Java record explanation

give me the deep and detailed explaination of record in java so that i can able to understand it well and do not have any type of doubt in this.

so give me the deep and detailed explaination

In Java, a **record** is a *special kind of class* designed to hold data with **minimal boilerplate code**.

Think of it as Java saying:

> "You just want a simple class that stores some values, compares by those values, and prints nicely? Okay, I'll generate most of the code for you."

We'll go step by step, in **simple but deep detail** so that you don't have doubts.

---

1. What is a record in Java?

A **record** is a **final, immutable, data-carrier class** that:

- Has a **fixed set of fields** (called *components*).
- Automatically gets:
  - a constructor
  - `getter` methods (called accessors)
  - `equals()`
  - `hashCode()`
  - `toString()`

You write **very little code**, and the compiler does the boring part.

> ✅ Records were introduced as a standard feature in **Java 16**.
> You should be using Java 16+ (most commonly Java 17 LTS) to use them.

---

2. Basic syntax of a record

Normal class vs record

Normal class:

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

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

    public int getX() { return x; }
    public int getY() { return y; }

    @Override
    public boolean equals(Object o) { /* big boilerplate */ }

    @Override
    public int hashCode() { /* big boilerplate */ }

    @Override
    public String toString() { /* more boilerplate */ }
}
```

Same thing as a record:

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

Just this line and Java automatically generates:

- private final fields: `private final int x; private final int y;`
- constructor: `public Point(int x, int y) { ... }`
- accessors: `public int x()` and `public int y()`
- `equals()`, `hashCode()`, and `toString()` methods based on `x` and `y`.

---

3. Components: the “state” of a record

In a record:

```java
public record User(String name, int age) {}
```

- `String name`
- `int age`

are called **record components**.

These components **define the state** of the record.

For each component:

- A `private final` field is generated.

- A public accessor method with the **same name** is generated.

So:

```
java

User u = new User("Piyush", 21);
System.out.println(u.name()); // not getName(), just name()
System.out.println(u.age());  // not getAge(), just age()
```

There are **no setters** because records are **immutable** (explanation next).

---

4. Immutability in records (very important)

Records are designed to be **immutable data carriers**.

That means:

- Fields are `final` → you cannot reassign them after construction.
- You **cannot** do:

  ```
  java

  u.name = "New Name";  // compile-time error (field is private and final)
  ```

- There are no setters like `setName()`.

But there is a subtle point:

Shallow immutability (not deep)

```
java

public record MyData(List<String> items) {}
```

- The **reference** `items` is final → you cannot change it to point to another list.
- But the **list object itself** can still be modified:

  ```
  java

  List<String> list = new ArrayList<>();
  list.add("A");

  MyData data = new MyData(list);
  data.items().add("B");  // This is allowed! The list is still mutable.
  ```

So, records give **shallow immutability**:

- The references (fields) are final.
- But the objects they refer to may still be mutable.

If you want *true immutability*, use immutable types (e.g., `List.of(...)`) or make defensive copies.

5. What does the compiler generate for a record?

For:

```
java

public record User(String name, int age) {}
```

Roughly, the compiler generates something like this:

```
java

public final class User extends java.lang.Record {
    private final String name;
    private final int age;

    // Canonical constructor
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Accessors
    public String name() { return name; }
    public int age() { return age; }

    // equals(), hashCode(), toString() are generated
    @Override
    public boolean equals(Object o) { /* compares name and age */ }

    @Override
    public int hashCode() { /* based on name and age */ }

    @Override
    public String toString() {
        return "User[name=" + name + ", age=" + age + "]";
    }
}
```

Key points:

- The class is **implicitly final** (you can't extend a record).
- It **extends `java.lang.Record`** (you don't write this; compiler does).
- Fields are **private final**.
- **`equals()`** and **`hashCode()`** are **value-based**, i.e., compare components.

---

6. Constructors in records

Records support several types of constructors:

6.1. Canonical constructor (full-parameter constructor)

For:

```java
public record User(String name, int age) {}
```

The **canonical constructor** is:

```java
public User(String name, int age) {
    this.name = name;
    this.age = age;
}
```

You can **override** it if you want to add validation or modify values.

Example: validating input using canonical constructor

```java
public record User(String name, int age) {
    public User(String name, int age) {
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException("name cannot be blank");
        }
        if (age < 0) {
            throw new IllegalArgumentException("age cannot be negative");
        }
        this.name = name;
        this.age = age;
    }
}
```

6.2. Compact constructor (short syntax, very useful)

Instead of writing parameters again, you can use a **compact constructor**:

```java
public record User(String name, int age) {
    public User {
        // name and age are already available
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException("name cannot be blank");
        }
        if (age < 0) {
            throw new IllegalArgumentException("age cannot be negative");
        }
        // Implicit:
        // this.name = name;
        // this.age = age;
    }
}
```

Rules for compact constructor:

- You **don't write the parameter list**; it's taken from the record header.
- You **don't assign** fields manually (compiler adds the assignments at the end, unless you call another constructor).

- Used mainly for **validation or normalization**.

6.3. Overloaded constructors

You can also define **extra constructors**, but they must **delegate** to the canonical one using `this(...)`.

```
java

public record User(String name, int age) {

    // Extra constructor with default age
    public User(String name) {
        this(name, 18); // calls the canonical constructor
    }
}
```

---

7. Methods in records

Records can have:

- **Instance methods**
- **Static methods**
- **Static fields**
- **Static initializers**
- Nested types (e.g., inner classes, enums, other records).

Example:

```
java

public record User(String name, int age) {

    public String greeting() {
        return "Hi, I'm " + name + " and I'm " + age + " years old.";
    }

    public static User defaultUser() {
        return new User("Unknown", 0);
    }
}
```

You can also override the generated methods if needed:

```
java

public record User(String name, int age) {

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}
```

---

8. Inheritance rules for records

Important points:

1. A record:
   - **extends `java.lang.Record`** (implicitly).
   - **cannot extend any other class**.

```
java

public record Employee(String name, int id) extends SomeOtherClass {} // ❌ Not allowed
```

2. A record can **implement interfaces**:

```
java

public record Employee(String name, int id) implements Comparable<Employee> {

    @Override
    public int compareTo(Employee other) {
        return Integer.compare(this.id, other.id);
    }
}
```

3. A record is **implicitly final**, so it **cannot be subclassed**:

```
java

public record A(int x) {}
class B extends A { } // ❌ Not allowed
```

4. Records can be:
   - Top-level
   - Static nested records
   - Local records (inside a method)

```
java

public class Test {
    public void method() {
        record LocalRecord(int a, int b) {}
        LocalRecord r = new LocalRecord(1, 2);
    }
}
```

---

9. Restrictions on fields in records (very important)

In a record:

```
java

public record User(String name, int age) {}
```

- You **cannot** declare extra **instance fields** that are not part of the header.

This is **illegal**:

```
java

public record User(String name, int age) {
    private String address; // ❌ Not allowed (non-static field)
}
```

Why?
Because records are meant to have a well-defined, fixed set of components that represent all its state.

You **can** have:

- **Static fields**:

```
java

public record User(String name, int age) {
    public static final String DEFAULT_COUNTRY = "India";
}
```

- **Static methods**
- **Instance methods**

But **no extra non-static fields**.

---

10. How `equals`, `hashCode`, and `toString` work

For:

```
java

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

equals()

Two **`Point`** objects are equal if their **components are equal**:

```
java

Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);
Point p3 = new Point(2, 3);

System.out.println(p1.equals(p2)); // true
System.out.println(p1.equals(p3)); // false
```

It's **value-based equality**, not identity-based.

hashCode()

`hashCode()` is computed based on the components (**`x`** and **`y`**).
So:

```java
Set<Point> points = new HashSet<>();
points.add(new Point(1, 2));
System.out.println(points.contains(new Point(1, 2))); // true
```

Works properly for use in **`HashSet`**, **`HashMap`** keys, etc.

toString()

`System.out.println(new Point(1, 2));` prints something like:

```text
Point[x=1, y=2]
```

Useful for debugging/logging.

---

11. How to “modify” a record (since it's immutable)

Since you can’t change fields, the pattern is:

> Create a **new record instance** with modified values.

Example:

```java
public record User(String name, int age) {

    public User withAge(int newAge) {
        return new User(this.name, newAge);
    }
}

User u1 = new User("Piyush", 21);
User u2 = u1.withAge(22); // u1 is unchanged, u2 is new
```

This is similar to “with” methods used in many immutable patterns.

---

12. When to use a record vs a normal class

Use a record when:

- The class is mainly a **data container**.
- You care about **value equality** (two objects are equal if their fields are equal).
- Fields are **logically immutable**.
- Examples:
  - DTOs (Data Transfer Objects)
  - API responses

- Request models
- Value objects (`Point`, `Range`, `Money`, `Coordinates`, etc.)
- Keys in Maps (`UserId`, `ProductId`, etc.)

Avoid records when:

- You need **mutable state** (frequent changes to fields).
- Class has a lot of complex behavior, not just data.
- You need a **JPA entity** (they usually require a no-arg constructor, setters, proxies, etc.).
- You need **inheritance hierarchies** using normal classes.

---

13. Records vs Lombok `@Data` / plain POJO

Aspect	Record	Lombok `@Data` / POJO
Boilerplate	Minimal	Reduced with Lombok, but needs plugin
Mutability	Immutable (fields `final`)	Usually mutable (`setters` present)
Standard Java	Yes, built-in	Requires Lombok library
Extra instance fields	Not allowed	Allowed freely
Inheritance	Cannot extend other classes	Can extend classes

---

14. Pattern Matching & Records (brief idea)

One advanced use is **pattern matching** with records (Java 19+ features, etc.).

Example:

```
java

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

public void printPoint(Object obj) {
    if (obj instanceof Point(int x, int y)) {
        System.out.println("x = " + x + ", y = " + y);
    }
}
```

Here, Java “deconstructs” the record into its components.

This is more advanced, but now you know records integrate nicely with newer Java features.

---

15. Full example combining everything

```java
public record Student(String rollNo, String name, int age) {

    // Compact constructor for validation
    public Student {
        if (rollNo == null || rollNo.isBlank()) {
            throw new IllegalArgumentException("rollNo cannot be blank");
        }
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException("name cannot be blank");
        }
        if (age <= 0) {
            throw new IllegalArgumentException("age must be positive");
        }
    }

    // Extra constructor with default age
    public Student(String rollNo, String name) {
        this(rollNo, name, 18);
    }

    // Instance method
    public String display() {
        return "Student[" + rollNo + " - " + name + " (" + age + ")]";
    }

    // 'With' style method to create modified copy
    public Student withAge(int newAge) {
        return new Student(this.rollNo, this.name, newAge);
    }

    // Static factory method
    public static Student createDefault(String rollNo) {
        return new Student(rollNo, "Unknown", 18);
    }
}
```

Usage:

```java
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("CSE123", "Piyush", 21);
        System.out.println(s1);            // auto toString
        System.out.println(s1.display());  // custom method

        Student s2 = s1.withAge(22);       // new object, old unchanged
        System.out.println(s2);

        Student s3 = Student.createDefault("CSE999");
        System.out.println(s3);
    }
}
```

If you want, next I can:

- give you **small exercises**: “convert this normal class into a record”,
- or show you **how to use records in a Spring Boot project** (as DTOs in controllers/services).