Journal of Technological Innovations

Est. 2020

Eliminating Toil Of Writing Value Classes in Java

Nilesh Jagnik
Email: nileshjagnik@gmail.com

Abstract:

Writing and maintaining value classes (classes that are mainly data carriers) in Java can be a cumbersome task. This is because traditionally Java did not have special support for value classes which is present in other languages like Python. In this paper we discuss the use of Google's Auto Value library to create value objects. We also discuss Java Record classes, which are a new feature added in Java 16 to support value classes.

Keywords*:* value classes, programming best practices

Introduction

A Java class is a definition for an object that contains both data and logic. Classes can contain several pieces of logic that use data members to perform complex operations generating some intended output from provided input.

However, there can be classes which do contain any complex logic and are simply used as containers for carrying data. Such classes are called value classes. Value classes are intended to be representation for a set of data fields. Developers can create these classes to passes several pieces of data as a single unit.

In Java, there is no distinction between regular classes and value classes. This can lead to several challenges. Developers are required to write lengthy class definitions for what is a very simple collection of data. This can lead to toil of repeatedly writing similar classes and also introduce bugs as now more code needs to be written. In this paper, we discuss the use of an open-source library called AutoValue which automatically creates value classes. Later we also discuss Records, which is a language feature since Java SE 16. We will also present a comparison of these two options.

Usage of Value Classes

Value classes are intended for handling multiple pieces of data as a unit. This is useful because handling multiple data variables separately in code can become cumbersome. Imagine a situation where a we have ten related data variables that need to be accessed together. Passing these variables around would require adding these as individual parameters to every method that needs these. This would make code unnecessarily verbose. It would add considerable toil if these variables had to be passed through multiple modules and levels of abstraction. On top of this, if a new variable needs to be added to this group, it would require updating code everywhere these fields are passed.

This is the utility of value classes. Instead of handling multiple data variables we could create a single class which stores all these data variables as member fields. Then the code would only need to pass a single combined value object. Updating the value object to add fields is also easy and would only require changes to the code that creates and consumes the new fields.

Many other languages provide native support for value classes, e.g., Python and Kotlin have data classes. But this has been a missing feature in Java until the release of Record in Java SE 16.

Properties of Value Classes

There are certain desirable properties from a value class. As expected, depending on the applications, some of these properties may be more desirable in comparison to others. We will refer to these properties when we present the solutions for easily creating value classes.

Equality

Two value objects that are instances of the same value class are considered equal if all of their member fields are equal. This is true even if there are actually two separate objects in memory. This is similar to how the language treats primitives like integers and strings as equal if their values are equal. As an implication of this, any value class should define an equals member method which formalizes this notion of equality.

HashCode and ToString

Value classes also need a hashCode member method. Without this, it would not be possible to use value objects as keys for HashMaps, etc. Additionally, if two objects are equal according to equals then their hashCode should also match. To ensure pretty printing of value objects, value classes should also declare a toString member methods.

Immutability

Value classes should be immutable, i.e., the values of member fields should never change. This is because value classes are meant to imitate behavior of primitive data types like integers which cannot be mutated once set.

Problems With Writing Value Classes in Java

Verbosity

As displayed by the value class definition in Fig. 1, writing a value class in Java is quite verbose. This can add quite a lot of toil on software developers. Even when software projects have generic base classes that use reflection for implementing common methods like equals, hashCode and toString, there is still a cost for maintaining these base classes, which can often get pretty complex themselves.

Error Prone

In addition to toil added due to writing lot of repetitive code, the code itself can get complex over time, especially if base classes are created to prevent duplication of logic. This complexity can lead to the introduction of bugs. Additionally, this complexity hurts the overall readability of code and makes code reviews tougher.

```
public class Cuboid {
  public int length;
  public int width;
  public int depth;

  Cuboid(int length, int width, int depth) {
    this.length = length;
    this.width = width;
    this.depth = depth;
  }

  public int hashCode() {
    return Objects.hash(length, width, depth);
  }

  public boolean equals(Cuboid other) {
```

Fig. 1. A typical value class in Java

Autovalue

Value Class Definitions

The AutoValue library developed by Google provides an easy way to automatically generate value classes. The code developer provides a specification for the value class by creating an abstract class with the desired member fields. Any code that needs the value object can use the abstract class defined. The AutoValue framework automatically generates a full class matching the specification of the abstract value class. This frees developers from implementing these value classes themselves.

```
// This annotation triggers auto generation of
// the value class.
@AutoValue
abstract class Cuboid {
  static Cuboid create(
      int length,
      int width,
      int depth) {
    return new AutoValue_Cuboid(length, width, depth);
  }
```

Fig. 2. Using AutoValue for creating a value class

Auto Generated Code

After defining an abstract value class like the one in Fig. 2, the framework generates all boilerplate for it to behave like the normal value class shown in Fig. 1. The framework generates the equals, hashCode and toString member methods. The framework also generates accessor methods for each field in the value class.

Nullable Properties

By default, field values can not be null. The framework has checks to ensure this. However, if null properties should be allowed, the abstract class can simply mark the accessor method and the corresponding parameter to the create method as @Nullable.

Immutability

AutoValue objects are immutable. There is no way to change the set values of fields after object creation. However, if the fields themselves are mutable collections, they can be altered. It is suggested to use immutable collections wherever possible. Google's Guava core libraries offer immutable versions of commonly used collection types like List, Set, Map, etc.

Primitive Arrays as Fields

The use of primitive arrays as member fields in AutoValue classes is fully supported. The auto generated code also ensures equality checks work correctly by comparing the values inside arrays. However, arrays of generic Object type are not supported.

Memoization

In some cases, a value class may have a property derived from the rest of its member fields. It may be a large amount of work to compute this property. It would be valuable to cache the value of such properties so that they do not need to be computed more than once. AutoValue classes support case. Simply annotating any method in the abstract class definition with the @Memoized annotation achieves caching of the return value generated by the method. Any subsequent calls to this method return the cached value. To qualify for memoization, the method should be non-abstract, have no parameters and should not return void.

Builder Pattern Support

When the number of member fields in a value class becomes very high, the static factory method create has to accept a lot of parameters. The caller of the factory method has to specify too many parameters causing the code to be harder to read and error prone since parameters need to passed in the right order. If the types of parameters are the same, then specifying parameters in the wrong order wouldn't be detected as an error by the compiler but will lead to runtime errors/bugs. In addition, some parameters may be optional for which null values needs to be explicitly passed in. A solution to this problem is the builder pattern which allows setting parameters in a declarative way.

The builder pattern support also allows setting default values for fields, so the client code can only set necessary parameters.

Records

Java added native support for value classes in starting from Java SE 16. These classes are called Record classes. Records have most of features of AutoValue classes while being better integrated with the Java language.

Class Definition

Defining a record class is very simple. The definition is similar to a constructor of a normal class. Member fields for each constructor parameter are automatically generated for the class. Accessor methods also generated for accessing each field. Also similar to AutoValue, equals, hashCode and toString are automatically generated.

```
record Cuboid(int length, int width, int depth) {}
```

Fig. 3. Record class definition

Constructors

A canonical constructor is automatically generated for each Record class. The canonical constructor can also be explicitly defined if more control is needed.

In addition, custom constructors can also be defined. Simply adding a new constructor to the class definition works as expected. All other constructors must invoke the canonical constructor of a class.

Auto Generated Methods

Similar to how the canonical constructor can be explicitly defined, accessors can also be explicitly defined. The equals, hashCode and toString methods can also be explicitly defined if needed.

Features

Record classes are similar to normal classes. However, they are implicitly final. Record classes cannot be extended. Apart from this restriction, Record classes are very similar to normal classes. Additional member functions can also be added inside them. However non-static member fields cannot be added (apart from ones defined in the constructor). Any member fields added outside of the constructor should be static. In the same vein, instance initializers are not allowed. Record classes can be defined using generic types too. Annotations can be added to member fields and functions. Record classes can also implement interfaces.

Comparison Between Autovalue and Records

Between AutoValue and Record classes, the general preference should be to use Record classes wherever possible since they are a language feature. However, there may be some scenarios where AutoValue classes are still preferable.

Java Version

For creating value classes in Java 15 or earlier, AutoValue is the only option. AutoValue on the other hand, is API-invisible. This means that for the user of an AutoValue class there is no difference from a normal class. This means it can be used in any version of Java.

Static Factory Method

Static factory methods are preferred over exposing constructors because they are much more efficient and less prone to error. Record classes have a public constructor forcing the constructor to be exposed. This could lead to less-than-ideal coding practices.

Support for Primitive Arrays

As discussed earlier, AutoValue has support for primitive arrays. The implementation of equals and hashCode account for the special case around arrays.

Caching Derived properties

Record classes cannot have member fields so it is hard to cache a derived property. This is very easy to do in AutoValue using memoization.

Extensions

AutoValue has support for extensions. This allows adding custom functionality to code generation behavior. There are several built in extensions supporting high level features which are missing in Records. One such feature is the support for memoization.

Conclusion

Value classes are an important part of software programming. These classes must be implemented in optimal ways that reduce toil due to boilerplate and errors. So it is a great idea to use either AutoValue or Record classes for defining value classes. Although there may preferences to use one solution over the other, but at the end of the day, either one works to eliminating toil from writing and maintaining value classes.

References

[1] Joshua Bloch, "Effective Java, 3rd Edition (Dec 2017)," https://books.google.com/books?id=auW80AEACAAJ

[2] Bhaskar Ghosh, "Value-Based Classes in Java (Jun 2024)," https://www.baeldung.com/java-value-based-classes

[3] Éamonn McManus, Kevin Bourrillion, "AutoValue (Jan 2024),"

https://github.com/google/auto/blob/main/value/userguide/index.md
[4] "Record Classes (March 2024)," https://docs.oracle.com/en/java/javase/22/language/records.html
[5] Gavin Bierman, "JEP 395: Records (Jun 2020)," https://openjdk.org/jeps/395