In this article, I'll introduce the concept of a record in Java. Records are a new form of Java class designed to

- Provide a first-class means for modeling data-only aggregates
- Close a possible gap in Java's type system
- Provide language-level syntax for a common programming pattern
- Reduce class boilerplate

The records feature is under active development and is targeted to appear as a preview feature in Java 14 (which will be released in March). To make the most of this article, you should be fairly knowledgeable about Java programming and curious about how programming languages are designed and implemented.

Let's start by exploring the basic idea of what a Java record is.

**What Is a Java Record?**

One of the most common complaints about Java is that you need to write a lot of code for a class to be useful. Quite often you need to write the following:

- `toString()`
- `hashCode()` and `equals()`
- Getter methods
- A public constructor

For simple domain classes, these methods are usually boring, repetitive, and the kind of thing that could easily be generated mechanically (and IDEs often provide this capability), but as of now, the language itself doesn't provide any way to do this.

This frustrating gap is actually worse when you are reading someone else's code. For example, it might look like the author is using IDE-generated `hashCode()` and `equals()` that handle all the fields of the class, but how can you be sure without checking each line of the implementation? What happens if a field is added during refactoring and the methods are not regenerated?

The goal of records is to extend the Java language syntax and create a way to say that a class is "the fields, just the fields, and nothing but the fields." By you making that statement about a class, the compiler can help by creating all the methods automatically and having all the fields participate in methods such as `hashCode()`.

**An Example Without Records**
In this article, I am going to use foreign exchange (FX) trading as an example domain to explain records. I will show how you can use them to improve your modeling of the domain and get cleaner, less verbose, simpler code as a result.

Note that in a short article, it's impossible to describe a fully realized production-grade trading system, so instead I will focus on a few basic aspects.

Let's consider how to place an order when trading FX. The basic order type might consist of

- The number of units you're buying or selling (in millions of currency units)
- The "side," that is, whether you're buying or selling (often called "bid" and "ask," respectively)
- The currencies you're exchanging (the "currency pair")
- The time at which you placed your order
- How long your order is good for before it times out (the time to live or "TTL")

So, if you have £1 million and want to sell it for US dollars, and you want $1.25 for each £1, in the jargon of FX trading you are "buying 1 million of the GBP/USD rate at $1.25." Traders also speak of when the order was created—usually at the present time, and how long the order is good for, which is often 1 second or less.

In Java, you might declare a domain class such as the following (I'm calling it "Classic" to underscore that you need to do this with a class at present):

```java
public final class FXOrderClassic {
    private final int units;
    private final CurrencyPair pair;
    private final Side side;
    private final double price;
    private final LocalDateTime sentAt;
    private final int ttl;

    public FXOrderClassic(int units,
                          CurrencyPair pair,
                          Side side,
                          double price,
                          LocalDateTime sentAt,
                          int ttl) {
        this.units = units;
        this.pair = pair; // CurrencyPair is a simple enum
        this.side = side; // Side is a simple enum
        this.price = price;
        this.sentAt = sentAt;
        this.ttl = ttl;
    }

    public int units() { return units; }

    public CurrencyPair pair() { return pair; }

    public Side side() { return side; }

    public double price() { return price; }

    public LocalDateTime sentAt() { return sentAt; }

    public int ttl() { return ttl; }

    @Override
```
Then the order can be created like this:

```java
var order = new FXOrderClassic(1,
    CurrencyPair.GBPUSD,
    Side.Bid, 1.25,
    LocalDateTime.now(), 1000);
```

But how much of the code to declare the class is really necessary? In current versions of Java, most developers would probably just declare the fields and then use their IDE to autogenerate all the methods. Let's see how records improve the situation.

As a side note, Java doesn't provide any way to talk about a data aggregate other than by defining a class, so it is clear that any type containing "just the fields" will be a class.

### The Example with Records

The new concept is a **record class** (usually just called a **record**). This is an immutable (in the usual "shallow" Java sense) transparent carrier for a fixed set of values known as the **record components**. Each component gives rise to a final field that holds the provided value and an accessor method to retrieve the value. The field name and the accessor name match the name of the component.

The list of fields provides a **state description** for the record. In a class, there might be no relation between a field `x`, the constructor argument `x`
and the accessor `x()`, but in a record, they are by definition talking about the same thing: A record is its state.

To enable the creation of new instances of record classes, a constructor called the canonical constructor is also generated, which has a parameter list that exactly matches the declared state description.

The Java language (as of a preview feature in Java 14) provides concise syntax for declaring records, in which all the programmer needs to do is to declare the component names and types that make up the record, like this:

```java
public record FXOrder(int units,
                     CurrencyPair pair,
                     Side side,
                     double price,
                     LocalDateTime sentAt,
                     int ttl) {}
```

By writing this record declaration, you are not just saving some typing. You are making a much stronger, semantic statement: that the `FXOrder` type is just the state provided and any instance is just a transparent aggregate of the field values.

One consequence of this is the field names become your API, so it becomes even more important to pick good names. (For example, `Pair` is not a good name for a type because it could refer to a pair of shoes.)

To access the new language features, you need to compile with the preview flag any code that declares records:

```
javac --enable-preview -source 14 FXOrder.java
```

If you now examine the class file with `javap`, you can see that the compiler has autogenerated a bunch of boilerplate code. (I am showing only the methods and their signatures in the decompilation below.)

```java
$ javap FXOrder.class
Compiled from "FXOrder.java"
public final class FXOrder extends java.lang.Record
    public FXOrder(int, CurrencyPair, Side, double,
                   java.time.LocalDateTime, int);
    public java.lang.String toString();
    public final int hashCode();
    public final boolean equals(java.lang.Object);
    public int units();
    public CurrencyPair pair();
    public Side side();
    public double price();
    public java.time.LocalDateTime sentAt();
    public int ttl();
}
```

This looks remarkably like the set of methods in the code for the class-based implementation. In fact, the constructor and accessor methods all behave exactly as before.

**Peculiarities of Records**

However, methods such as `toString()` and `equals()` use an implementation that might be surprising to some developers:

```java
public java.lang.String toString();
Code:
    0:  aload_0
    1:  invokedynamic #51, 0 // InvokeDynamic
    6:  areturn
```

This is because the `toString()` method is defined as an `Object` method, and the implementation is generated by the compiler to simply call `toString()` on this object. This ensures that the `toString()` method is also listed in the class file, even though it is not declared in the record.

The `equals()` method is also implemented using this approach, ensuring that records are compared in a consistent way.
That is, the `toString()` method (as well as `equals()` and `hashCode()`) is implemented using an `invokedynamic`-based mechanism. This is similar to how string concatenation has also been migrated to use `invokedynamic` in recent Java versions.

You can also see that there is a new class, `java.lang.Record`, that will act as the supertype for all record classes. It is abstract and declares `equals()`, `hashCode()`, and `toString()` to be abstract methods.

The `java.lang.Record` class cannot be directly extended, as you can see by trying to compile some code like the following:

```java
public final class FXOrderClassic extends Record {
    private final int units;
    private final CurrencyPair pair;
    private final Side side;
    private final double price;
    private final LocalDateTime sentAt;
    private final int ttl;

    // ... rest of class elided
}
```

The compiler will reject the attempt, as follows:

```bash
$ javac --enable-preview -source 14 FXOrderClassic.java
FXOrderClassic.java:3: error: records cannot directly extend
public final class FXOrderClassic extends Record {
  ^
Note: FXOrderClassic.java uses preview language features;
Note: Recompile with -Xlint:preview for details.
1 error
```

This means that the only way to get a record is to explicitly declare one and have `javac` create the class file. This approach also ensures that all record classes are created as `final`.

A couple of other core Java features also have special characteristics when applied to records.

First, records must obey a special contract regarding the `equals()` method:

If a record `r` has components `c1, c2, ... cn`, then if a record instance is copied as follows:

```java
R copy = new R(r.c1(), r.c2(), ..., r.cn());
```

Then it must be the case that `r.equals(copy)` is true. Note that this invariant is in addition to the usual familiar contract regarding `equals()` and `hashCode()`; it does not replace it.

Secondly, Java serialization of records is different than it is for regular classes. This is a good thing because, as is now widely recognized, the Java serialization mechanism is deeply flawed in the general case. As Brian Goetz, Java language architect, puts it: “Serialization constitutes an invisible but public constructor, and an invisible but public set of accessors for your internal state.”

Fortunately, records are designed to be very simple: They are just transparent carriers for their fields, so there is no need to invoke the weirdness in the detail of the serialization mechanism. Instead, you can always use the public API and canonical constructor to serialize and deserialize records.

In addition, the `serialVersionUID` of a record class is `0L` unless it is explicitly declared. The requirement for matching `serialVersionUID` values is also waived for record classes.
Before going on to the next section, I want to emphasize that there is a new programming pattern and there also is new syntax for a low-boilerplate class declaration, and they are not related to the inline classes feature being developed in Project Valhalla.

**Design-Level Considerations**

Let’s move on to explore some of the design-level aspects of the records feature. To do so, it’s helpful to recall how enums work in Java. An enum in Java is a special form of class that implements a pattern but with minimal syntax overhead; the compiler generates a bunch of code for us.

Similarly, a record in Java is a special form of class that implements a data carrier pattern with minimal syntax. All the boilerplate code that you expect will be autogenerated by the compiler.

However, while the simple concept of a data carrier class that just holds fields makes intuitive sense, what does that really mean in detail?

When records were first being discussed, many different designs were considered, for example:

- Boilerplate reduction of plain old Java objects (POJOs)
- JavaBeans 2.0
- Named tuples
- Product types (a form of algebraic data type)

These possibilities were discussed in some detail by Brian Goetz in his original design sketch. Each design option comes with additional secondary questions that follow from the choice of the design center for records, such as:

- Can Hibernate proxy them?
- Are they fully compatible with classic JavaBeans?
- Are two records that declare the same fields in the same order considered the same type?
- Will they come with pattern matching techniques such as pattern matching and destructuring?

It would be plausible to base the records feature on any one of these approaches, because each has advantages and disadvantages. However, the final design decision is that records are named tuples.

This choice was partially driven by a key design idea in Java’s type system known as nominal typing, which is the idea that every piece of Java storage (variables, fields) has a definite type and that each type has a name that should be meaningful to humans.

Even in the case of anonymous classes, the types still have names; it’s just that the compiler assigns the names and they are not valid names for types in the Java language (but are still OK within the VM), for example:

```java
jshell> var o = new Object() {
...>   public void bar() { System.out.println("ba
...> }
  o ==> $0@37f8bb67
jshell> var o2 = new Object() {
...>   public void bar() { System.out.println("ba
...> }
  o2 ==> $1@31cefde0
jshell> o = o2;
| Error: |
| incompatible types: $1 cannot be converted to $0 |
| o = o2; |
```

Notice that even though the anonymous classes were declared in exactly the same way, the compiler still produced two different anonymous
There are other (non-Java) languages where the overall shape of a class (for example, what fields and methods it has) can be used as the type (rather than an explicit type name); this is called structural typing.

It would have been a major change if records had broken with Java's heritage and brought in structural typing for records. As a result, the "records are nominal tuples" design choice means you should expect that records will work best where you might use tuples in other languages. This includes use cases such as compound map keys or to simulate multi-return from a method. An example compound map key might look like this:

```java
record OrderPartition(CurrencyPair pair, Side side)
```

Incidentally, records will not necessarily work well as a replacement for existing code that currently uses JavaBeans. There are several reasons for this: JavaBeans are mutable, whereas records are not, and they have different conventions for their accessors.

Records do allow some additional flexibility beyond the simple, single-line declaration form because they are genuine classes. Specifically, the developer can define additional methods, constructors, and static fields apart from the autogenerated defaults. However, these capabilities should be used carefully. Remember that the design intent of records is to enable developers to group related fields together as a single immutable data item.

A good rule of thumb is this: The more tempting it is to add additional methods to the basic data carrier (or to make it implement an interface), the more likely it is that a full class should be used rather than a record.

**Compact Constructors**

One important possible exception to this rule is the use of compact constructors, which are described like this in the Java specification:

The intention of a compact constructor declaration is that only validation and/or normalization code need be given in the body of the canonical constructor; the remaining initialization code is supplied by the compiler.

For example, you might want to validate orders to make sure that they don’t attempt to buy or sell negative quantities or set an invalid TTL value:

```java
public record FXOrder(int units,
                     CurrencyPair pair,
                     Side side,
                     double price,
                     LocalDateTime sentAt,
                     int ttl) {

    public FXOrder {
        if (units < 1) {
            throw new IllegalArgumentException(  
                "FXOrder units must be positive");
        }
        if (ttl < 0) {
            throw new IllegalArgumentException(  
                "FXOrder TTL must be positive, or 0");
        }
        if (price <= 0.0) {
            throw new IllegalArgumentException(  
                "FXOrder price must be positive");
        }
    }
}
```
A compact constructor does not cause a separate constructor to be generated by the compiler. Instead, the code that you specify in the compact constructor appears as extra code at the start of the canonical constructor. You do not need to specify the assignment of constructor parameters to fields—that is still generated automatically and appears in the constructor in the usual way.

One advantage that Java records have over the anonymous tuples found in other languages is that the constructor body of a record allows for code to be run when records are created. This allows for validation to occur (and exceptions to be thrown if an invalid state is passed). This would not be possible in purely structural tuples.

**Alternative Constructors**

It is also possible to use some static factory methods within the body of the record, for example, to work around the lack of default parameter values in Java. In the trading example, you might include a static factory like this to declare a quick way to create orders with default parameters:

```java
public static FXOrder of(CurrencyPair pair,
    Side side,
    double price) {
    return new FXOrder(1, pair, side, price,
        LocalDateTime.now(), 1000);
}
```

This could also be declared as an alternative constructor, of course. You should choose which approach makes sense in each circumstance.

One other use for alternative constructors is to create records for use as compound map keys, as in this example:

```java
record OrderPartition(CurrencyPair pair, Side side) {
    public OrderPartition(FXOrder order) {
        this(order.pair(), order.side());
    }
}
```

The type `OrderPartition` can then be easily used as a map key. For instance, you might want to construct an order book for use in the trade-matching engine:

```java
public final class MatchingEngine {
    private final Map<OrderPartition, RankedOrderBook> orderBooks = new TreeMap<>();

    public void addOrder(final FXOrder o) {
        orderBooks.get(new OrderPartition(o)).addAndCheckForCrosses(o.pair());
    }

    public void checkForCrosses(final CurrencyPair p) {
        // Do any buy orders match with sells now?
    }

    // ...
}
```

Then, when a new order is received, the `addOrder()` method extracts the appropriate order partition (consisting of a tuple of the currency pair and buy/sell side) and uses it to add the new order to the appropriate price-ranked order book. The new order might match against existing orders already on the books (which is called “crossing” of orders), so I need to check whether it does in the `checkForCrosses()` method.
Sometimes, you might not want to use the compact constructor and instead have a full, explicit canonical constructor. This signals that you need to do actual work in the constructor—and the number of use cases for this with simple data carrier classes is small. However, for some situations such as the need to make defensive copies of incoming parameters, this option is necessary. As a result, the possibility of an explicit canonical constructor is permitted by the compiler, but think very carefully before making use of it.

**Conclusion**

Records are intended to be simple data carriers, a version of tuples that fits into Java’s established type system in a logical and consistent manner. This will help many applications make domain classes clearer and smaller. It will also help teams eliminate many hand-coded implementations of the underlying pattern and reduce or remove the need for libraries like Lombok.

However, just as for sealed types, some of the most important use cases for records will emerge in the future. Pattern matching and, in particular, *deconstruction patterns* that allow a record to be broken up into its components show great promise and may well change the way that many developers program in Java. The combination of sealed types and records also provides Java with a version of a language feature known as *algebraic data types*.

If you’re familiar with these features from other programming languages, great. If not, don’t worry; they are being designed to fit with the Java language you already know and be easy to start using in your code.

However, you must always remember that until a finalized version of Java is delivered that contains a specific language feature, you should not rely on it. When talking about possible future features, as I have in this article, it should always be understood that the feature is being discussed for exploration purposes only.

---

**Ben Evans**

Ben Evans (@kittyst) is a Java Champion, a tech fellow and founder at jClarity, an organizer for the London Java Community (LJC), and a member of the Java SE/EE Executive Committee. He has written four books on programming, including the recent *Optimizing Java* (O’Reilly).