JAVA 15

Diving into Java records: Serialization, marshaling, and bean state validation

Existing frameworks and libraries that access instance variables through getters and setters won't work with records. Here's what to do.

by Frank Kiwy

November 6, 2020

Download a PDF of this article

Records were first introduced in Java 14 as a preview feature. Recently, there has been a second preview with the arrival of Java 15. Record classes are therefore not yet a regular part of the JDK and they are still subject to change.

Java records were introduced in *Java Magazine* by Ben Evans in his article "Records come to Java." If you are new to records, you can get a quick overview of what records are in the "Java language updates: Record classes" documentation.

In brief, the main goal of record classes is to model plain data aggregates with less ceremony than normal classes. A record class declares a sequence of fields, and may also declare methods. The appropriate `constructor`, `accessor`, `equals`, `hashCode`, and `toString` methods are created automatically. The fields are final because the class is intended to serve as a simple data carrier.

A record class declaration consists of a name, a header (which lists the fields of the class, known as its components), and a body. The following is an example of a record declaration:

```
record RectangleRecord(double length, double
}
```

In this article, I will focus on serialization and deserialization, marshaling and unmarshaling, and state validation of records. But first, take a look at the class members of a record using Java's Reflection API.

Introspection

With the introduction of records to Java, two new methods have been added to `java.lang.Class`:

- `isRecord()`, which is similar to `isEnum()` except that it returns `true` if the class was declared as a record
- `getRecordComponents()`, which returns an array of `java.lang.reflect.RecordComponent` objects corresponding to the record components

I'll use the latter with the record class declared above to get its components:

```
System.out.println("Record components:");
Arrays.asList(RectangleRecord.class.getRecord
        .forEach(System.out::println);
```

Here's the output:

```
Record components:
double length
double width
```

As you can see, the components are the variables (type and name pairs) specified in the header of the record declaration. Now, look at the record fields that are derived from the components:

```
System.out.println("Record fields:");
Arrays.asList(RectangleRecord.class.getDeclar
        .forEach(System.out::println);
```

The following is the output:

```
Record fields:
private final double record.test.RectangleRec
private final double record.test.RectangleRec
```

Note that the fields are generated by the compiler with the `private` and `final` modifiers. The field accessors and the constructor parameters are also derived from the record components, for example:

```
System.out.println("Field accessors:");
Arrays.asList(RectangleRecord.class.getDeclar
        .filter(m -> Arrays.stream(RectangleR
        .forEach(System.out::println);

System.out.println("Constructor parameters:")
Arrays.asList(RectangleRecord.class.getDeclar
        .forEach(c -> Arrays.asList(c.getPara
        .forEach(System.out::println));
```

Here's the output:

```
Field accessors:
public double record.test.RectangleRecord.len
public double record.test.RectangleRecord.wid
Constructor parameters:
double length
double width
```

Notice that the name of the field accessors does not start with `get` and, therefore, does not conform to the JavaBeans conventions.

(As Brian Goetz wrote in an online thread, ". . . the language has a responsibility to look forward as well as backward, and balance the needs of existing code with the needs of new code. Taking a bad library naming convention and burning it into the language forever would have been the worse choice." —Ed)

You're probably not surprised to not see any methods for setting the contents of a field, because records are supposed to be immutable.

Record components can also be annotated in the same way you would do for constructor or method parameters. For this purpose, I've created a simple annotation such as the following one:

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
}
```

Be sure to set the retention policy to `RUNTIME`; otherwise, the annotation is discarded by the compiler and will not be present at runtime. So, this is the modified record declaration with annotated components:

```
record Rectangle(@MyAnnotation double length,
}
```

The next step is to retrieve the annotation on the record components via reflection, for example:

```
System.out.println("Record component annotati
Arrays.asList(RectangleRecord.class.getRecord
        .forEach(c -> Arrays.asList(c.getDecl
        .forEach(System.out::println));
```

The following is the output:

```
Record component annotations:
@record.test.MyAnnotation()
@record.test.MyAnnotation()
```

As expected, the annotation is present on both components specified in the header of the record.

For records, however, the annotations that you add to the components are also propagated to the derived fields, accessors, and constructor parameters. I will quickly verify this by printing out the annotations of the component-derived artifacts:

Here are annotations on record fields:

```
System.out.println("Record field annotations:
Arrays.asList(RectangleRecord.class.getDeclar
        .forEach(f -> Arrays.asList(f.getDecl
        .forEach(System.out::println));
```

And here is the output:

```
Record field annotations:
@record.test.MyAnnotation()
@record.test.MyAnnotation()
```

Here are annotations on field accessors:

```
System.out.println("Field accessor annotation
Arrays.asList(RectangleRecord.class.getDeclar
        .filter(m -> Arrays.stream(RectangleR
        .forEach(m -> Arrays.asList(m.getDecl
        .forEach(System.out::println));
```

And here is the output:

```
Field accessor annotations:
@record.test.MyAnnotation()
@record.test.MyAnnotation()
```

Finally, here are annotations on record constructor parameters:

```
System.out.println("Constructor parameter ann
Arrays.asList(RectangleRecord.class.getDeclar
        .forEach(c -> Arrays.asList(c.getPara
        .forEach(p -> Arrays.asList(p.getDecl
        .forEach(System.out::println)));
```

And the following is the output:

```
Constructor parameter annotations:
@record.test.MyAnnotation()
@record.test.MyAnnotation()
```

As seen above, if you put an annotation on a record component, it will be automatically propagated to the derived artifacts. However, this behavior is not always desirable, because you might want the annotation to be present only on record fields, for instance. That's why you can change this behavior by specifying the target of an annotation.

For example, if you want an annotation to be present only on the record fields, you would have to add a `Target` annotation with a parameter of `ElementType.FIELD`:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
}
```

Rerunning the above code yields this output:

```
Record component annotations:
Record field annotations:
@record.test.MyAnnotation()
@record.test.MyAnnotation()
Field accessor annotations:
Constructor parameter annotations:
```

As you can see, the annotation is now present only on the record fields. In the same way, you can state that the annotation should be present only on the accessors (

`ElementType.METHOD`), or the constructor parameters (`ElementType.PARAMETER`), or any combination of those two and the record fields.

Be aware that in any of these cases, you must put the annotation always on the record components, because the fields, accessors, and constructor parameters simply don't exist in a record declaration. Those are generated and annotated (according to the element types specified in the annotation declaration) by the compiler and, thus, are present only in the compiled record class.

Serialization and deserialization

Because they are ordinary classes, records can also be serialized and deserialized. The only thing you need to do is to add the `java.io.Serializable` interface to the record's header, for example:

```
record RectangleRecord(double length, double
}
```

Here's the code to serialize a record:

```
private static final List<RectangleRecord> SA
        new RectangleRecord(1, 5),
        new RectangleRecord(2, 4),
        new RectangleRecord(3, 3),
        new RectangleRecord(4, 2),
        new RectangleRecord(5, 1)
);

try (
        var fos = new FileOutputStream("C:/Te
        var oos = new ObjectOutputStream(fos)
    oos.writeObject(SAMPLE_RECORDS);
}
```

And the following code can be used to deserialize a record:

```
try (
        var fis = new FileInputStream("C:/Tem
        var ois = new ObjectInputStream(fis))
    List<RectangleRecord> records = (List<Rec
    records.forEach(System.out::println);
    assertEquals(SAMPLE_RECORDS, records);
}
```

This is the output:

```
RectangleRecord[length=1.0, width=5.0]
RectangleRecord[length=2.0, width=4.0]
RectangleRecord[length=3.0, width=3.0]
```

```
RectangleRecord[length=4.0, width=2.0]
RectangleRecord[length=5.0, width=1.0]
```

However, there's one major difference compared to ordinary classes: When a record is deserialized, its fields are set, via the record constructor, to the values deserialized from the stream. By contrast, a normal class is first instantiated by invoking the no-argument constructor, and then its fields are set via reflection to the values deserialized from the stream.

Thus, records are deserialized using their constructor. This behavior allows you to add invariants to the constructor to check the validity of the deserialized data. Since this is not possible with normal classes, there's always a certain risk of deserializing bad or even hazardous data, which should not be underestimated, especially if the data comes from external sources.

```
import java.io.Serializable;
import java.lang.IllegalArgumentException;
import java.lang.StringBuilder;

public record RectangleRecord(double length,

    public RectangleRecord {
        StringBuilder builder = new StringBui
        if (length <= 0) {
            builder.append("\nLength must be
        }
        if (width <= 0) {
            builder.append("\nWidth must be g
        }
        if (builder.length() > 0) {
            throw new IllegalArgumentExceptio
        }
    }

}
```

Note that this code is using the record's compact constructor here, so there's no need to specify the parameters or to set the record fields explicitly. If you now deserialize the previously serialized records, every single instance is supposed to have a valid state; otherwise, an `IllegalArgumentException` is thrown by the record constructor.

You can verify this by modifying the serialized data of just one record in such a way that it doesn't conform to the validation logic anymore: `RectangleRecord[length=0.0, width=-5.0]`.

If you now execute the deserialization code from above, you'll get the expected

```
IllegalArgumentException:
java.lang.IllegalArgumentException:
```

```
Length must be greater than zero: 0.0
Width must be greater than zero: -5.0
  at record.test.RectangleRecord.<init>(Recta
  at java.base/java.io.ObjectInputStream.read
```

If you tried the same process with a normal class, no exception would occur, since the class's constructor wouldn't be called. The object would be deserialized with the erroneous data, without anyone noticing.

Look at the following `RectangleClass`, which is the counterpart of the `RectangleRecord`:

```
import java.io.Serializable;
import java.util.Objects;

public class RectangleClass implements Serial

    private final double width;
    private final double length;

    public RectangleClass(double width, doubl
        StringBuilder builder = new StringBui
        if (length <= 0) {
            builder.append("\nLength must be
        }
        if (width <= 0) {
            builder.append("\nWidth must be g
        }
        if (builder.length() > 0) {
            throw new IllegalArgumentExceptio
        }
        this.width = width;
        this.length = length;
    }

    @Override
    public String toString() {
        return "RectangleClass[" + "width=" +
    }

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

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        RectangleClass other = (RectangleClas
        return Objects.equals(length, other.l
    }

    public double width() {
        return width;
```

```
    }

    public double length() {
        return length;
    }

}
```

Although the constructor of the `RectangleClass` contains the same validation logic as the constructor of the `RectangleRecord`, it is not called during the deserialization process and, therefore, cannot prevent the creation of objects with invalid state.

Marshaling and unmarshaling

Just like normal classes, records can also be unmarshaled from and marshaled to a format of your choice, such as JSON, XML, or CSV. If you'd like to use an existing library to do so, be aware that it has to access the class fields via the `Field.set(Object obj, Object value)` method and not via the getter and setter methods, because records don't have those methods.

However, you should know about some restrictions. In JDK 15's second preview of Java records, a record's field can no longer be accessed via the `Field.set(Object obj, Object value)` method (which was possible in JDK 14).

The reason for this restriction is to ensure the immutability of records by preventing this kind of backdoor manipulation by libraries. However, most of the current libraries aren't aware of records yet. The libraries therefore treat records as ordinary classes and try to set the field values via the `Field.set(Object obj, Object value)` method. That's not going to work.

Here is an example that uses the popular Gson library to demonstrate the above restriction. With this library, marshaling to JSON should work without any problem because Gson reads the record data using the `Field.get(Object obj)` method:

```
private static final List<RectangleRecord> SA
        new RectangleRecord(1, 5),
        new RectangleRecord(2, 4),
        new RectangleRecord(3, 3),
        new RectangleRecord(4, 2),
        new RectangleRecord(5, 1)
);

try (Writer writer = new FileWriter("C:/Temp/
    new Gson().toJson(SAMPLE_RECORDS, writer)
}
```

And here is the file output:

```
[{"length":1.0,"width":5.0},{"length":2.0,"wi
```

But a problem will occur during the unmarshaling process in which Gson tries to set the field values using the `Field.set(Object obj, Object value)` method:

```
try (Reader reader = new FileReader("C:/Temp/
    List<RectangleRecord> records = new Gson(
    records.forEach(System.out::println);
}
```

The output:

```
java.lang.IllegalAccessException: Can not set
  at java.base/jdk.internal.reflect.UnsafeFie
  at java.base/jdk.internal.reflect.UnsafeFie
  at java.base/jdk.internal.reflect.UnsafeQua
  at java.base/java.lang.reflect.Field.set(Fi
```

Note that write access to the `RectangleRecord.length` field has been prevented by throwing a `java.lang.IllegalAccessException`. This means that the current libraries will need to be changed to take this restriction into account when dealing with records.

At the present time, the only way to set the field values of a record is by using its constructor. And if the constructor arguments are all immutable themselves (for example, when using primitive data types), it will indeed become very hard to change a record's state. Fortunately, this restriction also helps ensure consistent state validation of records, as discussed in the earlier section about deserialization.

If you currently have to unmarshal records from JSON or any other format, you'll probably have to write your own unmarshaler. Most libraries won't support explicit marshaling or unmarshaling for records until they have become a regular Java feature.

As long as they're not, they are still subject to change. Record field access has been restricted in JDK 15 by no longer allowing the fields to be changed via reflection, something that was still possible in JDK 14 (the first preview of records). That's a change in behavior that should not be neglected—especially not by library designers—as everyone looks forward to JDK 16.

Bean validation

You may think that records can't be subject to the bean validation specification (also known as JSR 303) because they

do not adhere to the JavaBeans standard. That's only partly true. A record's state cannot be validated through its getters or setters, because records don't have any getters or setters. However, a record's state can very well be validated via its constructor parameters or its fields.

The Bean Validation API defines a way for expressing and validating constraints using Java annotations. Because these annotations are reusable, they help to avoid code duplication and, thus, contribute to more-concise and less error-prone code. By putting constraint annotations on the components of a record, you can enforce constraint validation and guarantee that a record's state is always valid. Since records are immutable, you need to validate the constraints only once when you create a record instance. If no constraints are violated, the created instance always meets its invariants.

The following example shows how a record's state can be validated. To do so, I'm using the bean validation reference implementation, which is the Hibernate Validator.

But first, I'll add the necessary dependencies with the help of a favorite build tool:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifact
    <version>6.1.5.Final</version>
</dependency>
<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.el</artifactId>
    <version>3.0.0</version>
</dependency>
```

Note that the Hibernate Validator also requires an implementation of the Expression Language to evaluate dynamic expressions in constraint violation messages.

Now, I'll add some validation constraints to the `RectangleRecord` by the means of the `@javax.validation.constraints.Positive` annotation, which checks whether the element is strictly positive (zero values are considered invalid).

```
import javax.validation.constraints.Positive;

public record RectangleRecord(
    @Positive(message = "Length is ${validate
    @Positive(message = "Width is ${validated
) {}
```

To be able to validate the state of a record, you need an instance of `javax.validation.Validator`. But to get a `Validator`

instance, you first have to create a `ValidatorFactory`, for example:

```
ValidatorFactory factory = Validation.buildDe
Validator validator = factory.getValidator();
```

Now you can validate the state of a record instance as follows:

```
RectangleRecord rectangle = new RectangleReco
Set<ConstraintViolation<RectangleRecord>> con
constraintViolations.stream().map(ConstraintV
```

Here's the output:

```
Length is 0.0 but must be greater than zero.
Width is -5.0 but must be greater than zero.
```

The previous example demonstrates that record classes can be validated like normal classes using the Bean Validation API. However, since records do not conform to JavaBeans conventions, their state cannot be validated using getters or setters, for instance.

Wouldn't it be better to check the validity of an object's state during its construction process and, thus, avoid the creation of an instance with incorrect data? Well, this is possible by calling the constraint validation logic in the record's constructor itself.

In order not to have to add the above validation code to every single record constructor, I am going to implement it by using an interface. Because records are final, they cannot extend any other record class to inherit its methods. But a similar behavior can be achieved by declaring a `default` method in an interface, for example:

```
import java.lang.reflect.Constructor;
import java.util.Set;
import java.util.stream.Collectors;
import javax.validation.ConstraintViolation;
import javax.validation.ConstraintViolationEx
import javax.validation.Validator;

public interface Validatable {

    default void validate(Object... args) {
        Validator validator = ValidatorProvid
        Constructor constructor = getClass().
        Set<ConstraintViolation<?>> violation
                .validateConstructorParameter
        if (!violations.isEmpty()) {
            String message = violations.strea
                    .map(ConstraintViolation:
                    .collect(Collectors.joini
            throw new ConstraintViolationExce
```

```
        }
    }

}
```

The following class provides the required `Validator` instance:

```
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

public class ValidatorProvider {

    private static final Validator VALIDATOR;

    static {
        ValidatorFactory factory = Validation
        VALIDATOR = factory.getValidator();
    }

    public static Validator getValidator() {
        return VALIDATOR;
    }

}
```

Now, everything's in place to call the interface's `validate` method in my record constructor. To do so, I have to specify an explicit constructor, which allows me to call the `validate` method:

```
import javax.validation.constraints.Positive;

public record RectangleRecord(double length,

    public RectangleRecord (
            @Positive(message = "Length is ${
            @Positive(message = "Width is ${v
        ) {
        validate(length, width);
        this.length = length;
        this.width = width;
    }

}
```

Note that when you provide an explicit constructor, you have to annotate the constructor parameters and not the components of the record. You have previously seen that the annotations added to the components are also propagated to the derived fields, accessors, and constructor parameters. Regarding the constructor parameters, this is true only as long as you do not provide an explicit constructor.

Now, I'll try to create a `RectangleRecord` instance with an invalid length and width:

```
RectangleRecord rectangle = new RectangleReco
```

Here's the output:

```
javax.validation.ConstraintViolationException
Length is 0.0 but must be greater than zero.
Width is -5.0 but must be greater than zero.
  at record.test.Validatable.validate(Validat
  at record.test.RectangleRecord.<init>(Recta
```

So, with the validation logic called already at instantiation time (in the record constructor), you can prevent the creation of an object with invalid data. In the first bean validation example from above, you first had to create an object with invalid state before you were able to validate it. But that's exactly what you want to *avoid*: creating records with invalid state.

However, by providing an explicit canonical constructor, you also have to explicitly specify all the constructor parameters and set all the record field values manually. But isn't that again quite a lot of clutter that you are trying to avoid when using records? In the following section, I'm going to show how you can omit an explicit constructor declaration and still get the record's data validated during the instantiation process.

Byte Buddy

Byte Buddy is a library for creating and modifying Java classes during the runtime of Java applications without the need of a compiler. Unlike the code generation utilities included in the JDK (such as the Java Instrumentation API), Byte Buddy allows you to create arbitrary classes, and it does not require the implementation of any interface to create runtime proxies.

In addition, it offers a convenient API. Using the API, you can change classes either manually using a Java agent or during a build. You can use the library to manipulate existing classes, create new classes on demand, or intercept method calls, for instance. Using Byte Buddy does not require you to have an understanding of Java bytecode or the class file format. However, you can define custom bytecode, if needed.

The API was designed to be nonintrusive, so Byte Buddy does not leave any traces in class files after the code manipulation has taken place. That's why the generated classes do not require Byte Buddy on the classpath.

Byte Buddy is a lightweight library that depends only on the visitor API of the ASM Java bytecode parser library, so it offers excellent runtime performance.

What I am interested in here is code manipulation at build time, which can be achieved easily by using a dedicated Maven plugin

that ships with the Byte Buddy library.

As you probably know, a Maven build lifecycle consists of phases. One of these phases is the so-called `compile` phase after which Byte Buddy plugs in and changes the Java bytecode according to your instructions. Hence, there's no code manipulation at runtime that could affect runtime performance.

I'll start by adding the required dependencies for the Byte Buddy library:

```
<dependency>
    <groupId>net.bytebuddy</groupId>
    <artifactId>byte-buddy</artifactId>
    <version>1.10.14</version>
</dependency>
```

The following XML adds the Byte Buddy Maven plugin to the build lifecycle:

```
<plugin>
    <groupId>net.bytebuddy</groupId>
    <artifactId>byte-buddy-maven-plugin</arti
    <version>1.10.14</version>
    <executions>
        <execution>
            <goals>
                <goal>transform</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <transformations>
            <transformation>
                <plugin>
                    record.test.RecordValidat
                </plugin>
            </transformation>
        </transformations>
    </configuration>
</plugin>
```

The Byte Buddy Maven plugin uses a custom class called `RecordValidationPlugin` that implements the `net.bytebuddy.build.Plugin` interface, for example:

```
import java.io.IOException;
import javax.validation.Constraint;

import static net.bytebuddy.matcher.ElementMa
import static net.bytebuddy.matcher.ElementMa

import net.bytebuddy.build.Plugin;
import net.bytebuddy.description.method.Metho
import net.bytebuddy.description.type.TypeDes
import net.bytebuddy.dynamic.ClassFileLocator
import net.bytebuddy.dynamic.DynamicType.Buil
```

```
import net.bytebuddy.dynamic.scaffold.TypeVal
import net.bytebuddy.implementation.MethodDel
import net.bytebuddy.implementation.SuperMeth

public class RecordValidationPlugin implement

    @Override
    public boolean matches(TypeDescription ta
        return target.isRecord() && target.ge
                .stream()
                .anyMatch(m -> m.isConstructo
    }

    @Override
    public Builder<?> apply(Builder<?> builde
        try {
            builder = new ByteBuddy().with(Ty
        } catch (ClassNotFoundException ex) {
            throw new RuntimeException(ex);
        }
        return builder.constructor(this::hasC
                .intercept(SuperMethodCall.IN
    }

    private boolean hasConstrainedParameters(
        return m.getParameters()
                .asDefined()
                .stream()
                .anyMatch(p -> !p.getDeclared
                .asTypeList()
                .filter(hasAnnotation(annotat
                .isEmpty());
    }

    @Override
    public void close() throws IOException {
    }

}
```

The interface has three methods: `matches`, `apply`, and `close`. I don't need to implement the last one.

The first method is used by Byte Buddy to find all the classes whose code I want to change. I need only the record classes that have a constructor with constrained parameters (having bean validation annotations). This is where the new method `Class.isRecord()` comes into play.

The second method applies the changes to the bytecode generated during the `compile` phase. It adds to those record constructors that have constrained parameters a call to a method in a custom class called `RecordValidationInterceptor`.

Also, note that I have to use a custom `Builder` instance as follows, because Java records are still a preview feature and, therefore, type validation needs to be disabled:

```
builder = new ByteBuddy().with(TypeValidation
```

And here's the code for the `RecordValidationInterceptor`:

```
import java.lang.reflect.Constructor;
import java.util.Set;
import java.util.stream.Collectors;
import javax.validation.ConstraintViolation;
import javax.validation.ConstraintViolationEx
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;
import net.bytebuddy.implementation.bind.anno
import net.bytebuddy.implementation.bind.anno

public class RecordValidationInterceptor {

    private static final Validator VALIDATOR;

    static {
        ValidatorFactory factory = Validation
        VALIDATOR = factory.getValidator();
    }

    public static <T> void validate(@Origin C
        Set<ConstraintViolation<T>> violation
                .validateConstructorParameter
        if (!violations.isEmpty()) {
            String message = violations.strea
                    .map(ConstraintViolation:
                    .collect(Collectors.joini
            throw new ConstraintViolationExce
        }
    }

}
```

As a result of the code manipulation, the `validate` method gets called from the record constructor and passes a `Constructor` object along with the according parameter values to the bean validator instance.

You can give the method any name; Byte Buddy will identify it with the help of its own annotations such as `@Origin` or `@AllArguments`.

Now I'll build the project using the previously declared `RectangleRecord` with validation constraints added to the components, for example:

```
import javax.validation.constraints.Positive;

public record RectangleRecord(
    @Positive(message = "Length is ${validate
    @Positive(message = "Width is ${validated
) {}
```

After the build has completed, you can look at the resulting bytecode. To do so, execute the following command (allowing you to disassemble a class file) from the command line:

```
javap -c RectangleRecord
```

In the following, I show only the constructor bytecode:

```
public record.test.RectangleRecord(double, do
    Code:
       0: aload_0
       1: dload_1
       2: dload_3
       3: aconst_null
       4: invokespecial #75                 /
       7: getstatic     #79                 /
      10: iconst_2
      11: anewarray     #81                 /
      14: dup
      15: iconst_0
      16: dload_1
      17: invokestatic  #87                 /
      20: aastore
      21: dup
      22: iconst_1
      23: dload_3
      24: invokestatic  #87                 /
      27: aastore
      28: invokestatic  #93                 /
      31: return
```

Notice the last instruction just before the `return` statement. That's where the method `RecordValidationInterceptor.validate` is called.

Now I'll test the code refactored by Byte Buddy:

```
RectangleRecord rectangle = new RectangleReco
```

Here's the output:

```
javax.validation.ConstraintViolationException
Length is 0.0 but must be greater than zero.
Width is -5.0 but must be greater than zero.
  at csv.to.records.RecordValidationIntercept
  at record.test.RectangleRecord.<init>(Recta
```

As you can see, the creation of a `RectangleRecord` instance with invalid data has been avoided just by using regular bean validation constraints on record components. The use of the Byte Buddy plugin helps you to enforce Java record invariants through the means of bean validation.

Conclusion

Java records behave in many ways as normal Java classes, but there are some differences to take into account. One is that records don't conform to JavaBeans conventions, because they don't have getters and setters, for instance. That's why using existing frameworks or libraries, and accessing instance variables through getters and setters, won't work with records.

Another difference is that the fields of a record can be set only via its constructor and they are, therefore, de facto final, which makes it easy to validate the record's state by using either an explicit constructor with the corresponding validation logic or constraint annotations. In addition to the fact that records can be declared with less ceremony than normal classes, convenient state validation is one of their major advantages.

I'm looking forward to when records will become a regular Java feature, hopefully in Java 16. In the meantime, I will keep enjoying their second preview with Java 15.

Dig deeper

- Records come to Java
- Java language updates: Record classes
- Record class documentation in Java 15
- Data classes and sealed types for Java
- JEP 384: Records (second preview) in Java 15
- JEP 395: Records in Java 16

Frank Kiwy

Frank Kiwy is a senior software developer and project leader who works for a government IT center in Europe. His focus is on Java SE, Java EE, and web technologies. Kiwy is also interested in software architecture and is committed to continuous integration and delivery. He is currently involved in implementing the European Union's Common Agricultural Policy, where he's in charge of several projects. When programming, he values well-designed software with clear and easy-to-understand APIs.