Don’t use Lombok – Alternatives

Dealing with boilerplate code. No runtime dependencies
Extra code obscures that our class is simply a data class
A better approach - declare class as a data class.
Separation of concerns and minimization of mutability.

Azat Satklichov
azats@seznam.cz,
http://sahet.net/htm/java.html,
https://github.com/azatsatklichov/LombokAndAlternatives

My Blog: Beauty of Modern Java — Handling Boilerplate Code Using Records

Lombok

https://projectlombok.org/

The word Lombok (island in Indonesia) is a word that comes from the local Sasak language. Translated into Indonesian it means lurus or straight.   Lombok is also a lesser-used word for chili in Bahasa Indonesian, which has led many people to believe the island is named for its **spicy cuisine**.

v1.18.16 (October 15th, 2020)
.. v1.18.14 (broken), v1.18.12() – Java 13,14 (yield)
v1.18.10 (September 10th, 2019)

Lombok, is a library used to reduce boilerplate code (simplifying data objects) for model/data objects. e.g., can generate getters and setters, toString, hashCode, ….

- @Slf4j|@Log4j|@Log, @NonNull, @Synchronized, @Getter, @Setter, @ToString, @EqualsAndHashCode - provide logger, puts null-check (NPEs), thread-safety, getters, setters, toString, equals and hashCode `javax.annotation.Nonnull` is part of JSR-305, **dormant** since 2012. Used by *Findbugs, .. SonarQube*
- Guava -> forces us to require jsr305 automatic module,
- `@ToString(exclude = {“events”}), or on field @ToString.Exclude` **??** StackOverflow.. (biDir)
- `@EqualsAndHashCode(of = {“authToken”}). Also can be` **`excluded`**

All fields marked as static, transient will not be considered for hashCode and equals

- `@Getter(AccessLevel.PRIVATE)` … `Waiter(); DatabusBrokerTest`

- Lazy Getter (performance: cache it to allow in-memory reads or retrieve when its needed)

```
@Getter(lazy = true) private final Map<String, Long> transactions =
getTransactions();
```

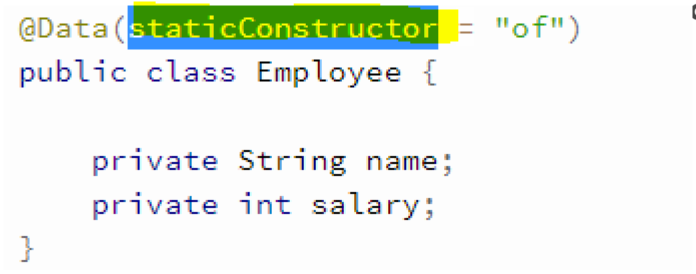
- @Accessors(fluent = .., chain = .., prefix) e.g. `DatabusBrokerTest, DatabusConfigProvider`

```
 e.g. private BigDecimal bdBalance; → @Accessors(prefix = {“bd"}) ->
obj.getBalance()
```

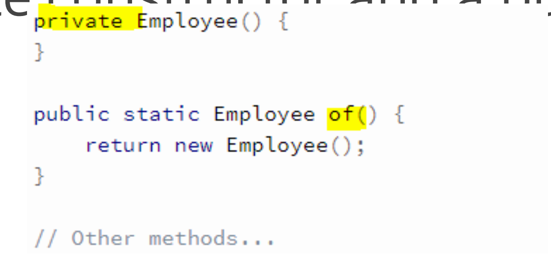
```
- Also project-wide: Lombok.config
```

- @NoArgsConstructor, @RequiredArgsConstructor (for the final and @NonNull fields), @AllArgsConstructor - `DataEvent,UnknownEventSubscriber(X)`
- `- for all e.g.` @AllArgsConstructor(staticName = "of"), - allows creation of static factory method

@Data - get all this for free: A shortcut for @ToString, @EqualsAndHashCode, @Getter on all fields, @Setter on all non-final fields, and @RequiredArgsConstructor!
e.g. `UnknownEvent, CobolDocumentModel`
- staticConstructor = "of", Lombok generates a private constructor and a public static factory method:



```
@Data(staticConstructor = "of")
public class Employee {

    private String name;
    private int salary;
}
```



```
private Employee() {
}

public static Employee of() {
    return new Employee();
}

// Other methods...
```

- DTO - @Value (thread safe), immutable entity (final class with imm. members, **@Value** is immutable variant of @Data): No setter by Default, constructor arguments (except final fields that are initialized in the field declaration) is also generated. @Value is shorthand for: final @ToString @EqualsAndHashCode @AllArgsConstructor @FieldDefaults(makeFinal = true, level = AccessLevel.PRIVATE) @Getter .
  Also, *any* explicit constructor, no matter the arguments list, implies lombok will not generate a constructor

`e.g. Locality, ElementItem`
- @Value(staticConstructor = "of"), to make some fields @NonFinal

- @Builder - build immutable data objects with their simple, fluent syntax. If needed a builder for specific fields, we should create a constructor/method with only those fields.

- @Singular - the builder doesn't generate a *setter* method. Instead, it generates two *adder* methods.

`e.g. SyntaxError, AnalysisFinishedEvent`

.rules(Arrays.asList("rule1","rule2")) → .rule("rule1").rule("rule2").

- Builder with Default Value

1. `via toBuilder()`
2. `@Builder.Default - e.g AnalysisFinishedEvent`

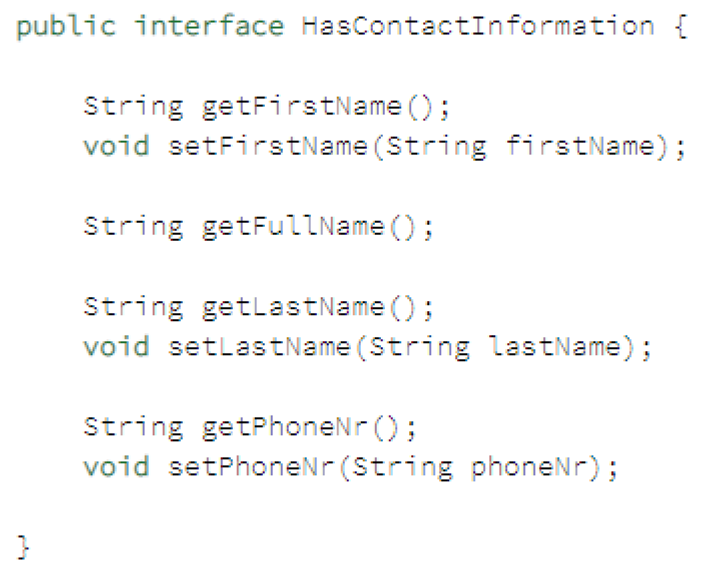
lombok.singular.useGuava to *true*, Lombok uses Guava's immutable builders and types.

CheckedExceptions and Ensure Your Resources Are Released

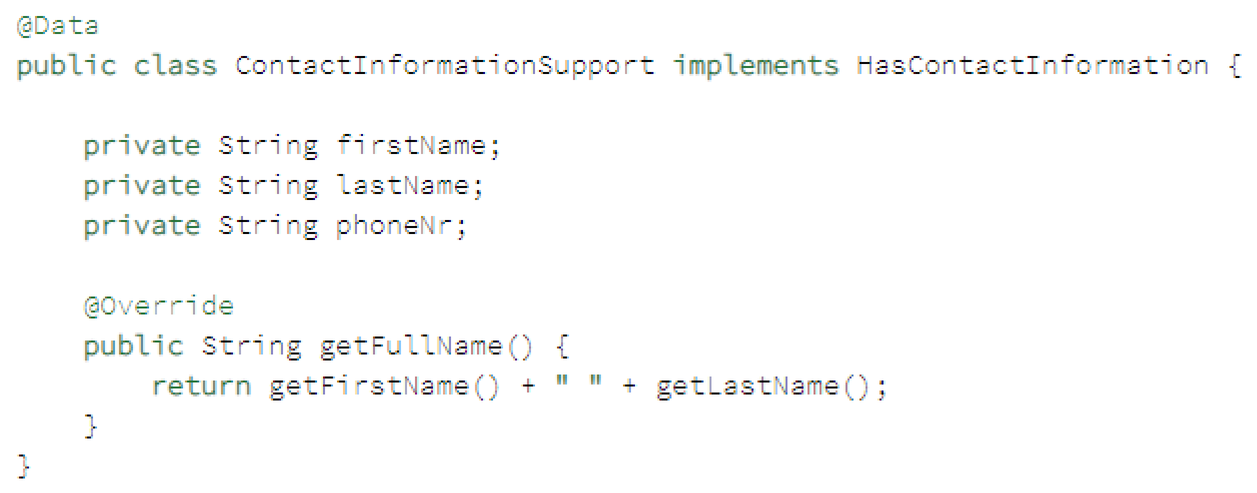
`@SneakyThrows, @Cleanup`

e.g. `Messages`

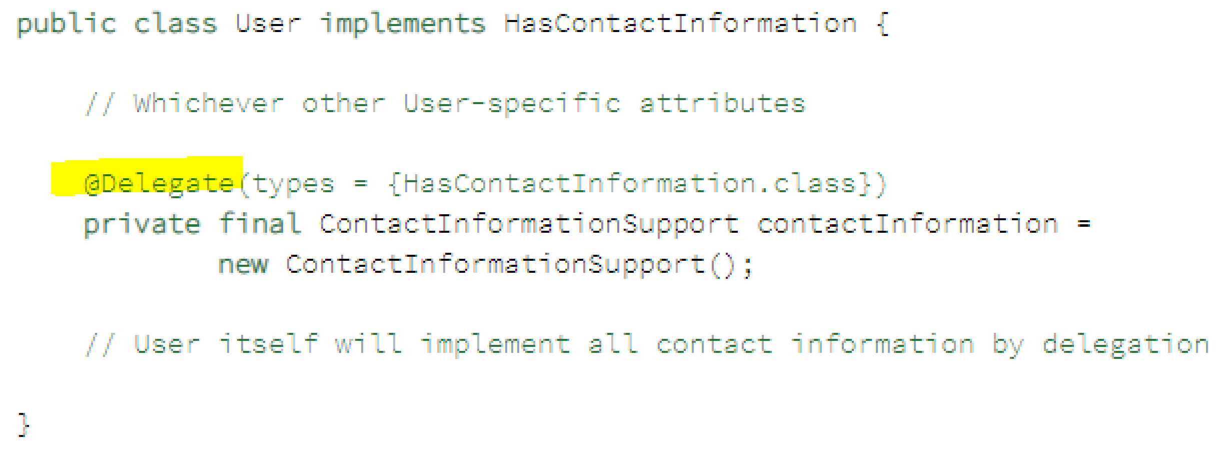
Experimental: Automate Objects Composition - “favor composition inheritance”, (*Traits* or *Mixins*)



```java
public interface HasContactInformation {

    String getFirstName();
    void setFirstName(String firstName);

    String getFullName();

    String getLastName();
    void setLastName(String lastName);

    String getPhoneNr();
    void setPhoneNr(String phoneNr);

}
```



```java
@Data
public class ContactInformationSupport implements HasContactInformation {

    private String firstName;
    private String lastName;
    private String phoneNr;

    @Override
    public String getFullName() {
        return getFirstName() + " " + getLastName();
    }
}
```



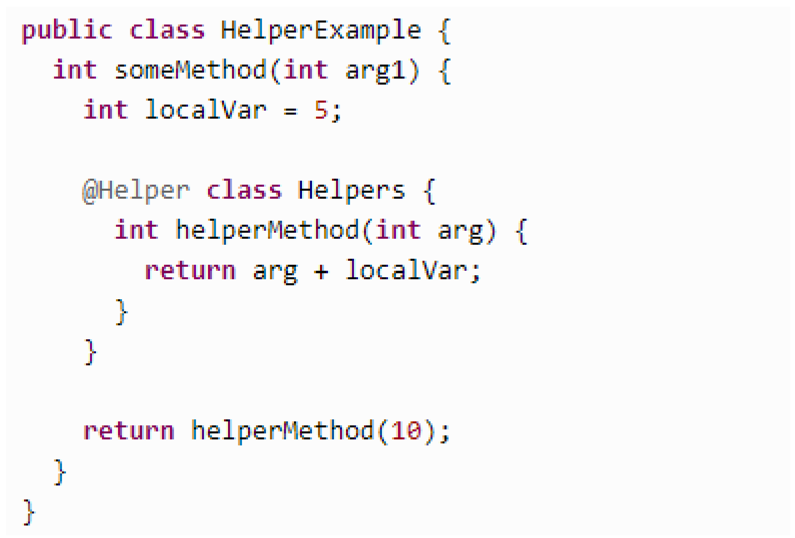
```java
public class User implements HasContactInformation {

    // Whichever other User-specific attributes

    @Delegate(types = {HasContactInformation.class})
    private final ContactInformationSupport contactInformation =
            new ContactInformationSupport();

    // User itself will implement all contact information by delegation

}
```

Experimental Lombok - https://projectlombok.org/features/experimental/all

@UtilityClass

@Accessors

@Helper - you can declare classes inside methods



```
public class HelperExample {
  int someMethod(int arg1) {
    int localVar = 5;

    @Helper class Helpers {
      int helperMethod(int arg) {
        return arg + localVar;
      }
    }

    return helperMethod(10);
  }
}
```

@Delegate

Val: val example = **new** ArrayList<String>();,

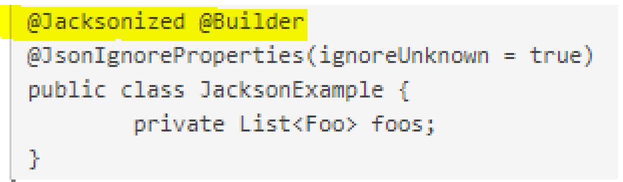
CobolLineIndicatorProcessorImplTest

Var(promoted): var (mutable) like val, but *not* marked as final

@Tolerate - lombok pretend it doesn't exist, i.e., to generate a method which would otherwise be skipped due to possible conflicts. Similar concept: **Hijack in ClearCase (or ignore)**

@Jacksonized - v1.18.14: add-on annotation for @Builder, @SuperBuilder,

auto- configures builder to be used by Jackson's deserializatic



```
@Jacksonized @Builder
@JsonIgnoreProperties(ignoreUnknown = true)
public class JacksonExample {
        private List<Foo> foos;
}
```

Promoted: **@Value**, **@Builder**, @Wither: renamed to **@With**, and promotec
Immutable 'setters' - methods that create a clone but with one changed field.

- Delombok: java -jar lombok.jar delombok src -d src-delomboked
- Or and Ant-task:

Delombok tries to preserve your code as much as it can, but comments may move around a little bit, especially comments that are in the middle of a syntax node.

- @Builder - build immutable data objects with their simple, fluent syntax

@SuperBuilder - https://www.baeldung.com/lombok-builder-inheritance

- `@Setter` → `@With:` The next best alternative to a setter for an immutable property is to construct a clone of the object, but with a new value for this one field.

- **if the superclass doesn't have a no-args constructor, Lombok can't generate any constructor in the subclass**

Why Not Lombok

- Lombok is largely about *syntactic* convenience (*extralinguistic*); it is a macro-processor pre-loaded with some known useful patterns of code.

- More bytecode manipulation tools tends to make the app. behavior less deterministic in prod. (already aspectj with Spring)
- Invisible code (poor experience for debuggers and code explorers), compiler hacking ;),
- Mixing to other analysis tools (code coverage, findbugs, checkstyle, JRebel, ..) can be tricky and messes up the metrics.
- Lombok-enabled classes/jars can't be used easily anymore by third parties. Ex: debug step (decompile, or delombok, ..)
-
- It heavily relies on APIs internal to the compiler, which can change at any time and which means projects using it can break on any minor **Java update**.

- Code generation can be handled by IDEs or other tools

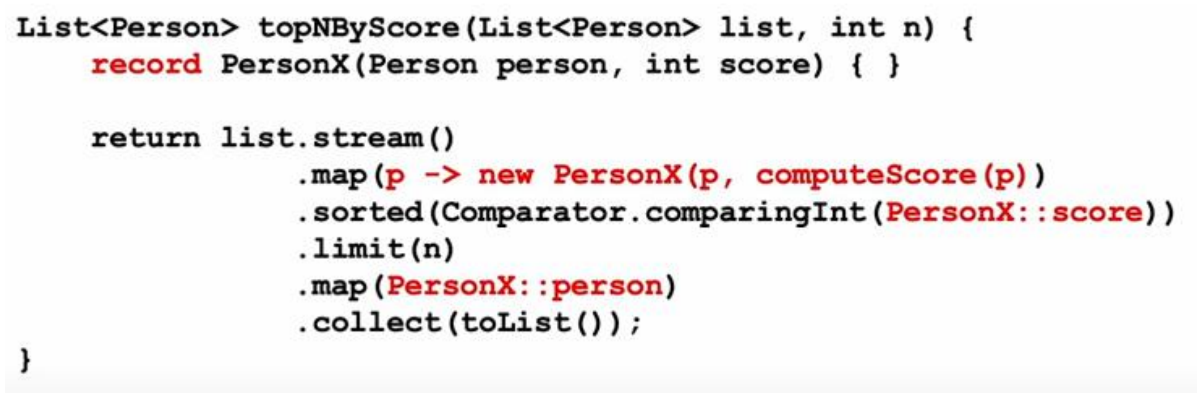
- Avoiding OO principles - @Getters, .. . Too many side effects, both in the bytecode and on developer's behavior/**misuse (**, introduces bugs hard to detect..**)**

- E.g. Import static @Utility.. Not buildable, @NonNull (use Guava Preconditions) @Cleanup (no need Java 7 try-catch), @SneakyThrows (just not use CheckedEx.), @val, @var (already in Java 10), @toString or ..[stacked once used with JAXB or Jackson, ORM],
- Migration to KOTLIN … delombok ;)

Lombok   Alternatives

Java 14 Record, AutoValue, and Immutables

Separation of concerns and **minimization of mutability**.



```
List<Person> topNByScore(List<Person> list, int n) {
    record PersonX(Person person, int score) { }

    return list.stream()
                .map(p -> new PersonX(p, computeScore(p)))
                .sorted(Comparator.comparingInt(PersonX::score))
                .limit(n)
                .map(PersonX::person)
                .collect(toList());
}
```

Lombok **@Value**
This is close to what records are

Can records replace lombok? **No.** Record's can't provide mutable Object. E.g. @Data in Lombok, Builder, etc. ..

Local RECORD not polluting program name spaces, just serving here. Like local classes.

- **Immutable objects** are constructed once, in a consistent state, and can be safely shared
  <u>Will fail if mandatory attributes are missing </u>(Lombok not guarantee this)
  Cannot be sneakily modified when passed to other code
- **Immutable objects** are naturally thread-safe and can therefore be safely shared among threads
  No excessive copying
  No excessive synchronization
- **Object definitions** are pleasant to write and read
  No boilerplate setter and getters
  No ugly IDE-generated hashCode, equals and toString methods that end up being stored in source control.

Java 14 [preview] RECORD (standard part of Java 16)

Records are immutable data classes that require only the type and name of fields. Good choice when modeling things like domain model classes (via ORM), or DTOs. E.g. Data holders: TypeScript property, or data classes in **Kotlin** or **Scala**.

Using records – with their compiler-generated methods – we can reduce boilerplate code and improve the reliability of our immutable classes. Records are a *semantic* feature; they are ***nominal tuples (data carriers)*** also tied later to Sealed classes.

The *equals*, *hashCode*, and *toString* methods, as well as the *private*, *final* fields, and *public* constructor, are generated by the Java compiler.

```
AdamRec adam2 = new AdamRec("Adam", "Mary");
```

- **Constructor:** public constructor is generated for us. Also can be **customized** (e.g. for validation purposes), or **default** one.

- **Getters, no setters:** public getters methods – names match the name of our field – for free. (like **TS style:** property)

- **equals(), hashCode(), toString():** generated. **Overriding** is possible..

```
System.out.println(adam2.name());
AdamRec[name=Adam, address=Mary]
```

- **Static variables and Methods:** can be included in records as like in regular classes **BUT,** user declared non-static fields <u>XYZ are not permitted in a record.</u> **In Enum OK.**

- Records are not by default **serializable**, but can implement java.io.Serializable marker interface (…)

Compiler generated methods already exist in JAVA Enum e.g.: values(), valueOf()

How does RECORD Look Under the Hood?

Record is just a class with the purpose of holding and exposing data.

```
>javac AdamRec.java
>javap -v -p AdamRec.class
```

1. The class is marked final (no sub.): public final class net.sahet.record.AdamRec **extends java.lang.Record** (like as Enum)

1. You can implement interfaces, as like Enums

2. Two private final fields: private final java.lang.String name; private final java.lang.String address;

3. generated public constructor: public net.sahet.record.AdamRec(java.lang.String, java.lang.String);

4. Two getters (accessors): public java.lang.String name(); public java.lang.String address();

5. Also generated: toString(), hashCode(), and equals() which are rely on invokedynamic [Indy].

6. Also see java.lang.ObjectMethods.BootstrapMethods

Alternatives to Lombok: AutoValue / Immutables

Project Lombok is a java library that automatically plugs into your editor and build tools, spicing up your java.
Never write another getter or equals method again, with one annotation your class has a fully-featured builder, Automate your logging variables, and much more.

Immutables Java **annotation processors** to generate simple, safe, and consistent value objects. Do not repeat yourself, try Immutables, the most comprehensive tool in this field!

AutoValue provides an **easier way to create immutable value classes**, with a lot less code and less room for error, while not restricting your freedom to code almost any aspect of your class exactly the way you want it.

A) The main difference between AutoValue/Immutables and Lombok is that the first ones **are based on interfaces to set the definitions of what will be generated** and the result will be a new class implementing the interface, instead of **Lombok which injects code inside an existing implementation class. Nulls….**

B) Immutables/AutoValue generate new classes with the **processor**, and Lombok modifies the *bytecode* of the original class.

C) Generated code is **visible**, no need to be in repository. **No RUNTIME** impact. Only simple POJOs.
D) It allows for **greater flexibility**, because you can actually change the builder method names, other oprations, (hash, ..).
E) Disadvantage: AutoValue_XYZ, or Immutable_XYZ, explicit defensive copy, order of params which may break tests.

Google AutoValue

AutoValue is a source code generator for Java, and more specifically it's a library for **generating source code for value objects or value-typed objects**.
Value-Typed Objects (immutable POJO**). Value-Types (not Java Beans** [+ default constructor and setter]**)** must consume all field values through a constructor or a factory method. Typically, value-types are immutable

e.g. final class AutoValue_Person extends Person

Why AutoValue
The reason value-types must be immutable is to prevent any change to their internal state by the application after they have been instantiated.
Issues With Hand-Coding : e.g. ImmutableMoney. Can be a bad design and a lot of boilerplate code. **IDEs to Rescue?** Not so

Use @AutoValue, then compiler generates _AutoValue_XYZ class, … . Basically you define your interfaces and the implementation is left to *AutoValue* generated code, you don't have to actually implement the code that is in getters and setters. ***Javac* will always regenerate updated code for us**.

What is generated is a value object with accessor methods, parameterized constructor, properly overridden *toString(), equals(Object)* and *hashCode()* methods.

- **Builders: @AutoValue.Builder:** Our AutoValue class does not really change much, except that the static factory method is replaced by a builder
- **Defensive Copies:** Java 10 introduced defensive copy static factory methods such as *List.copyOf*.

Immutables

The library generates (via utilizing annotation processing) immutable objects from abstract types: *Interface*, *Class*, *Annotation*
You can think of Immutables as Guava's Immutable Collections but for **regular objects.** The core of *Immutables* is **modelling**.

The key to achieving this is the proper use of ***@Value.Immutable/*****@Value.Modifiable** annotation. **It generates an immutable version of an annotated type and prefixes its name with the *Immutable* keyword**.

Generated class ImmutablePerson comes with implemented *toString*, *hashcode*, *equals* methods and with a builder *ImmutablePerson.Builder*. Notice that the generated constructor has *private* access. **Withers, ..**

*- You can customize generated class names to have other prefixes than Immutable**

@Value.Parameter can be used for specifying fields, for which constructor method should be generated.
@Value.Default annotation allows you to specify a default value that should be used when an initial value is not provided.

@Value.Auxiliary annotation can be used for annotating a property that will be stored in an object's instance, but will be ignored by *equals*, *hashCode* and *toString* implementations.

***@Value.Immutable(Prehash = True)* -** *hashCode()* computed only once during the object's (immutable obj.) instantiation

THANK YOU

Refernces:
https://codeburst.io/lombok-autovalue-and-immutables-or-how-to-write-less-and-better-code-returns-2b2e9273f877
https://cr.openjdk.java.net/~briangoetz/amber/datum.html
https://www.youtube.com/watch?v=J6fegDQPgps&t=1532s