Using Record to Model Immutable Data

The Java language gives you several ways to create an immutable class. Probably the most straightforward way is to cre

```java
public class Point {
    private final int x;
    private final int y;

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

Now that you have written these elements, you need to add the accessors for your fields. You will also add a `toString()`

If you need to carry the instances of this class from one application to another, either by sending them over a network or t

In the end, your `Point` class may be a hundred lines long, mostly populated with code generated by your IDE, just to mode

Records have been added to the JDK to change this. Records give you all this with a single line of code. All you need to do

Calling Records to the Rescue

Records are here to help you make this code much simpler. Starting with Java SE 14, you can write the following code.

```java
public record Point(int x, int y) {}
```

This single line of code creates the following elements for you.

1. It is an immutable class with two fields: `x` and `y`, of type `int`.
2. It has a canonical constructor, to initialize these two fields.
3. The `toString()`, `equals()` and `hashCode()` methods have been created for you by the compiler with a default behavi
4. It can implement the `Serializable` interface, so that you can send instances of `Point` to other applications over a ne

Records are making the creation of immutable aggregates of data much simpler, without the help of any IDE. It reduces th

The Class of a Record

A record is class declared with the `record` keyword instead of the `class` keyword. Let us declare the following record.

```java
public record Point(int x, int y) {}
```

The class that the compiler creates for you when you create a record is final.

This class extends the `java.lang.Record` class. So your record cannot extend any class.

A record can implement any number of interfaces.

Declaring the Components of a Record

The block that immediately follows the name of the record is `(int x, int y)`. It declares the *components* of the record r

In this example, the compiler creates two private final fields of type `int`: `x` and `y`, corresponding to the two components yo

Along with these fields, the compiler generates one *accessor* for each component. This accessor is a method that has the

```
public int x() {
    return this.x;
}

public int y() {
    return this.y;
}
```

If this implementation works for your application, then you do not need to add anything. You may define your own access

The last elements generated for you by the compiler are overrides of the `toString()`, `equals()` and `hashCode()` methods

Things you Cannot Add to a Record

There are three things that you cannot add to a record:

1. You cannot declare any instance field in a record. You cannot add any instance field that would not correspond to a co
2. You cannot define any field initializer.
3. You cannot add any instance initializer.

You can create static fields with initializers and static initializers.

Constructing a Record with its Canonical Constructor

The compiler also creates a constructor for you, called the *canonical constructor*. This constructor takes the components

There are situations where you need to override this default behavior. Let us examine two use cases:

1. You need to validate the state of your record
2. You need to make a defensive copy of a mutable component.

Using the Compact Constructor

You can use two different syntax to redefine the canonical constructor of a record. You can use a compact constructor or

Suppose you have the following record.

```java
public record Range(int start, int end) {}
```

For a record of that name, one could expect that the `end` is greater that the `start`. You can add a validation rule by writing

```java
public record Range(int start, int end) {

    public Range {
        if (end <= start) {
            throw new IllegalArgumentException("End cannot be lesser than start");
        }
    }
}
```

The compact canonical constructor does not need to declare its block of parameters.

Note that if you choose this syntax, you cannot directly assign the record's fields, for example with `this.start = start`

```java
public Range {
    // set negative start and end to 0
    // by reassigning the compact constructor's
    // implicit parameters
    if (start < 0)
        start = 0;
    if (end < 0)
        end = 0;
}
```

Using the Canonical Constructor

If you prefer the non-compact form, for example because you prefer not to reassign parameters, you can define the canor

```java
public record Range(int start, int end) {

    public Range(int start, int end) {
        if (end <= start) {
            throw new IllegalArgumentException("End cannot be lesser than start");
        }
        if (start < 0) {
            this.start = 0;
        } else {
            this.start = start;
        }
        if (end > 100) {
            this.end = 10;
        } else {
            this.end = end;
        }
    }
}
```

In this case the constructor you write needs to assign values to the fields of your record.

If the components of your record are not immutable, you should consider making defensive copies of them in both the ca

Defining any Constructor

You can also add any constructor to a record, as long as this constructor calls the canonical constructor of your record. T

Let us examine the following `State` record. It is defined on three components:

1. the name of this state
2. the name of the capital of this state
3. a list of city names, that may be empty.

We need to store a defensive copy of the list of cities, to ensure that it will not be modified from the outside of this record

Having a constructor that does not take any city is useful in your application. This can be another constructor, that only ta

Then, instead of passing a list of cities, you can pass the cities as a vararg. To do that, you can create a third constructor,

```java
public record State(String name, String capitalCity, List<String> cities) {

    public State {
        // List.copyOf returns an unmodifiable copy,
        // so the list assigned to `cities` can't change anymore
        cities = List.copyOf(cities);
    }

    public State(String name, String capitalCity) {
        this(name, capitalCity, List.of());
    }

    public State(String name, String capitalCity, String... cities) {
        this(name, capitalCity, List.of(cities));
    }

}
```

Note that the `List.copyOf()` method does not accept null values in the collection it gets as an argument.

Getting the State of a Record

You do not need to add any accessor to a record, because the compiler does that for you. A record has one accessor met

The `Point` record from the first section of this tutorial has two accessor methods: `x()` and `y()` that return the value of the

There are cases where you need to define your own accessors, though. For instance, assume the `State` record from the p

```java
public List<String> cities() {
    return List.copyOf(cities);
}
```

Serializing Records

Records can be serialized and deserialized if your record class implements `Serializable`. There are restrictions though.

1. None of the systems you can use to replace the default serialization process are available for records. Creating a wri
2. Records can be used as proxy objects to serialize other objects. A `readResolve()` method can return a record. Adding
3. Deserializing a record *always* calls the canonical constructor. So all the validation rules you may add in this constructo

This makes records a very good choice for creating data transport objects in your application.

Using Records in a Real Use Case

Records are a versatile concept that you can use in many contexts.

The first one is to carry data in the object model of your application. You can use records for what they have been designe

Because you can declare local records, you can also use them to improve the readability of your code.

Let us consider the following use case. You have two entities modeled as records: `City` and `State`.

```java
public record City(String name, State state) {}
```

```java
public record State(String name) {}
```

Suppose you have a list of cities, and you need to compute the state that has the greatest number of cities. You can use t

```java
List<City> cities = List.of();

Map<State, Long> numberOfCitiesPerState =
    cities.stream()
          .collect(Collectors.groupingBy(
                   City::state, Collectors.counting()
          ));
```

Getting the max of this histogram is the following generic code.

```java
Map.Entry<State, Long> stateWithTheMostCities =
    numberOfCitiesPerState.entrySet().stream()
                          .max(Map.Entry.comparingByValue())
                          .orElseThrow();
```

This last piece of code is technical; it does not carry any business meanings; because is uses `Map.Entry` instance to moc

Using a local record can greatly improve this situation. The following code creates a new record class, that aggregates a s

Because you need to compare these aggregates by the number of cities, you can add a factory method to provide this co

```java
record NumberOfCitiesPerState(State state, long numberOfCities) {

    public NumberOfCitiesPerState(Map.Entry<State, Long> entry) {
        this(entry.getKey(), entry.getValue());
    }

    public static Comparator<NumberOfCitiesPerState> comparingByNumberOfCities() {
        return Comparator.comparing(NumberOfCitiesPerState::numberOfCities);
    }
}

NumberOfCitiesPerState stateWithTheMostCities =
    numberOfCitiesPerState.entrySet().stream()
                          .map(NumberOfCitiesPerState::new)
                          .max(NumberOfCitiesPerState.comparingByNumberOfCities())
                          .orElseThrow();
```

Your code now extracts a max in a meaningful way. Your code is more readable, easier to understand and less error-prone

More Learning

Last update: *January 5, 2024*

Back to Tutorial List