying, memoization, trampolining, etc.