Java Record Classes-2024

* Java Record Classes are **by default immutable**. Removes boiler plate code.
* Record classes contains default equals, hashcode and toString methods.
* Record class contain mutable static fields.
* Mainly used for DTO objects or configuration settings.

However, if a record contains a mutable object, for example an ArrayList this object itself can be modified

public class Person {  
 private String name;  
 private int age;  
  
 public Person(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 @Override  
 public boolean equals(Object obj) {  
 *// Manually written equals() method*  
 }  
  
 @Override  
 public int hashCode() {  
 *// Manually written hashCode() method* }  
  
 @Override  
 public String toString() {  
 *// Manually written toString() method* }  
 *//getter methods*}

**The above can be written as using java record**

**public record Person(String name, int age) {  
 public Person() {  
 *// we must call the canonical constructor* this("Foo", 50);  
 }  
}**

**Records extend java.lang.Record, are final, and cannot be extended.**

**With Default Constructor**

public record Person(String name, int age) {  
 public Person() {  
 *// we must call the canonical constructor* this("Foo", 50);  
 }  
}

// Usage

public static void main(String[] args) {  
 Person p = new Person("John", 23);  
 System.*out*.println("Age: "+p.age());  
 System.*out*.println("Name: "+p.name());  
  
}

**How to Override Methods**

public record Person(String name, int age) {

// Automatically generated methods can be overridden

@Override  
public String toString() {  
 return String.*format*("Person{name='%s', age=%d}", name, age);  
}

}

**Records with Generics**

record Container<T>(int id, T value) {

}

Container<String> stringContainer = new Container<>(1, "1");

String strValue = stringContainer.value();

**How to Delegate Methods**

public record Person(String name, int age) {

public Person withName(String name) {

return new Person(name, age);

}

}

// Usage

Person person = new Person("Alice", 50); // Person[name=Alice, age=50]

Person newPerson = person.withName("Tom"); // Person[name=Tom, age=50]

**How to Implement Methods**

public record Person(String name, int age) {

public boolean isAdult() {

return age >= 18;

}

}

// Usage

Person person = new Person("Alice", 30);

System.out.println(person.isAdult()); // Output: true

**Serialization and Deserialization with Java Records**

**The serialVersionUID of a record class is 0L unless it is explicitly declared.**

public record Employee (...) implements Serializable { }

**Using Annotations in Records**

public record Employee(@Transient Long id, String firstName, String email,int age) {

// ...

}

**Composite Records**

public record Author(String name, String email) {}

public record Book(String title, Author author, int yearPublished) {

@Override

public String toString() {

return title + " by " + author.name() + " (" + yearPublished + ")";

}

}

public class Main {

public static void main(String[] args) {

Author author = new Author("George Orwell", "orwell@example.com");

Book book = new Book("1984", author, 1949);

// Displaying the book details

System.out.println(book);

}

}

**Local Records**

I see records have a very useful place when we just want to temporarily hold immutable data inside a function.

Let me explain this with an example.

public List<Person> sortPeopleByAge(List<Person> people) {

record Data(Person person, int age){};

return people.stream()

.map(person -> new Data(person, computeAge(person)))

.sorted((d1, d2) -> Double.compare(d2.age(), d1.age()))

.map(Data::person)

.collect(toList());

}