Object Oriented Programming in Java

Records

Suhel Hammoud

Records in Java

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 create a final class with final fields and a constructor to initialize these fields.

Records in Java

▒▒ example of an immutable data class.

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

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

Records in Java

Problem with Traditional Approach

- You need to add the accessors for your fields.
- Add a `toString()` method.
- Add `equals()` along with an `hashCode()` method.
- Consider making this class serializable if you need to send the data of this class over network or through a file system.

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

Records in Java

Calling Records to the Rescue

Records have been added to the JDK to change this. Starting with Java SE 14:

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

This single line of code creates:

- An immutable class with two fields: x and y
- A canonical constructor
- toString(), equals() and hashCode() methods
- Can implement Serializable

The Class of a Record

A record is class declared with the `record` keyword instead of `class`.

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

Key characteristics:

- The class is final
- Extends `java.lang.Record`
- Cannot extend any other class
- Can implement any number of interfaces

Declaring Record Components

The block that immediately follows the name of the record declares its components.

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

For each component:

- Private final field is created
- Accessor method is generated (e.g., `x()`)
- Default `toString()`, `equals()` and `hashCode()` are created

Restrictions on Records

Things you cannot add to a record:

1. Cannot declare any instance field not corresponding to a component
2. Cannot define any field initializer
3. Cannot add any instance initializer

You can:

- Create static fields with initializers
- Add static initializers

Constructing a Record

The compiler creates a canonical constructor that takes the components as arguments. Two ways to customize it:

1. Compact constructor (doesn't declare parameters)
2. Regular canonical constructor

Constructing a Record

▒▒ Example of compact constructor:

```
public record Range(int start, int end) {
    public Range {
        if (end <= start) throw new IllegalArgumentException(...);
    }
}
```

Constructing a Record

▓▓ Defining Additional Constructors: (that calls the canonical one)

```
public record State(String name, String capitalCity, List<String> 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));
  }
}
```

Accessor Methods

Records automatically generate accessor methods named after components.

▓▓▓ Example for Point:

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

You can override them if needed, for example to return defensive copies.

Serializing Records

Records can be serialized if they implement `Serializable`:

- Cannot customize serialization process (no `writeObject`/`readObject`)
- Deserialization always calls the canonical constructor
- Can use `readResolve()` and `writeReplace()`

This makes records excellent for data transport objects.