



Java Records

Java Records (JEP 395)



Final since **Java 16** • “Nominal tuples” for immutable data

Java Records - Why Records? ❌➡️✅

POJOs for plain data = tons of boilerplate (ctor, accessors, equals/hashCode, toString)

IDEs generate code, but intent gets buried

Records: declare the **state**, get the **canonical API** for free

Java Records - The 1-liner vs. the Wall of Code

```
record Rectangle(double length, double width) {}
```

Equivalent to a final class with:

- private final fields
- public accessors length(), width()
- canonical constructor
- equals, hashCode, toString

Java Records - Using a Record

```
Rectangle r = new Rectangle(4, 5);
```

```
System.out.println(r.length() + " x " + r.width());
```

- Instances are created with new
- Access via accessors, not fields

Java Records - Constructors: Canonical & Compact

Canonical (explicit params):

```
record Rectangle(double length, double width) {  
    public Rectangle(double length, double width) {  
        if (length <= 0 || width <= 0) throw new IllegalArgumentException();  
        this.length = length;  
        this.width = width;  
    }  
}
```

Java Records - Constructors: Canonical & Compact

Compact (implicit params):

```
record Rectangle(double length, double width) {  
    public Rectangle {  
        if (length <= 0 || width <= 0) throw new IllegalArgumentException();  
    }  
}
```

👉 Compact = validate/normalize; assignments are **implicit**.

Java Records - What You Can/Can't Declare

✓ You can:


- `static` fields/methods/initializers
- instance methods
- nested types (incl. nested records — implicitly `static`)
- implement interfaces
- generics (`record Box<T>(T value) {}`)

✗ You can't:

- add instance (non-static) fields or instance initializers
- make it non-final or extend a class (super is `java.lang.Record`)
- declare `native` methods

Java Records - Overriding Accessors / equals / hashCode / toString

You **may** override, but keep invariants:

- Accessors must stay **public** and return the **component type**
- **equals/hashCode** must remain consistent with component equality
 Avoid “silent” transformations in accessors (breaks invariants).

Java Records - Validation & Normalization Patterns

Normalize in compact constructor:

```
record Rational(int num, int denom) {  
    public Rational {  
        int g = gcd(num, denom);  
        num /= g; denom /= g;  // normalized params  
    }  
}
```

Best practice: validate/normalize in ctor; keep accessors simple.

Java Records - Annotations on Components

```
@Retention(RUNTIME) @Target(FIELD) @interface GreaterThanZero {}  
  
record Rectangle(@GreaterThanZero double length,  
                 @GreaterThanZero double width) {}
```

- Component annotations propagate to field/ctor param/accessor when targets match
- Also supports type-use annotations

Java Records - Local & Nested Records

- **Local records** inside methods (implicitly `static`) improve readability near usage

```
record MerchantSales(Merchant merchant, double sales) {}
```

- Inner classes (since Java 16) can declare static members → includes records

Java Records - Records × Sealed × Pattern Matching

- Great with **sealed** hierarchies (ADTs): sealed interfaces + record implementations
- Plays nicely with **pattern matching for switch** (JEP 441)
(future/deconstruction patterns deepen this synergy)

Java Records - Serialization & Reflection

- Records are serializable; **components** govern serialization, **canonical constructor** governs deserialization (no custom `writeObject/readObject`)
- `Class::isRecord()`, `Class::getRecordComponents()` in reflection API

Java Records - Gotchas & Tips

- Name clash with your own `Record` class? Use fully-qualified imports.
- Records are **immutable by default**; for mutable structures, use classes.
- Don't smuggle extra state via outer instance — nested records are implicitly `static`.

Java Records - Key Takeaways 💡

- Records = concise, immutable data carriers with a standard API
- Focus on **state + invariants**; boilerplate handled by the compiler
- First-class citizens for modern Java: **Generics, Sealed types, Pattern matching**