# Java Magazine

Coding, Java 16, Tools

# Simpler object and data serialization using Java records

Julia Boes | January 21, 2022

Chris Hegarty | January 21, 2022

## Learn how you can leverage the design of Java's records to improve Java serialization.

Record classes enhance Java's ability to model plain-data aggregates without a lot of coding verbosity or, in the phrase used in JEP 395, without too much ceremony. A record class declares some immutable state and commits to an API that matches that state. This means that record classes give up a freedom that classes usually enjoy—the ability to decouple their API from their internal representation—but in return, record classes become significantly more concise.

Record classes were a preview feature in Java 14 and Java 15 and became final in Java 16 in JEP 395. Here is a record class declared in the JDK's `jshell` tool.

```
jshell> record Point (int x, int y) { }
| created record Point
```

The state of `Point` consists of two components, `x` and `y`. These components are immutable and can be accessed only via accessor methods `x()` and `y()`, which are automatically added to the `Point` class during compilation. Also added during compilation is a *canonical constructor* for initializing the

components. For the `Point` record class, it is equivalent to the following:

```
public Point(int x, int y) {
 this.x = x;
 this.y = y;
}
```

Unlike the no-argument default constructor added to normal classes, the canonical constructor of a record class has the same signature as the state. (If an object needs mutable state, or state that is unknown when the object is created, a record class is *not* the right choice; you should declare a normal class instead.)

Here is `Point` being instantiated and used. In terms of terminology, say that `p`, the instance of `Point`, is a record.

```
jshell> Point p = new Point(5, 10)
p ==> Point[x=5, y=10]

jshell> System.out.println("value of x: " + p.x())
value of x: 5
```

Copy code snippet

Taken together, the elements of a record class form a succinct protocol for you to rely on: The elements include a concise description of the state, a canonical constructor to initialize the state, and controlled access to the state. This design has many benefits, such as for object serialization.

## What is object serialization?

Serialization is the process of converting an object into a format that can be stored on disk or transmitted over the network (also termed *serialized* or *marshaled*) and from which the object can later be reconstituted (*deserialized* or *unmarshaled*).

Serialization provides the mechanics for extracting an object's state and translating it to a persistent format, as well as the means for reconstructing an object with equivalent state from that format. Given their nature as plain data carriers, records are well suited for this use case.

The idea of serialization is powerful, and many frameworks have implemented it, one of them being Java Object Serialization in the JDK, which we'll refer to simply as *Java Serialization*.

In Java Serialization, any class that implements the `java.io.Serializable` interface is serializable. That's suspiciously simple, right? However, the interface has no members and serves only to mark a class as serializable.

During serialization, the state of all nontransient fields is scraped (even for private fields) and written to

the serial byte stream. During deserialization, a superclass no-argument constructor is called to create an object before its fields are populated with the state read from the serial byte stream. The format of the serial byte stream (the *serialized form*) is chosen by Java Serialization unless you use the special methods `writeObject` and `readObject` to specify a custom format.

## Problems with Java Serialization

It's not news that Java Serialization has flaws, and Brian Goetz's June 2019 blog post, "Towards better serialization," provides a summary of the problems.

The core of the problem is that Java Serialization was not designed as part of Java's object model. This means that Java Serialization works with objects using backdoor techniques such as reflection, rather than relying on the API provided by an object's class. For example, it is possible to create a new deserialized object without invoking one of its constructors, and data read from the serial byte stream is not validated against constructor invariants.

## Serialization with records

With Java Serialization, a record class is made serializable just like a normal class, simply by implementing `java.io.Serializable`.

```
jshell> record Point (int x, int y) implements Serializable { }
|  created record Point
```

Copy code snippet

However, under the hood, Java Serialization treats a record (that is, an instance of a record class) very differently than an instance of a normal class. (This July 2020 blog post by Chris Hegarty and Alex Buckley provides a good comparison.) The design aims to keep things as simple as possible and is based on two properties.

- The serialization of a record is based solely on its state components.
- The deserialization of a record uses only the canonical constructor.

**Important note:** No customization of the serialization process is allowed for records. That's by design: The simplicity of this approach is enabled by, and is a logical continuation of, the semantic constraints placed on records.

Because a record is an immutable data carrier, a record can only ever have one state, which is the value of its components. Therefore, there is no need to allow customization of the serialized form.

Similarly, on the deserialization side, the only way to create a record is through the canonical constructor of its record class, whose parameters are known because they are identical to the state description.

Going back to the sample record class `Point`, the serialization of a `Point` object using Java Serialization looks as follows:

```
jshell> var out = new ObjectOutputStream(new FileOutputStream("serial.data"))
out ==> java.io.ObjectOutputStream@5f184fc6

jshell> out.writeObject(new Point(5, 10));

jshell> var in = new ObjectInputStream(new FileInputStream("serial.data"));
in ==> java.io.ObjectInputStream@504bae78

jshell> in.readObject();
$5 ==> Point[x=5, y=10]
```

Copy code snippet

Under the hood, a serialization framework can use the `x()` and `y()` accessors of `Point` during serialization to extract the state of `p`'s components, which are then written to the serial byte stream. During deserialization, the bytes are read from `serial.data` and the state is passed to the canonical constructor of `Point` to obtain a new record.

Overall, the design of records naturally fits the demands of serialization. The tight coupling of the state and the API facilitates an implementation that is more secure and easier to maintain. Furthermore, the design allows for some interesting efficiencies of the deserialization of records.

## Optimizing record deserialization

For normal classes, Java Serialization relies heavily on reflection to set the private state of a newly deserialized object. However, record classes expose their state and means of reconstruction through a well-specified public API—which Java Serialization leverages.

The constrained nature of record classes drives a re-evaluation of Java Serialization's strategy of reflection.

If, as outlined above, the API of a record class describes the state of a record, and since this state is immutable, the serial byte stream no longer has to be the single source of truth and the serialization framework doesn't need to be the single interpreter of that truth.

Instead, the record class can take control of its serialized form, which can be derived from the components. Once the serialized form is derived, you can generate a matching *instantiator* based on that form ahead of time and store it in the `class` file of the record class.

In this way, control is inverted from Java Serialization (or any other serialization framework) to the record class. The record class now determines its own serialized form, which it can optimize, store, and make

available as required.

This control inversion can enhance record deserialization in several ways, with two interesting areas being class evolution and throughput.

**More freedom to evolve record classes.** The potential for this arises from an existing well-specified feature of record deserialization: default value injection for absent stream fields. When no value is present in the serial byte stream for a particular record component, its default value is passed to the canonical constructor. The following example demonstrates this with an evolved version of the record class `Point`:

```
jshell> record Point (int x, int y, int z) implements Serializable { }
|  created record Point
```

Copy code snippet

After you serialized a `Point` record in the previous example, the `serial.data` file contained a representation of a `Point` with values for `x` and `y` *only*, not for `z`. For reasons of compatibility, however, you might want to be able to deserialize that original serialized object in the context of the new `Point` declaration. Thanks to the default value injection for absent field values, this is possible, and deserialization completes successfully.

```
jshell> var in = new ObjectInputStream(new FileInputStream("serial.data"));
in ==> java.io.ObjectInputStream@421faab1

jshell> in.readObject();
$3 ==> Point[x=5, y=10, z=0]
```

Copy code snippet

This feature can be taken advantage of in the context of record serialization. If you inject default values during deserialization, do those default values need to be represented in the serialized form? In this case, a more compact serialized form could still fully capture the state of the record object.

More generally, this feature also helps support record class versioning, and it makes serialization and deserialization overall more resilient to changes in record state across versions. Compared with normal classes, record classes are therefore even more suitable candidates for storing data.

**More throughput when processing records.** The other interesting area for enhancement is throughput during deserialization. Object creation during deserialization usually requires reflective API calls, which are expensive and hard to get right. These two problems can be addressed by making the reflective calls more efficient and by encapsulating the instantiation mechanics in the record class itself.

For this, you can leverage the power of *method handles* combined with *dynamically computed constants*.

The method handle API in `java.lang.invoke` was introduced in Java 7 and offers a set of low-level operations for finding, adapting, combining, and invoking methods/setting fields. A method handle is a typed reference that allows transformations of arguments and return types and can be faster than traditional reflection from Java 1.1, if it's used wisely. In this case, several method handles can be chained together to tailor the creation of records based on the serialized form of their record class.

This method handle chain can be stored as a dynamically computed constant in the `class` file of the record class, which is lazily computed at first invocation.

Dynamically computed constants are amenable to optimizations by the JVM's dynamic compiler, so the instantiation code adds only a small overhead to the footprint of the record class. With this, the record class is now in charge of both its serialized form and its instantiation code, and it no longer relies on other intermediaries or frameworks.

This strategy further improves performance and code reuse. It also reduces the burden on the serialization framework, which can now simply use the deserialization strategy provided by the record class, without writing complex and potentially unsafe mapping mechanisms.

## Conclusion

You have seen how serialization can capitalize on the semantic constraints placed on records by the design of the Java language, and many further potential optimizations can be explored from here. It is evident that putting a record class in charge of its own serialized form allows Java developers to go further with record serialization.

## Dig deeper

- Java records: Serialization, marshaling, and bean state validation
- Records come to Java
- Why is Java making so many things immutable?
- What are they building—and why? Questions for the top Java architects

**Julia Boes**

Julia Boes is an OpenJDK developer in Oracle's Java Platform Group. She holds a master's degree in linguistics and computer science and works in the areas of core libraries, networking, and serialization. Her other passions are the outdoors, gardening, and sports.

**Chris Hegarty**

Chris Hegarty (@chegar999) is a principal engineer at Elastic. He was formerly a consulting member of technical staff and the networking lead of the Java Platform Group at Oracle.

‹ Previous Post

Next Post ›