FUNCTIONAL PROGRAMMING IN JAVA 6

JAN OUWENS

ABOUT ME

JAN OUWENS

PROGRAMMING JAVA SINCE 2005

FP SINCE ±2011

CODESTAR **(POWERED BY ORDINA)**

EQUALSVERIFIER

JAN.OUWENS@GMAIL.COM

@JQNO

LET'S START WITH A PUZZLER

```
public class Looper {
    public static void main(String[] args) throws Exception {
        int five = 5;

        Field field = Integer.class.getDeclaredField("value");
        field.setAccessible(true);
        field.set(five, 4);

        for (Integer i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }
}
```

WHAT'S THE RESULT?

- THROWS AN EXCEPTION
- 1. 2. 3. 4. 5. 6. 7. 8. 9. 10
- INFINITE LOOP
- 1. 2. 3. 4. 6. 7. 8. 9. 10

WHAT'S THE RESULT?

- THROWS AN EXCEPTION
- 1. 2. 3. 4. 5. 6. 7. 8. 9. 10
- INFINITE LOOP
- 1. 2. 3. 4. 6. 7. 8. 9. 10

THINK ABOUT THIS

WHAT WOULD YOU DO IF YOUR COWORKER WROTE THIS?

WHAT IS FUNCTIONAL PROGRAMMING?

YOU ALREADY KNOW
FUNCTIONAL PROGRAMMING

A20

Running.Totals

	A	B	C	D	E	F	G
1						*Installed Base*	*Installed Base*
2		February	March	April	This Month	Before	After
3	*Pacific*						
4	Deluxe	654	723	734	768	9570	12449
5	Standard	462	582	54			
6	Economy	276	354	32			
7							
8	*Midwest*						
9	Deluxe	812	768	64			
10	Standard	467	554	46			
11	Economy	412	354	43			
12							
13	*Rockies*						
14	Deluxe	687	676	75			
15	Standard	453	582	54			
16	Economy	387	401	34			
17							
18	*South*						
19	Deluxe	746	645	76			

Update macro

	A
1	Update
2	=SELECT("R3C2:R25C2")
3	=COPY()
4	=SELECT("R3C6:R25C6")
5	=PASTE.SPECIAL(3,2)
6	=SELECT("R2C3:R25C4")
7	=CUT()
8	=SELECT("R2C2:R25C3")
9	=PASTE()
10	=SELECT("R3C5:R25C5")
11	=CUT()
12	=SELECT("R3C4:R25C4")
13	=PASTE()

IT'S PROGRAMMING WITH FUNCTIONS

- PURE FUNCTIONS
- WITH NO SIDE-EFFECTS
- THAT'S ALL
- THAT'S EVERYTHING

GIVEN

```
public int multiply(int x, int y) { return x * y; }
```

WHEN

```
int product = multiply(3, 4);
```

THEN

```
int product = 12;
```

```
multiply(3, 4) == 12
```

ALWAYS

WHEN **YOU FIRST CALL IT**
WHEN **YOU CALL IT A SECOND TIME**
WHEN **YOU CHANGE A THING THEN CALL IT AGAIN**
WHEN **AN EXCEPTION IS THROWN**
WHEN **YOU UPDATE THE DATABASE THEN CALL IT AGAIN**

THIS IS KNOWN AS REFERENTIAL TRANSPARENCY

TAKE THIS TO THE EXTREME, AND YOU GET HASKELL
TAKE THIS A LITTLE LESS FAR, AND YOU GET SCALA
DON'T DO THIS AT ALL, AND YOU GET C

WHY?

- IT MAKES THREADS EASIER
- IT MAKES EVERYTHING EASIER

THREADS?

THE HARD THING ABOUT THREADS IS SHARED MUTABLE DATA

LET'S SAY WE HAVE AN ARRAY

```
g = [1, 2, 3, 4, 5]
```

THREAD A AND THREAD B BOTH ACCESS IT

```
g = [1, 2, 3, 4, 5]
```

```
Thread A:
arr = g
```

```
Thread B:
arr = g
```

WHAT IF THREAD B CHANGES IT?

```
g = [1, 2, 42, 4, 5]
```

```
Thread A:
arr = g
```

```
Thread B:
arr = g
arr[2] = 42
```

THE VALUE IN A DEPENDS ON TIMING

```
g = [1, 2, 42, 4, 5]
```

```
Thread A:
arr = g

println(arr)
```

```
Thread B:
arr = g
arr[2] = 42
```

CAN PRINT DIFFERENT THINGS

```
[1, 2, 3, 4, 5]

[1, 2, 42, 4, 5]
```

WHAT IF WE WANT TO SORT IT?

```
g = [1, 2, 4, 5, 42]
```

```
Thread A:
arr = g


println(arr)
```

```
Thread B:
arr = g
arr[2] = 42
sort(arr)
```

CAN PRINT EVEN MORE DIFFERENT THINGS

```
[1, 2, 3, 4, 5]

[1, 2, 42, 4, 5]

[1, 2, 4, 5, 42]

[1, 2, 42, 5, 42]
```

NOW WE HAVE TO SYNCHRONIZE

```
g = [1, 2, 4, 5, 42]
```

```
Thread A:
arr = g




synchronized {
  println(arr)
}
```

```
Thread B:
arr = g
synchronized {
  arr[2] = 42
  sort(arr)
}
```

REFERENTIAL TRANSPARENCY TO THE RESCUE

```
g = [1, 2, 4, 5, 42]
```

```
Thread A:
arr = g


println(arr)
```

```
Thread B:
arr = g
newArr = arr.updated(2, 42)
sortedArr = sort(newArr)
```

EASIER?

CONSIDER THESE REACTIONS TO ESSENTIALLY THE SAME THING

MODIFYING INTEGER

WTF!?!

DEFENSIVE COPIES OF GREGORIANCALENDAR & ARRAY

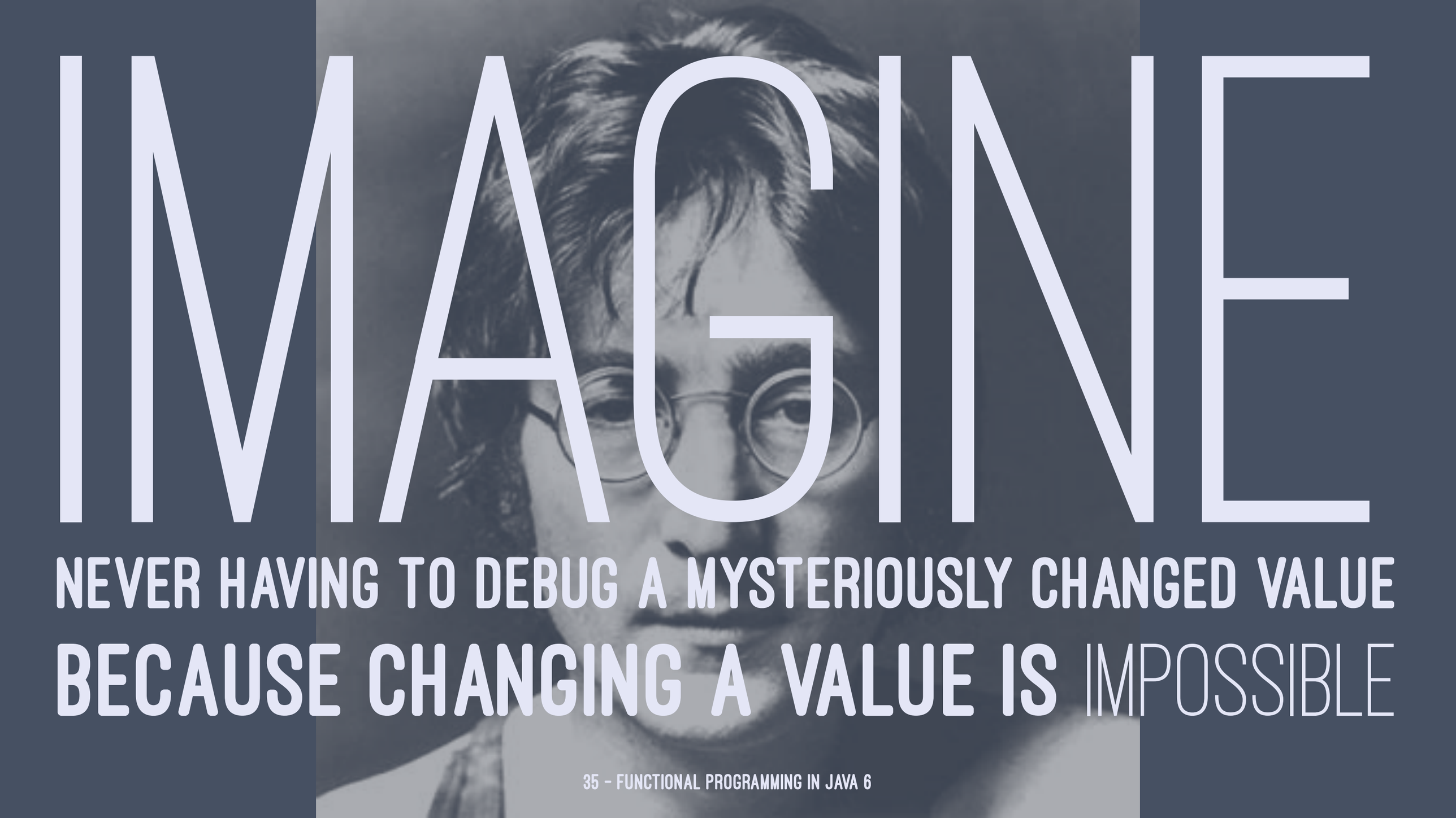
ANNOYING FACT OF LIFE

JAVA CLASSES

MEH

THESE ARE ALL VALUE OBJECTS
WHY TREAT THEM DIFFERENTLY?

OK MAYBE BECAUSE JPA WON'T LET YOU BUT ISN'T THAT PART OF THE PROBLEM?



IMAGINE

NEVER HAVING TO DEBUG A MYSTERIOUSLY CHANGED VALUE

BECAUSE CHANGING A VALUE IS IMPOSSIBLE

HOW TO DO IMMUTABILITY IN JAVA?

- MAKE EVERYTHING FINAL
- MAKE DEFENSIVE COPIES OF MUTABLE TYPES
- OR SIMPLY AVOID USING THEM

INSTEAD OF GREGORIANCALENDAR
USE JODA-TIME

INSTEAD OF JAVA.UTIL.LIST
USE GUAVA'S IMMUTABLELIST

IF YOU CAN'T AVOID MUTABILITY
SHOVE IT INTO A CORNER
DON'T LET IT ESCAPE!

BUT

WHAT ABOUT PERFORMANCE?

COPYING ALL THAT DATA **MUST BE ROUGH**

PERSISTENT DATA STRUCTURES

LINKED LIST

x → y → z

PERSISTENT DATA STRUCTURES

PRE-PENDING TO A LINKED LIST

a → x → y → z

NO COPYING **OF DATA**
JUST RE-USING **WHAT'S THERE**

IN FUNCTIONAL LANGUAGES,
ALL DATA STRUCTURES ARE LIKE THIS

VERY EFFICIENT

IN JAVA.
YOU'RE SHIT OUT OF LUCK

BUT STILL

IS THERE **A PERFORMANCE PROBLEM?**

[YES / NO]

IS THERE **A PERFORMANCE PROBLEM?**

[YES / NO]

USE IMMUTABLE DATA STRUCTURES

IS THERE **A PERFORMANCE PROBLEM?**

[YES / NO]

DID YOU MEASURE THE PERFORMANCE?

[YES / NO]

DID YOU MEASURE THE PERFORMANCE?

[YES / NO]

PICK UP A PROFILER

DID YOU MEASURE THE PERFORMANCE?

[YES / NO]

WAS THE IMMUTABLE DATA STRUCTURE YOUR BOTTLENECK?

[YES / NO]

WAS THE IMMUTABLE DATA STRUCTURE YOUR BOTTLENECK?

[YES / NO]

USE IMMUTABLE DATA STRUCTURES

WAS THE IMMUTABLE DATA STRUCTURE YOUR BOTTLENECK?

[YES / NO]

USE SOMETHING MUTABLE

IF YOU CAN'T AVOID MUTABILITY
SHOVE IT INTO A CORNER
DON'T LET IT ESCAPE!

A FINAL THOUGHT

LET JAVA BE JAVA

DON'T TURN IT INTO SOMETHING IT'S NOT

IT'S PAINFUL TO SEE JAVA
PRETENDING TO BE C

IT'S JUST AS PAINFUL TO SEE JAVA
PRETENDING TO BE SCALA

IMPORTANT IN FP: FUNCTIONS ARE FIRST CLASS
A.K.A. "LAMBDAS"

BUT JAVA 6 HAS NO LAMBDAS

FOR EXAMPLE

```
List<Person> people = ...;
Iterable<String> names = Iterables.transform(people, new Function<Integer, String>() {
    @Override
    public String apply(Person input) {
        return input.getName();
    }
});
```

VERSUS

```
List<Person> people = ...;
List<String> names = new ArrayList<>();
for (Person p : people) {
    names.add(p.getName());
}
```

WHICH IS MORE READABLE?

WHICH IS MORE FAMILIAR?

DO GUAVA'S FP FEATURES REALLY FIT THE LANGUAGE?

"EXCESSIVE USE OF GUAVA'S FUNCTIONAL PROGRAMMING IDIOMS CAN LEAD TO VERBOSE, CONFUSING, UNREADABLE, AND INEFFICIENT CODE."

– THE GUAVA TEAM

THIS IS STILL REFERENTIALLY TRANSPARENT!

```
public List<String> findNamesOf(List<Person> people) {
    List<String> names = new ArrayList<>();
    for (Person p : people) {
        names.add(p.getName());
    }
    return names;
}
```

SHOVE **THE MUTABILITY INTO A** CORNER

NAMES **IS THE ONLY MUTABLE VARIABLE**

- FIRST **IT'S** COMPLETELY **CONSTRUCTED**
  - THEN **IT'S RETURNED**

THIS IS EVEN BETTER

```
public ImmutableList<String> findNamesOf(List<Person> people) {
    List<String> names = new ArrayList<>();
    for (Person p : people) {
        names.add(p.getName());
    }
    return ImmutableList.copyOf(names);
}
```

IN CONCLUSION

- IMMUTABILITY **SHOULD BE YOUR DEFAULT**
  - **USE** FINAL **WHERE POSSIBLE**
- DON'T **GO CRAZY WITH ANONYMOUS INNER CLASSES**

IF YOU ONLY REMEMBER ONE THING, MAKE IT THIS:

REFERENTIAL TRANSPARENCY

BECAUSE RT WILL

- ENABLE ASYNCHRONICITY WITHOUT LOCKING
- ALLOW YOU TO CALL A GETTER IN A TIGHT INNER LOOP (WITHOUT WORRYING ABOUT ROUND-TRIPS TO THE DATABASE)
- REDUCE YOUR TIME WASTED IN THE DEBUGGER

THANK YOU

QUESTIONS?